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

数据：MNIST data set

• 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 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。

2. 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，draw.py
使用的数据集：mnist-original.mat
来源网站：https://www.kaggle.com/avnishnish/mnist-original/download
代码 github 网址：neuralnetwork/neuralnetwork/digitrec at main • piedpiperG/neuralnetwork (github.com)

2.2 数据集导入

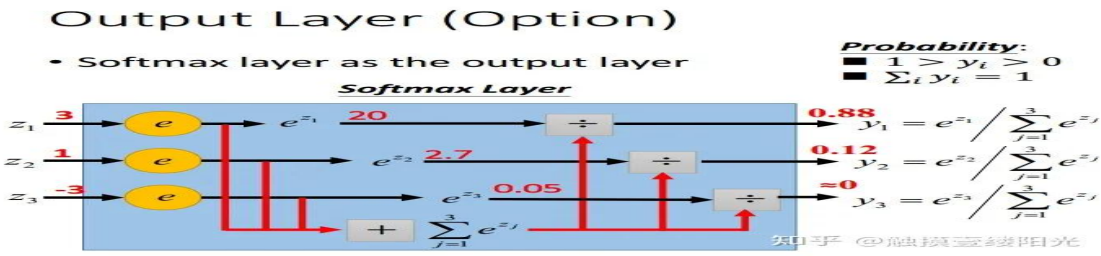
在 Softmax 和全连接神经网络分类器中都使用了相同的数据集导入方式，数据集来源于网站：https://www.kaggle.com/avnishnish/mnist-original/download
，下载到本地后为 mnist-original.mat 文件，使用 from scipy.io import loadmat 函数将其导入，并划分为 60000 个训练集和 10000 个测试集。其中 x 中每个元素为 28*28 的矩阵，代表数字图像像素；y 为数字图像像素对应的正确数字标签。

```
from scipy.io import loadmat
def loadFile():
    # 加载数据文件
    data = loadmat('mnist-original.mat')
    # 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
    x = data['data']
    x = x.transpose()
    # 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
    x = x / 255.0
    # 从数据中提取 labels，即 x 数字像素对应的数字类别 y
    y = data['label']
    y = y.flatten()
    # 将数据分割为 60,000 个训练集
    x_train = x[:60000, :]
    y_train = y[:60000]
    # 和 10,000 个测试集
    x_test = x[60000:, :]
    y_test = y[60000:]

    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：

$$\sigma(z)_i = \frac{e^{zi}}{\sum_{j=1}^{K} e^{zj}}$$

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

```
# 设置超参数
learning_rate = 0.1
num_epochs = 100
batch_size = 128
# 加载数据集
x_train, y_train, x_test, y_test = loadFile()
```

```
# 训练 softmax 分类器
train_softmax_classifier(x_train, y_train, x_test, y_test, learning_rate, num_epochs, batch_size)
```

（2）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。

```
def train_softmax_classifier(x_train, y_train, x_test, y_test, learning_rate, num_epochs, batch_size):
    input_size = x_train.shape[1]
    num_classes = len(np.unique(y_train))
    W, b = initialize_parameters(input_size, num_classes)

    m = x_train.shape[0]
    for epoch in range(num_epochs):
        for i in range(0, m, batch_size):
            x_batch = x_train[i:i + batch_size]
            y_batch = y_train[i:i + batch_size]
            y_batch = y_batch.astype(int)  # 强制转换为整数类型
            # 计算预测值
            scores = np.dot(x_batch, W) + b
            y_pred = softmax(scores)
            # 计算损失
            loss = cross_entropy_loss(y_batch, y_pred)
            # 计算梯度
            grad = y_pred
            grad[np.arange(x_batch.shape[0]), y_batch] -= 1
            grad /= batch_size

            dW = np.dot(x_batch.T, grad)
            db = np.sum(grad, axis=0)

            # 更新权重和偏置
            W -= learning_rate * dW
            b -= learning_rate * db

        # 在每个 epoch 结束后打印损失和准确率
        y_test_pred = softmax(np.dot(x_test, W) + b)
        test_loss = cross_entropy_loss(y_test, y_test_pred)
        test_accuracy = accuracy(y_test, y_test_pred)
        print(
```

```
            f"Epoch {epoch + 1}/{num_epochs}, Loss: {loss:.4f}, Test Loss: {test_loss:.4f}, Test Accuracy: {test_accuracy:.2%}")
```

（3）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。

```
# 定义 softmax 函数
def softmax(x):
    e_x = np.exp(x - np.max(x, axis=-1, keepdims=True))
    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。

```
# 初始化权重矩阵和偏置向量
def initialize_parameters(input_size, output_size):
    W = np.random.randn(input_size, output_size) * 0.001
    b = np.zeros(output_size)
    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。

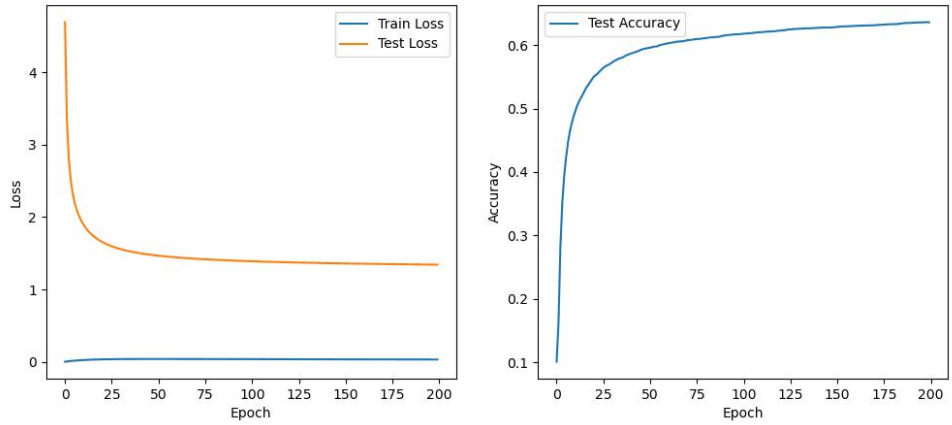
```
# 定义交叉熵损失函数
def cross_entropy_loss(y_true, y_pred):
    m = y_true.shape[0]
    y_true = y_true.astype(int)  # 强制转换为整数类型
    return -np.sum(np.log(y_pred[np.arange(m), y_true])) / m
```

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：

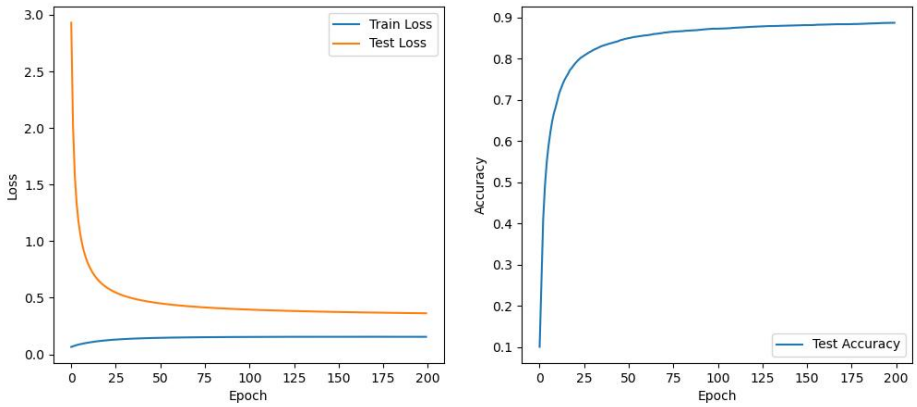
Epoch 200/200, Loss: 0.0343, Test Loss: 1.3429, Test Accuracy: 63.59%

绘出图像为：



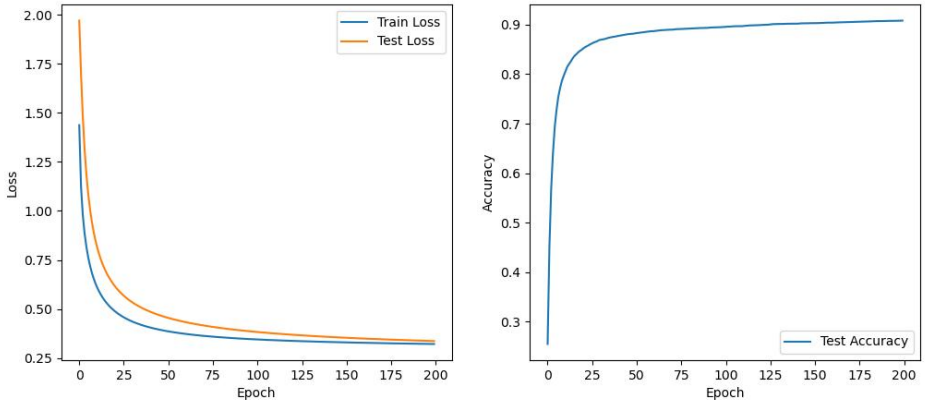
以学习率为 0.01，迭代次数为 200，批次为 128 得到的最终结果是：

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：

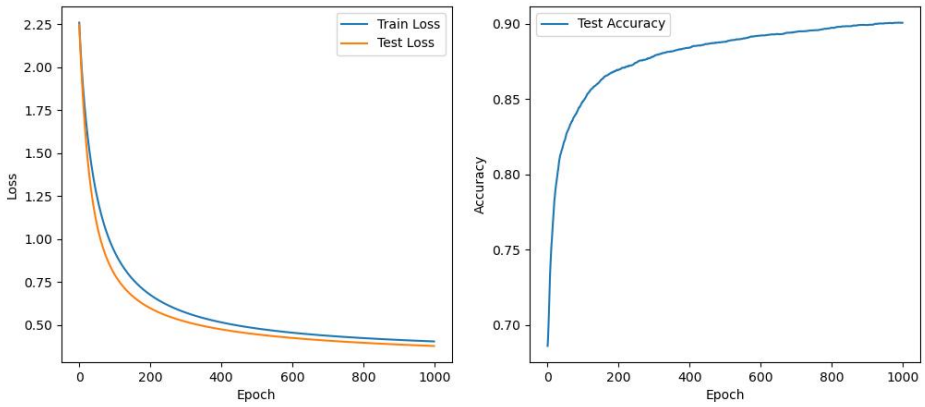
Epoch 200/200, Loss: 0.3221, Test Loss: 0.3371, Test Accuracy: 90.82%绘出图像为：



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。

```
# 输入层，隐藏层，输出层节点个数
    input_layer_size = 784  # 图片大小为 (28 X 28) px 所以设置 784 个特征
    hidden_layer_size = 16
    num_labels = 10  # 拥有十个标准为 [0, 9] 十个数字
    # 初始化层之间的权重 Thetas
    initial_Theta1 = initialise(hidden_layer_size, input_layer_size)  # 输入层和隐藏层之间的权重
    initial_Theta2 = initialise(num_labels, hidden_layer_size)  # 隐藏层和输出层之间的权重
    # 设置神经网络的参数
    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。

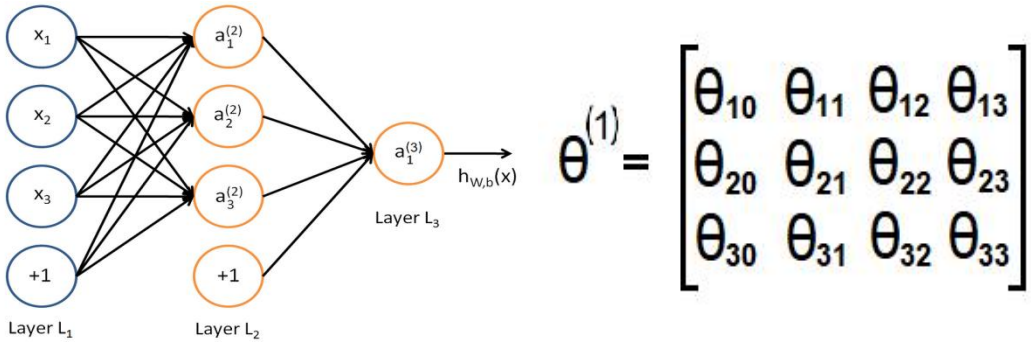
输出层的激活值表示每个数字类别的预测概率。

```
# 向前传播
    m = X.shape[0]
    one_matrix = np.ones((m, 1))
    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
    a1 = X
    z2 = np.dot(X, Theta1.transpose())
    a2 = 1 / (1 + np.exp(-z2))  # 采用 Sigmoid 函数对隐藏层进行激活
    one_matrix = np.ones((m, 1))
    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
    z3 = np.dot(a2, Theta2.transpose())
    a3 = 1 / (1 + np.exp(-z3))  # 采用 Sigmoid 函数对输出层进行激活
    # 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
    y_vect = np.zeros((m, 10))
    for i in range(m):
        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。

```
# 计算损失值
    J = loss(Theta1, Theta2, y_vect, a3, lamb, m)
    # 向后传播
    Delta3 = a3 - y_vect
    Delta2 = np.dot(Delta3, Theta2) * a2 * (1 - a2)
    Delta2 = Delta2[:, 1:]
    # 计算梯度
    Theta1[:, 0] = 0
    Theta1_grad = (1 / m) * np.dot(Delta2.transpose(), a1) + (lamb / m) * Theta1
    Theta2[:, 0] = 0
    Theta2_grad = (1 / m) * np.dot(Delta3.transpose(), a2) + (lamb / m) * Theta2
    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。

```
# 交叉熵损失（Cross-Entropy Loss）
J = (1 / m) * (np.sum(np.sum(-y_vect * np.log(a3) - (1 - y_vect) * np.log(1 - a3)))) + Reg
```

```
    均方误差损失（Mean Squared Error Loss）
    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。

```
# 正则化方法选择
    # L2 正则化
    L2_Reg = (lamb / (2 * m)) * (
            np.sum(np.square(Theta1[:, 1:])) + np.sum(np.square(Theta2[:, 1:])))
    # L1 正则化
    L1_Reg = (lamb / (2 * m)) * (
            np.sum(np.abs(Theta1[:, 1:])) + np.sum(np.abs(Theta2[:, 1:]))
    )
    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_rat
e, 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_rat
e, 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_rat
e, 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, alph
a_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_ra
te, 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。

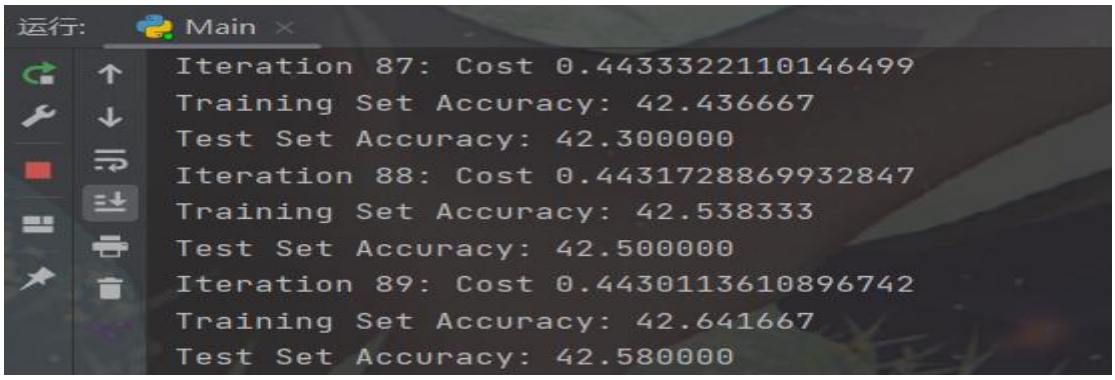
beta1 和 beta2 控制了动量项和学习率自适应项的权重。

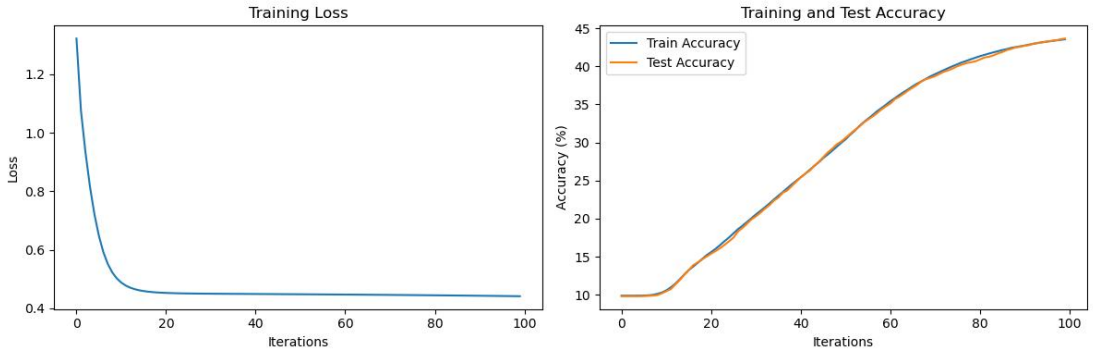
```
def Adamax(nn_params, input_layer_size, hidden_layer_size, num_labels, X, y, lambda_reg, iter_num, alpha_
rate, X_test,  y_test):
```

2.4.3 输出结果

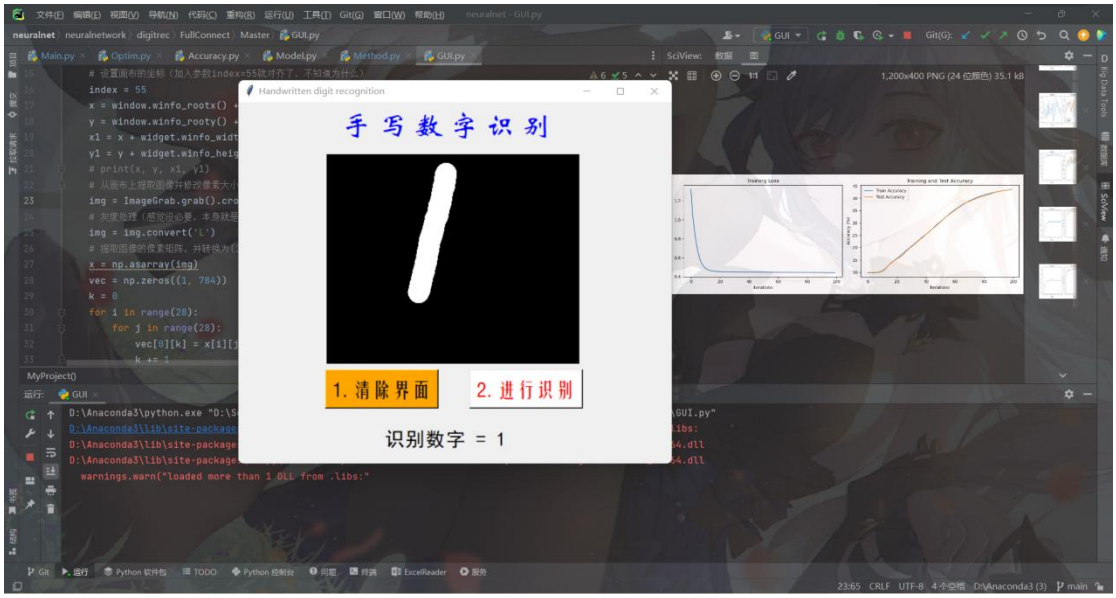
在每一次迭代中打印出当前迭代后的损失函数和关于训练集和测试集的准确率

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



3. 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。

```
# 加载数据文件
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。

```
    # 提取数据的特征矩阵，并进行转置
    X = data['data']
    X = X.transpose()
    # 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
    X = X / 255
    # 从数据中提取 labels
    y = data['label']
    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。

```
# 将数据分割为 60,000 个训练集
    train_size = 60000
    X_train = X[:train_size, :]
    y_train = y[:train_size]
    # 和 10,000 个测试集
    test_size = 10000
    X_test = X[train_size:train_size + test_size, :]
    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。

```
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。

```
def loss(Theta1, Theta2, y_vect, a3, lamb, m):
    # 计算损失值
    # 正则化方法选择
    # L2 正则化
    L2_Reg = (lamb / (2 * m)) * (
            np.sum(np.square(Theta1[:, 1:])) + np.sum(np.square(Theta2[:, 1:])))
    # L1 正则化
    L1_Reg = (lamb / (2 * m)) * (
            np.sum(np.abs(Theta1[:, 1:])) + np.sum(np.abs(Theta2[:, 1:]))
    )
    Reg = L2_Reg
    # 交叉熵损失（Cross-Entropy Loss）
    # J = (1 / m) * (np.sum(np.sum(-y_vect * np.log(a3) - (1 - y_vect) * np.log(1 - a3)))) + Reg
    # 均方误差损失（Mean Squared Error Loss）
    J = (1 / (2 * m)) * np.sum(np.square(a3 - y_vect)) + Reg
```

```
    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。

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

预测阶段：

模型加载：加载之前训练好的模型。

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

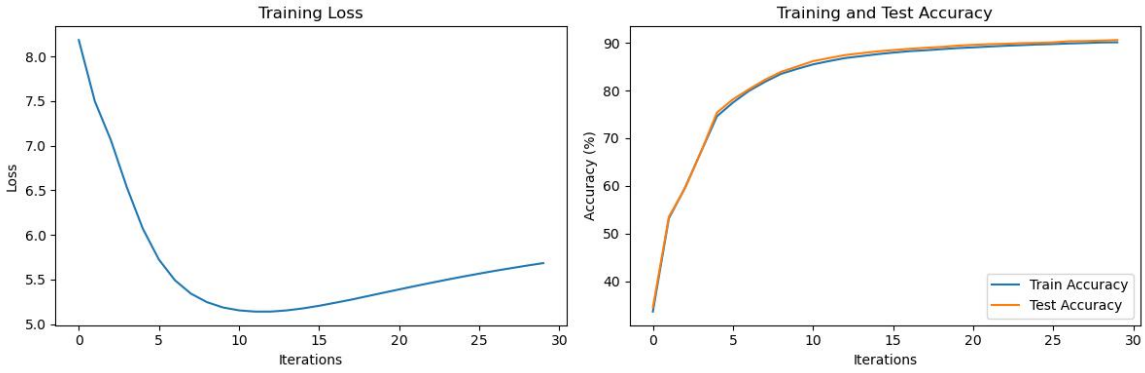
结果输出：将最终的识别结果呈现给用户或应用程序。

```
def plot_loss_and_accuracy(loss_history, train_accuracy_history, test_accuracy_history):
    # 绘制损失曲线
    plt.figure(figsize=(12, 4))
    plt.subplot(1, 2, 1)
    plt.plot(range(len(loss_history)), loss_history, label='Loss')
    plt.xlabel('Iterations')
    plt.ylabel('Loss')
    plt.title('Training Loss')

    # 绘制准确度曲线
    plt.subplot(1, 2, 2)
    plt.plot(range(len(train_accuracy_history)), train_accuracy_history, label='Train Accuracy')
    plt.plot(range(len(test_accuracy_history)), test_accuracy_history, label='Test Accuracy')
    plt.xlabel('Iterations')
    plt.ylabel('Accuracy (%)')
    plt.title('Training and Test Accuracy')
```

```
    plt.legend()  # 添加图例

    plt.tight_layout()
    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。

```
    # 寻找最佳超参数
    best_asem = []
    for alpha in [0.5, 0.1, 0.01, 0.001]:
        for lambda_reg in [0.01, 0.1, 1.0]:
            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

```
    # 验证集的准确度
    pred = predict(Theta1, Theta2, X_test)
    print(f'alpha={alpha},lambda_reg={lambda_reg},max_iter={max_iter}')
    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：

```
    alpha=0.5,lambda_reg=0.01,max_iter=10
```

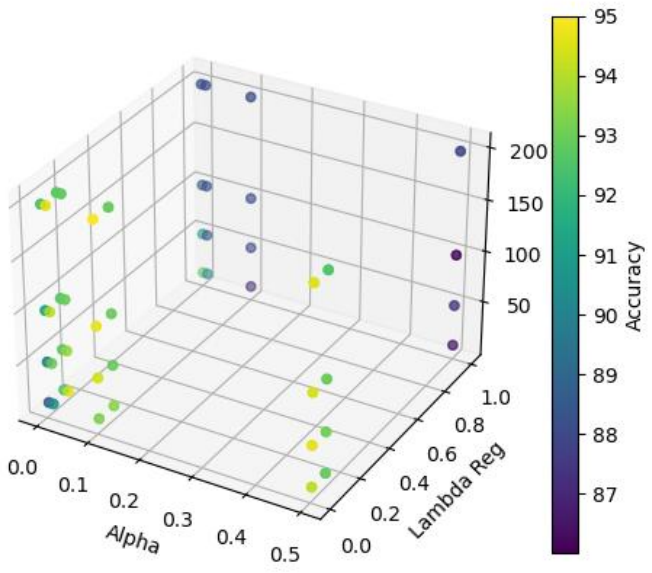
```
Accuracy: 94.090000
alpha=0.5,lambda_reg=0.01,max_iter=50
Accuracy: 94.630000
alpha=0.5,lambda_reg=0.01,max_iter=100
Accuracy: 94.430000
alpha=0.5,lambda_reg=0.01,max_iter=200
Accuracy: 94.560000
alpha=0.5,lambda_reg=0.1,max_iter=10
Accuracy: 92.850000
alpha=0.5,lambda_reg=0.1,max_iter=50
Accuracy: 93.100000
alpha=0.5,lambda_reg=0.1,max_iter=100
Accuracy: 92.970000
alpha=0.5,lambda_reg=0.1,max_iter=200
Accuracy: 92.600000
alpha=0.5,lambda_reg=1.0,max_iter=10
Accuracy: 86.640000
alpha=0.5,lambda_reg=1.0,max_iter=50
Accuracy: 87.190000
alpha=0.5,lambda_reg=1.0,max_iter=100
Accuracy: 86.010000
alpha=0.5,lambda_reg=1.0,max_iter=200
Accuracy: 87.960000
alpha=0.1,lambda_reg=0.01,max_iter=10
Accuracy: 93.220000
alpha=0.1,lambda_reg=0.01,max_iter=50
Accuracy: 94.390000
alpha=0.1,lambda_reg=0.01,max_iter=100
Accuracy: 94.710000
alpha=0.1,lambda_reg=0.01,max_iter=200
Accuracy: 95.000000
alpha=0.1,lambda_reg=0.1,max_iter=10
Accuracy: 93.370000
alpha=0.1,lambda_reg=0.1,max_iter=50
Accuracy: 92.910000
alpha=0.1,lambda_reg=0.1,max_iter=100
Accuracy: 92.960000
alpha=0.1,lambda_reg=0.1,max_iter=200
Accuracy: 92.900000
alpha=0.1,lambda_reg=1.0,max_iter=10
Accuracy: 87.170000
alpha=0.1,lambda_reg=1.0,max_iter=50
Accuracy: 87.920000
alpha=0.1,lambda_reg=1.0,max_iter=100
```

```
Accuracy: 88.190000
alpha=0.1,lambda_reg=1.0,max_iter=200
Accuracy: 88.090000
alpha=0.01,lambda_reg=0.01,max_iter=10
Accuracy: 90.600000
alpha=0.01,lambda_reg=0.01,max_iter=50
Accuracy: 93.020000
alpha=0.01,lambda_reg=0.01,max_iter=100
Accuracy: 94.110000
alpha=0.01,lambda_reg=0.01,max_iter=200
Accuracy: 94.660000
alpha=0.01,lambda_reg=0.1,max_iter=10
Accuracy: 94.600000
alpha=0.01,lambda_reg=0.1,max_iter=50
Accuracy: 93.700000
alpha=0.01,lambda_reg=0.1,max_iter=100
Accuracy: 92.980000
alpha=0.01,lambda_reg=0.1,max_iter=200
Accuracy: 92.820000
alpha=0.01,lambda_reg=1.0,max_iter=10
Accuracy: 89.760000
alpha=0.01,lambda_reg=1.0,max_iter=50
Accuracy: 88.190000
alpha=0.01,lambda_reg=1.0,max_iter=100
Accuracy: 88.140000
alpha=0.01,lambda_reg=1.0,max_iter=200
Accuracy: 88.170000
alpha=0.001,lambda_reg=0.01,max_iter=10
Accuracy: 88.500000
alpha=0.001,lambda_reg=0.01,max_iter=50
Accuracy: 89.950000
alpha=0.001,lambda_reg=0.01,max_iter=100
Accuracy: 91.360000
alpha=0.001,lambda_reg=0.01,max_iter=200
Accuracy: 92.430000
alpha=0.001,lambda_reg=0.1,max_iter=10
Accuracy: 92.480000
alpha=0.001,lambda_reg=0.1,max_iter=50
Accuracy: 92.650000
alpha=0.001,lambda_reg=0.1,max_iter=100
Accuracy: 92.630000
alpha=0.001,lambda_reg=0.1,max_iter=200
Accuracy: 92.710000
alpha=0.001,lambda_reg=1.0,max_iter=10
```

```
Accuracy: 92.640000
alpha=0.001,lambda_reg=1.0,max_iter=50
Accuracy: 90.940000
alpha=0.001,lambda_reg=1.0,max_iter=100
Accuracy: 88.850000
alpha=0.001,lambda_reg=1.0,max_iter=200
Accuracy: 88.260000
```

使用更直观的绘图方式可以得到，代码和图如下：

```
alphas = [item[0] for item in data]
lambda_regs = [item[1] for item in data]
max_iters = [item[2] for item in data]
accuracies = [item[3] for item in data]
# 创建一个三维散点图
fig = plt.figure()
ax = fig.add_subplot(111, projection='3d')
ax.scatter(alphas, lambda_regs, max_iters, c=accuracies, cmap='viridis', marker='o')
# 设置坐标轴标签
ax.set_xlabel('Alpha')
ax.set_ylabel('Lambda Reg')
ax.set_zlabel('Max Iter')
# 添加颜色条
cbar = plt.colorbar(ax.scatter(alphas, lambda_regs, max_iters, c=accuracies, cmap='viridis', marker='o'))
cbar.set_label('Accuracy')
# 显示图表
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。

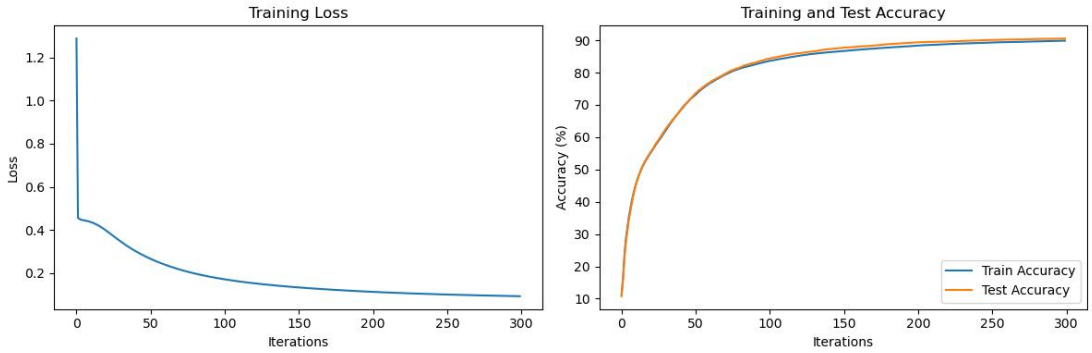
顺序而言，BGD 由于每次迭代是对整个数据集进行操作，不存在顺序。OGD 则是按照顺序更新，SGD 是将顺序打乱，而后进行更新。

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）:

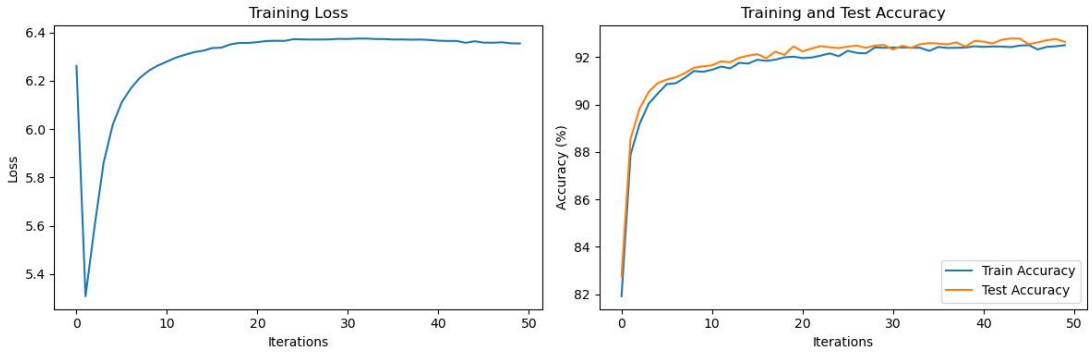
学习率=0.9，正则化参数=0.1，迭代次数=300



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

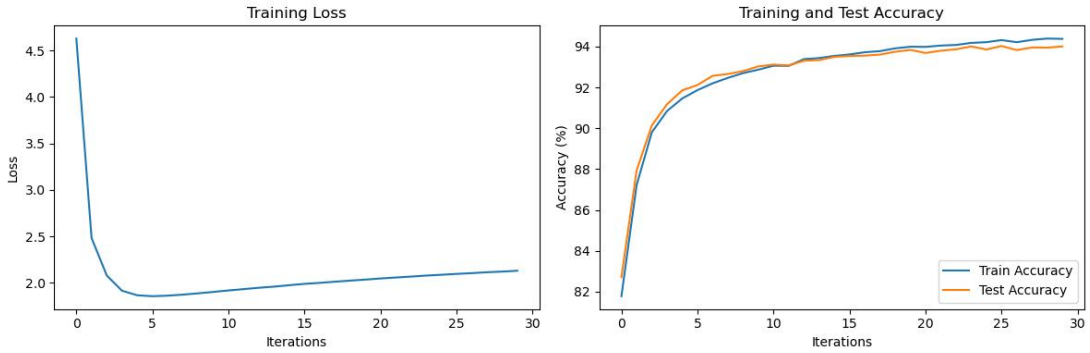
新的四种方法通过引入新的参数和技巧，对更新方式进行了优化，因此我在与原本小批量梯度下降方法的基础上，进行更改，得到了这四种新的方法。

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。



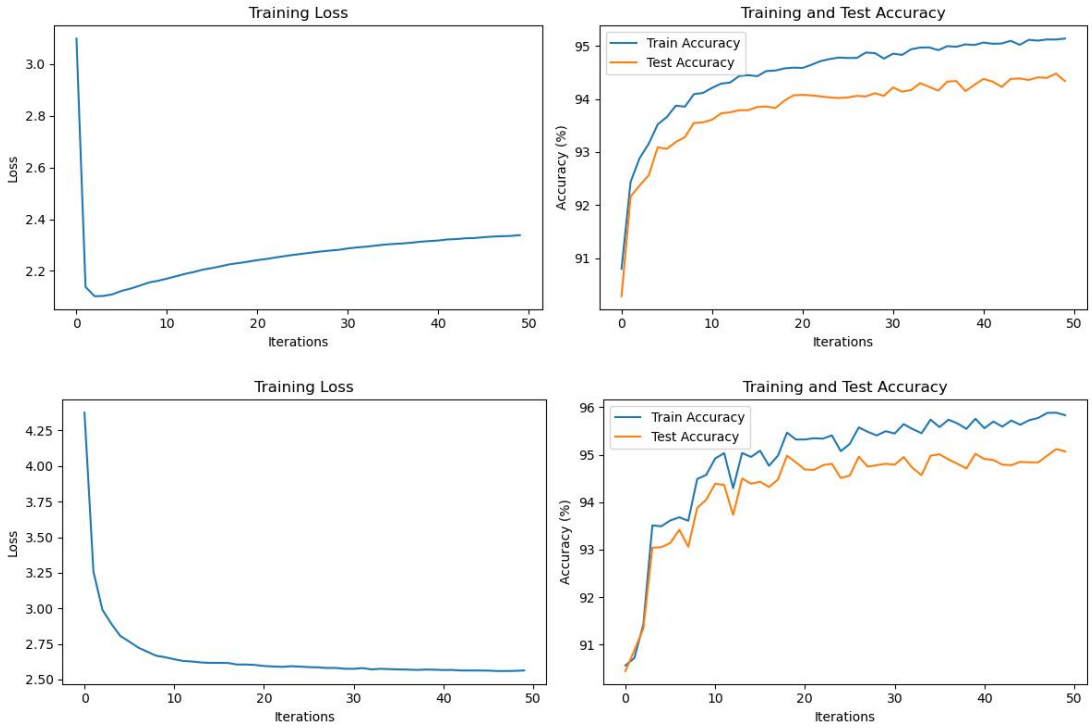
通过实验结果可以发现其收敛速度确实有了较大的提升，仅在 30 此循环以内就明显看到收敛趋势。

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。

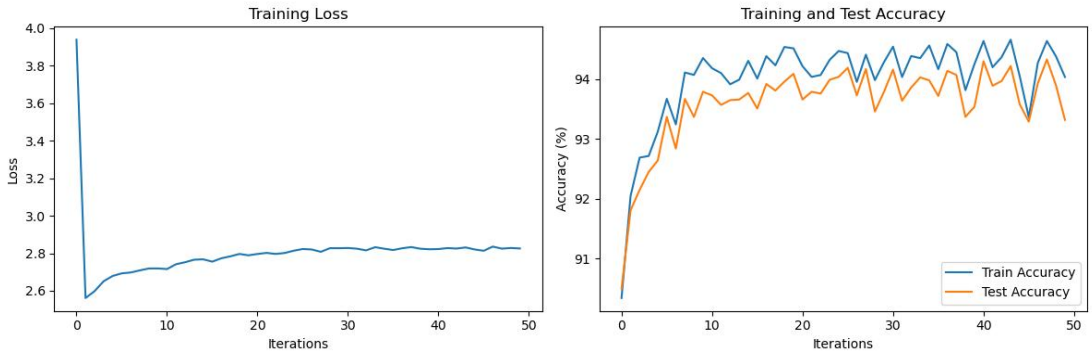


这种方法训练后测试集准确度明显低于训练集，可能是适用于较为稀疏的数据集，而本数据集相对密集，过拟合较为明显。

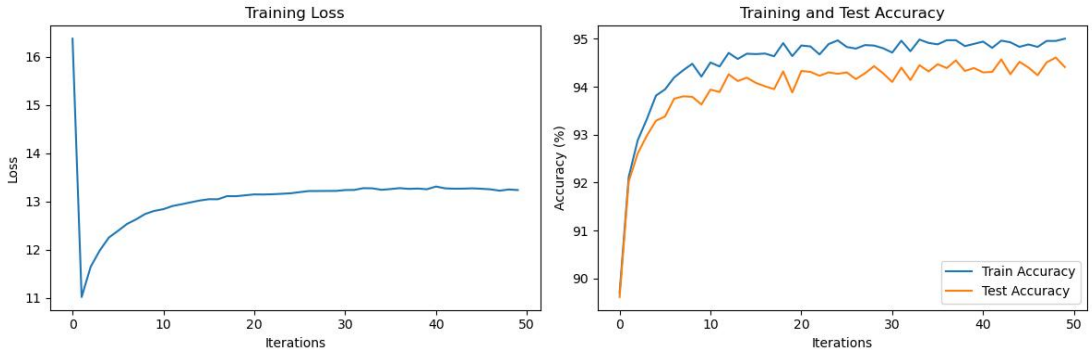
Adam (Adaptive Moment Estimation):

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。

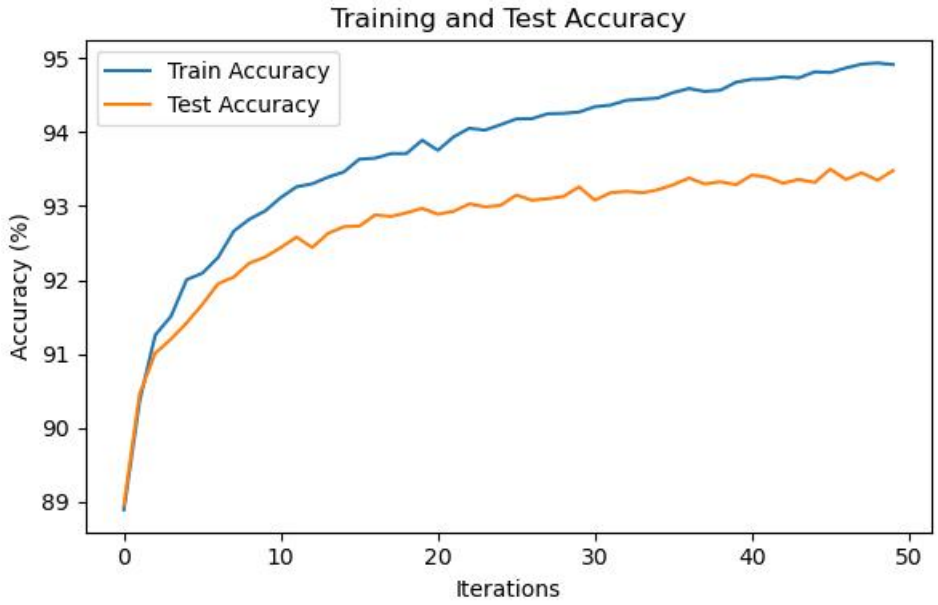


3.4 使用不同的损失函数和正则化方法

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。

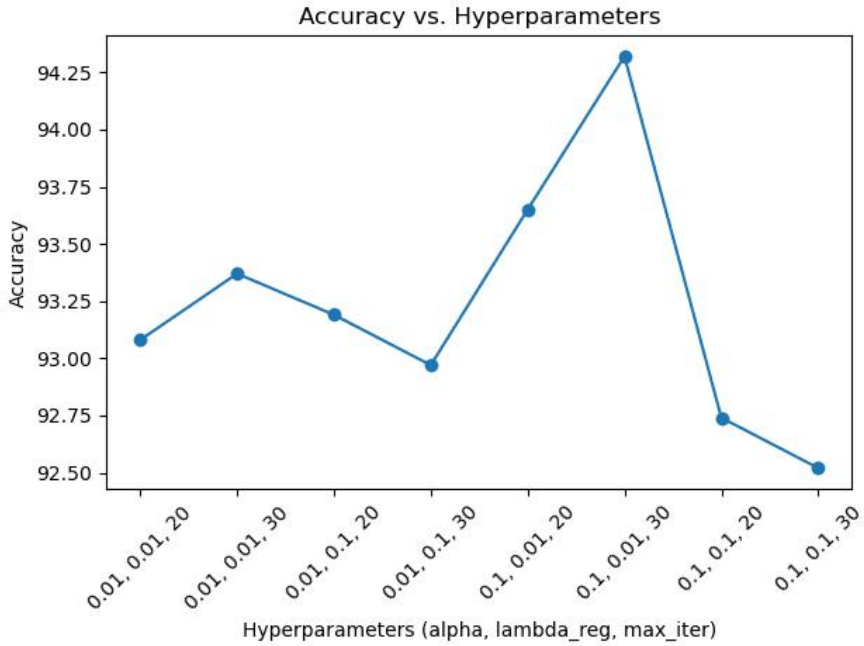
3.4.2 损失函数

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。

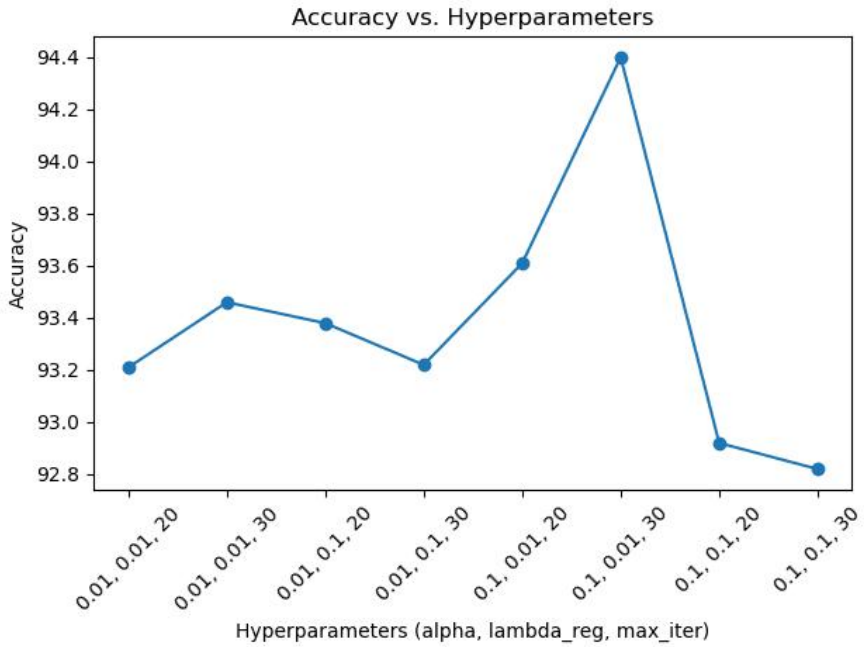
交叉熵损失（Cross-Entropy Loss）：通常用于分类问题，对于多类别分类问题表现良好。它惩罚分类错误的程度较大，因此在分类任务中通常能够更好地推动神经网络学习，特别是当输出类别是 one-hot 编码时。

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：



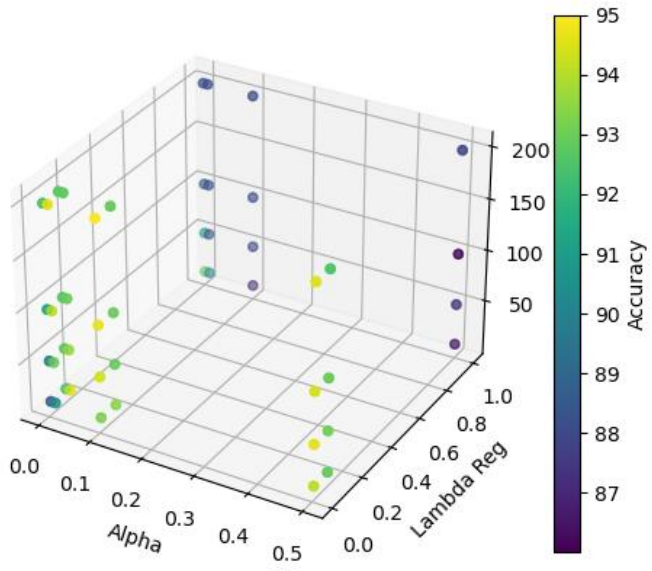
使用均方误差损失函数：



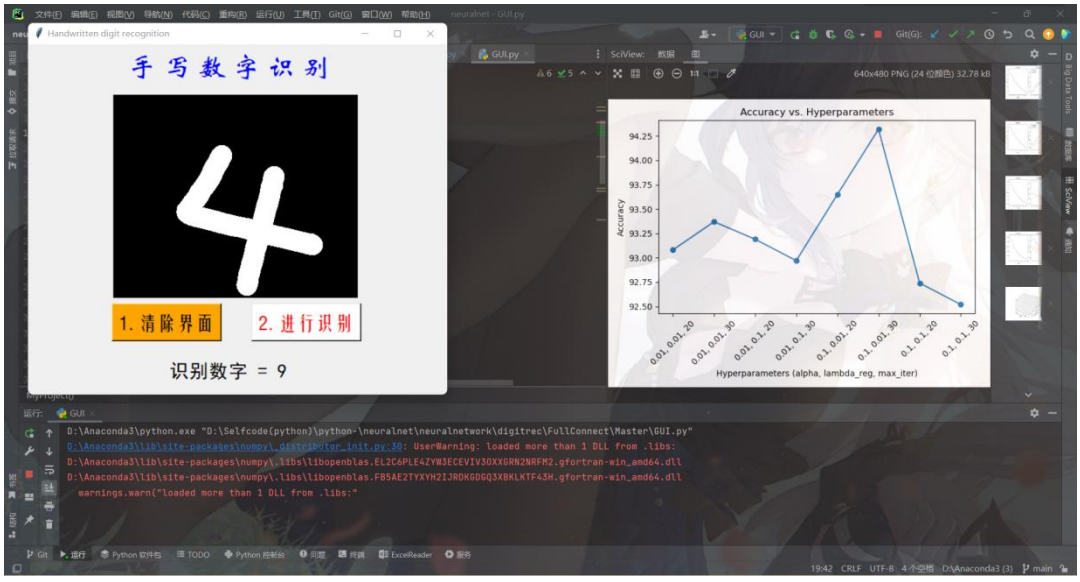
观察发现，使用两种损失函数得到的实验结果区别并不大，均方误差损失得到的准确度比使用交叉熵损失函数得到的准确度略高。

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...