实验报告：基于全链接网络的手写数字体识别

引言

手写数字体识别是计算机视觉领域中的一个重要问题，它对于自动化字符识别、光学字符识别、自动化标识等应用具有重要意义。本实验旨在设计并实现一个简单的图像分类器，使用基于全链接网络的方法对手写数字进行识别。我们将使用经典的MNIST数据集，其中包含了60000个训练样本和10000个测试样本。

基本的图像识别流程及数据驱动的方法

基本的图像识别流程通常包括以下几个阶段：数据准备、模型训练、模型评估和预测。下面是这些阶段的一般流程：

1. 数据准备阶段：

- 收集图像数据集：收集包含不同类别的图像数据集，每个类别都有相应的标签或类别标识。
- 数据预处理：对图像数据进行预处理，包括调整图像大小、归一化、增强对比度等操作，以提高模型训练的效果。
- 数据划分：将数据集划分为训练集、验证集和测试集，用于模型的训练、调优和评估。

2. 模型训练阶段：
   - 选择模型架构：选择适合图像识别任务的模型架构，如卷积神经网络（CNN）。
   - 初始化模型参数：初始化模型的权重和偏置参数。
   - 前向传播：将图像数据输入模型，通过前向传播计算得到预测结果。
   - 计算损失函数：将预测结果与真实标签进行比较，计算模型的损失函数。
   - 反向传播：根据损失函数，通过反向传播算法更新模型的参数，以减小损失函数的值。
   - 重复训练：重复执行前向传播、损失计算和反向传播，更新模型参数，直到达到预定的训练轮次或收敛条件。
3. 模型评估阶段：
   - 使用验证集评估模型性能：将验证集的图像输入训练好的模型，计算预测结果，并与真实标签比较，计算评估指标如准确率、精确率、召回率等。
   - 调整模型超参数：根据验证集的评估结果，调整模型的超参数，如学习率、正则化强度等，以提高模型性能。
4. 预测阶段：
   - 使用测试集进行最终评估：将测试集的图像输入训练好的模型，计算预测结果，并与真实标签比较，评估模型在未知数据上的性能。
   - 应用模型进行预测：使用训练好的模型对新的图像进行预测，得到图像的类别标签或预测概率。

数据预处理

在开始实验之前，我们首先从本地文件夹加载MNIST数据集（可以用其他方法加载该数据集，许多机器学习库和深度学习框架提供了内置函数或模块来加载MNIST数据集），然后对数据进行预处理。MNIST数据集中的手写数字被表示为28x28的向量，我们将其展开为一个784维的特征向量。此外，我们还需要对输入数据进行归一化，通常将像素值从0-255缩放到0-1的范围内。

```
xxx = "D:/software/Python/Hand_written_digits_recognition/data/MNIST/raw"
def load_labels(path):
    with open(path, 'rb') as f:
        labels = np.frombuffer(f.read(), dtype=np.uint8, offset=8)
    return labels

# 读取MNIST数据集
def load_mnist(path):
    with open(path, 'rb') as f:
        data = np.frombuffer(f.read(), dtype=np.uint8, offset=16)
    data = data.reshape(-1, 28*28)
    return data / 255.0

# 数据预处理
def preprocess_data():
    train_data = load_mnist(xxx+'/train-images-idx3-ubyte')
    train_labels = load_labels(xxx+'/train-labels-idx1-ubyte')
    test_data = load_mnist(xxx+'/t10k-images-idx3-ubyte')
    test_labels = load_labels(xxx+'/t10k-labels-idx1-ubyte')

    return train_data, train_labels, test_data, test_labels
```

此外，我们还需要把标签转换为one-hot形式

```
def build_labels(y):
    m = len(y)  # 样本数量
    num_classes = 10  # 类别数量
    matrix = np.zeros((m, num_classes))  # 创建零矩阵
    matrix[np.arange(m), y.astype(int)] = 1  # 根据索引设置对应位置的元素为1

    return matrix
```

独热编码是一种常用的表示分类问题标签的方法，它将每个类别表示为一个二进制向量，其中只有一个元素为1，其余元素为0。

在这段代码中，输入`y`是原始的标签数据，类型为一维数组。函数首先根据标签数据的长度确定样本数量`m`，然后根据类别数量`num_classes`创建一个`m * num_classes`的零矩阵。接下来，根据标签数据的值，将矩阵中对应位置的元素设置为1，其他位置保持为0，从而实现独热编码的转换。

数据划分

为了评估模型的性能，我们将原始训练集划分为训练集和验证集。通常情况下，我们将大约80%的数据用于训练，20%的数据用于验证。训练集用于模型的参数更新，验证集用于调整模型的超参数，如学习率、正则化参数等。

训练集、验证集和测试集的数据划分在机器学习和深度学习任务中具有重要意义。它们的作用如下：

1. 训练集（Training Set）：训练集用于训练模型的参数和权重。模型通过观察训练集中的样本和对应的标签来学习模式和特征。训练集通常是最大的数据集，用于模型的参数更新和优化。
2. 验证集（Validation Set）：验证集用于调整模型的超参数和评估模型的性能。在训练过程中，通过在验证集上评估模型的性能，可以选择不同的超参数设置、网络结构或其他模型配置，以提高模型的性能。验证集的目的是帮助选择最佳的模型配置，以在未见过的数据上获得更好的泛化能力。
3. 测试集（Test Set）：测试集用于最终评估训练好的模型在未知数据上的性能。测试集是模型从未见过的数据集，用于评估模型在实际应用中的泛化能力。测试集的结果可以用于评估模型的性能和进行模型之间的比较。

使用验证数据调整模型的超参数是一个迭代的过程，可以按照以下步骤进行：

1. 将训练集划分为训练集和验证集：将训练集分成两个部分，一部分用于模型的训练，另一部分用于验证模型的性能。
2. 定义模型的超参数空间：确定需要调整的超参数，例如学习率、批量大小、正则化参数等。
3. 选择一组超参数设置：从超参数空间中选择一组初始的超参数设置。
4. 在训练集上训练模型：使用训练集和选定的超参数设置来训练模型。
5. 在验证集上评估模型性能：使用验证集评估训练好的模型在不同超参数设置下的性能。可以使用一些评估指标，如准确率、损失函数等。
6. 调整超参数：根据在验证集上的性能，调整超参数的值。可以尝试不同的超参数组合，比较它们在验证集上的性能。
7. 重复步骤4-6：根据验证集的结果，反复迭代调整超参数，直到找到最佳的超参数设置。
8. 最终评估：使用测试集对最终选择的模型进行评估，以获得模型在未知数据上的性能。

激活函数及其梯度的实现

Softmax分类器

Softmax分类器是一种常用的多类别分类器，常用于机器学习和深度学习任务中。它是一种将输入数据映射到概率分布上的分类器，可以将输入数据分为多个不同的类别，并且每个类别的概率之和为1。

Softmax函数的数学公式如下：

$$Softmax(z_j) = \frac{e^{z_j}}{\sum_{i=0}^{n} e^{z_i}}$$

在机器学习和深度学习中，Softmax函数常用于多类别分类问题中，例如手写数字识别中的数字分类任务。

Softmax分类器常用于图像分类、自然语言处理等任务中，它可以输出每个类别的概率分布，从而提供更详细的分类结果。

下面是代码实现：

```
def softmax(x):
    exp_scores = np.exp(x)
    # softmax 对输出进行归一化 输出分类概率
    return exp_scores / (np.sum(exp_scores, axis=1, keepdims=True) + 1e-10)
```

在代码中，首先使用NumPy的exp函数对输入向量x进行指数运算，得到一个与x同维度的向量exp_scores。接着，使用np.sum函数对exp_scores进行求和，指定axis=1以计算每个样本的和，并通过keepdims=True保持维度一致性。

最后，将exp_scores除以求和结果，通过加上一个小的常数（1e-10）来避免除以零的情况。这样就得到了归一化后的概率分布，即Softmax函数的输出。

softmax求导

softmax 函数的导数 (或梯度)不是一个简单的闭合表达式，而是一个雅可比矩阵，因为个输入都会影响所有的输出。在实际应用中，通常不需要直接计算 Softmax 函数的导数。当使用交熵损失函数时，梯度的计算会因为 Softmax 和交叉熵的结合而变得非常简洁。

RELU

ReLU（Rectified Linear Unit）是一种常用的激活函数，广泛应用于深度学习模型中。它的定义很简单，对于输入x，ReLU函数的输出为：

ReLU(x) = max(0, x)

即，当输入x大于等于零时，输出为x；当输入x小于零时，输出为零。

ReLU激活函数的优点包括：

1. 非线性：ReLU函数在x大于零时是恒等映射，具有线性特性；在x小于零时输出为零，具有非线性特性。这种非线性特性使得ReLU能够学习和表示更加复杂的函数关系。
2. 稀疏激活：ReLU函数在输入为负时输出为零，这意味着它可以将负值抑制为零，从而使得神经网络中的神经元更加稀疏激活。这种稀疏性有助于减少模型的参数数量，提高计算效率，并且有助于防止过拟合。
3. 计算高效：ReLU函数的计算非常简单，只需要比较输入和零的大小即可，不涉及复杂的数学运算。这使得ReLU在深度神经网络中的前向传播和反向传播过程中计算效率高，加速了模型的训练和推断过程。

下面是代码实现：

```
def ReLU(z):
    # ReLU 激活函数
```

```
    return np.maximum(0, z)
```

RELU求导

ReLU函数的导数在不同的输入范围内有不同的取值：

$$ReLU'(x) = \begin{cases} 1 & \text{if } x > 0 \\ 0 & \text{if } x <= 0 \end{cases}$$

代码实现：

```
def relu_gradient(x):
    return np.where(x > 0, 1, 0)
```

Sigmoid

Sigmoid函数，也称为Logistic函数，是一种常用的激活函数，它将输入的实数映射到0和1之间的概率值。Sigmoid函数的定义如下：

$$sigmoid(x) = \frac{1}{1 + e^{-x}}$$

代码实现：

```
def sigmoid(x):
    # sigmoid 激活函数
    return 1 / (1 + np.exp(-x))
```

Sigmoid函数的特点如下：

1. 输出范围：Sigmoid函数的输出范围在0到1之间，可以看作是一个概率值，表示样本属于某个类别的概率。当输入x趋近于正无穷大时，Sigmoid函数的输出趋近于1；当输入x趋近于负无穷大时，Sigmoid函数的输出趋近于0。
2. 平滑性：Sigmoid函数在整个实数域上是连续且可导的，具有平滑的特性。这使得它在梯度下降等优化算法中可以进行反向传播，用于训练神经网络。
3. 非线性：Sigmoid函数是一种非线性函数，它的输出不是输入的简单线性变换。这种非线性特性使得神经网络能够学习和表示更加复杂的函数关系。

然而，Sigmoid函数也存在一些问题：

1. 饱和性：当输入x较大或较小时，Sigmoid函数的导数接近于零，导致梯度消失的问题。这使得在深度神经网络中，梯度传播可能会受到限制，导致训练困难。
2. 输出偏移：Sigmoid函数的输出不是以零为中心，而是以0.5为中心。这可能导致网络输出的分布不均衡，对某些优化算法产生不利影响。

sigmoid求导

$$sigmoid'(x) = \frac{e^{-x}}{(1 + e^{-x})^2} = sigmoid(x)^2 \cdot e^{-x}$$

代码实现:

```
def sigmoid_gradient(x):
    sig = sigmoid(x)
    return sig * sig * np.exp(-x)
```

tanh

tanh函数是双曲正切函数的缩写，也是一种常用的激活函数。它将输入映射到范围在-1到1之间的实数值。tanh函数的定义如下：

$$tanh(x) = \frac{e^x - e^{-x}}{e^x + e^{-x}}$$

```
def tanh(x):
    return (np.exp(x) - np.exp(-x)) / (np.exp(x) + np.exp(-x))
```

tanh函数的特点如下：

1. 输出范围：tanh函数的输出范围在-1到1之间，当输入x趋近于正无穷大时，tanh函数的输出趋近于1；当输入x趋近于负无穷大时，tanh函数的输出趋近于-1。
2. 平滑性：tanh函数在整个实数域上是连续且可导的，具有平滑的特性。这使得它在梯度下降等优化算法中可以进行反向传播，用于训练神经网络。
3. 非线性：tanh函数是一种非线性函数，它的输出不是输入的简单线性变换。这种非线性特性使得神经网络能够学习和表示更加复杂的函数关系。

与Sigmoid函数相比，tanh函数具有以下优点：

1. 零中心：tanh函数的输出以零为中心，即在输入为零时，函数的输出为零。这有助于减少输出分布的偏移，使得神经网络的学习更加稳定。
2. 动态范围：tanh函数的输出范围更广，相对于Sigmoid函数而言，tanh函数可以产生更大的激活值，有助于提高神经网络的表达能力。

然而，与Sigmoid函数类似，tanh函数也存在梯度饱和的问题。当输入的绝对值较大时，tanh函数的导数接近于零，导致梯度消失的问题。这可能影响深度神经网络的训练效果。

tanh求导

$$tanh'(x) = 1 - \left(\frac{e^x - e^{-x}}{e^x + e^{-x}}\right)^2 = 1 - tanh(x)^2$$

```
def tanh_derivative(x):
    return 1 - np.tanh(x) ** 2
```

Leaky ReLU

Leaky ReLU（Leaky Rectified Linear Unit）是ReLU的一种变体，用于解决ReLU函数中的死亡神经元问题。在Leaky ReLU中，对于输入x小于零的情况，不再将输出设为零，而是乘以一个小的斜率值。Leaky ReLU的定义如下：

$$LeakyReLU(x) = max(ax, x)$$

其中，a是一个小于1的斜率值，通常设置为0.01或0.2等较小的常数。

```
def leaky_relu(x):
    return np.maximum(0.01 * x, x)
```

Leaky ReLU函数的特点如下：

1. 非线性：Leaky ReLU函数在x大于零时与ReLU函数相同，保留了ReLU的非线性特性，使得神经网络能够学习和表示更加复杂的函数关系。
2. 避免死亡神经元：通过引入小的斜率值a，Leaky ReLU克服了ReLU函数中由于输入小于零导致的死亡神经元问题。即使输入小于零，Leaky ReLU也会有一个非零的输出，从而使得神经元能够更新权重。
3. 具有线性性质：在输入小于零的情况下，Leaky ReLU函数的输出与输入之间存在线性关系，相比于ReLU的截断性质，这种线性性质可能在某些情况下对模型的表达能力更有利。

Leaky ReLU求导

$$LeakyReLU'(x) = \begin{cases} 1 & \text{if } x > 0 \\ 0.01 & \text{if } x <= 0 \end{cases}$$

```
def leaky_relu_derivative(x, alpha=0.01):
    dx = np.ones_like(x)
    dx[x < 0] = alpha
    return dx
```

损失函数及其梯度的实现

交叉熵损失函数

交叉熵损失函数（Cross-Entropy Loss Function）是一种常用的损失函数，通常用于分类问题的训练中。它衡量了模型的预测结果与真实标签之间的差异，并用于优化模型参数。

对于二分类问题，交叉熵损失函数可以定义为：

$$L = -\frac{1}{m}\sum_{x\in D}\left(yln\left(\hat{y}\right) + \left(1-y\right)ln\left(1-\hat{y}\right)\right)$$

其中，y是真实标签（0或1），$\hat{y}$是模型预测的概率值（范围在0到1之间）。ln表示自然对数。

对于单标签多分类问题，交叉熵损失函数可以定义为：

$$L = -\frac{1}{m}\sum_{x\in D}yln\left(\hat{y}\right)$$

其中，y是一个真实标签向量，$\hat{y}$是一个模型预测概率向量，D表示训练集。

交叉熵损失函数的特点如下：

1. 目标一致性：交叉熵损失函数的目标是最小化预测结果与真实标签之间的差异，使得模型的预测结果更接近真实情况。

2. 梯度友好：交叉熵损失函数对于模型参数的梯度计算相对较简单，有利于使用梯度下降等优化算法进行模型的训练。
3. 高效性：交叉熵损失函数在分类问题中具有较好的效果，并且在实践中被广泛应用。

```
# 损失函数及其梯度
def cross_entropy_loss(Y_pred, Y_true):
    # 交叉熵损失函数
    m = Y_pred.shape[0]     # 样本数量
    log_probs = -np.log(Y_pred + 1e-10) * Y_true    # 修正概率矩阵
    loss = np.sum(log_probs) / m
    return loss
```

交叉熵和softmax求导

在多类分类问题中，Softmax 函数通常用于最后的输出层，将原始输出(即logits或scores转换为概率分布。softmax 函数的输出是一个概率分布，总和为1。
Softmax 函数的导数 (或梯度)不是一个简单的闭合表达式，而是一个雅可比矩阵，因为个输入都会影响所有的输出。

对于单个类的概率p，由 Softmax计算而来，其相对于输入z的导数是

$$\text{如果} i=j\text{，则}\frac{\partial p_i}{\partial z_j}=p_i\cdot\left(1-p_j\right),$$
$$\text{如果} i\neq j\text{，则}\frac{\partial p_i}{\partial z_j}=-p_i\cdot p_j$$

然而，在实际应用中，通常不需要直接计算 Softmax 函数的导数，当使用交熵损失函数时，梯度的计算会因为 Softmax 和交叉的结合而变得非常简洁。

Softmax与交叉熵损失的实现及求导 - 知乎 (zhihu.com)

当神经网络的最后一层激活函数为Softmax，并且损失函数为交叉熵时，求导的结果可以有一个简化形式。具体推导过程可以看上面的知乎文章

$$\frac{\partial Loss}{\partial x}=\hat{y}\ -y$$

其中，Loss为损失函数，x为softmax的输入矩阵，$\hat{y}$是预测的标签结果，为one-hot矩阵，y为真实标签。

```
def soft_cross_gradient(Y_pred, Y_true):
    '''交叉熵-softmax 导数'''
    return Y_pred - Y_true
```

均方误差损失函数

均方误差（Mean Squared Error，MSE）是一种常用的损失函数，用于衡量模型预测值与真实值之间的差异。

$$L=\frac{1}{m}\sum_{x\in D}\ \sum_i\frac{1}{2}\left(y_i-\hat{y}_i\ \right)^2$$

```
def mean_squared_error(Y_pred, Y_true):
    # 均方差损失函数
    m = Y_pred.shape[0]     # 样本数量
```

```
    loss = np.sum((Y_pred - Y_true) ** 2) / (2 * m)
    return loss
```

均方误差损失函数具有以下特点：

1. 平方项：均方误差损失函数对预测值与真实值之间的差异进行平方运算。这使得较大的差异被放大，较小的差异被缩小。平方项的引入使得损失函数对预测误差更加敏感。
2. 非负性：均方误差损失函数始终为非负值。这是因为平方项的结果始终为非负值。当预测值与真实值完全一致时，损失为0，否则损失大于0。
3. 全局性：均方误差损失函数考虑了所有样本的预测误差。通过对所有样本的误差进行求和或平均，均方误差损失函数提供了一个全局性的指标来衡量模型的性能。
4. 连续性：均方误差损失函数是一个连续可导的函数。这使得使用梯度下降等优化算法进行模型参数优化成为可能。通过计算损失函数关于参数的导数，可以确定参数的更新方向。
5. 异常值敏感性：均方误差损失函数对异常值（预测误差很大的样本）非常敏感。由于平方项的存在，异常值的平方误差会显著增加，从而对模型的训练产生较大的影响。这可能导致模型对异常值过度拟合。

需要注意的是，均方误差损失函数在某些情况下可能不适用。例如，在存在分类问题的情况下，如果使用均方误差损失函数来训练模型，可能会导致梯度消失的问题。在这种情况下，通常会选择适合分类问题的损失函数，例如交叉熵损失函数。

前向传播

一个隐藏层

```
self.W1 = np.random.randn(input_size, hidden_size) * 0.01   #  (784, hid_size)
self.b1 = np.zeros((1, hidden_size))                        # (1, hid_size)
self.W2 = np.random.randn(hidden_size, output_size) * 0.01  # (hid_size, 10)
self.b2 = np.zeros((1, output_size))                        # (1, 10)
```

```
def forward(self, X):
    self.z1 = np.dot(X, self.W1) + self.b1      #z1:   (m, h) x:(m, 784)
    self.a1 = self.activate(self.z1)  # ReLU 激活函数 (m, h)
    self.z2 = np.dot(self.a1, self.W2) + self.b2  # z2:   (m, 10)

    # softmax 对输出进行归一化，输出分类概率
    self.probs = softmax(self.z2)
```

在前向传播过程中，输入数据 `X` 的维度为 `(m, 784)`，其中 `m` 是样本数量，`784` 是输入特征的维度。具体步骤如下：

1. 线性变换1：计算隐藏层的线性变换，将输入数据 `X` 与隐藏层的权重 `self.W1` 相乘并加上偏置 `self.b1`，得到隐藏层的加权输入 `self.z1`。
2. 激活函数1：对隐藏层的加权输入 `self.z1` 应用ReLU激活函数，得到隐藏层的激活输出 `self.a1`。
3. 线性变换2：计算输出层的线性变换，将隐藏层的激活输出 `self.a1` 与输出层的权重 `self.W2` 相乘并加上偏置 `self.b2`，得到输出层的加权输入 `self.z2`。
4. Softmax函数：对输出层的加权输入 `self.z2` 应用Softmax函数，将其归一化为概率分布，得到输出层的分类概率 `self.probs`。这可以通过调用一个Softmax函数来实现。

最终，`self.probs` 表示输出层的分类概率，维度为 `(m, 10)`，其中 `m` 是样本数量，`10` 是输出层的类别数量。这样，前向传播过程就完成了，输出层的分类概率可以用于计算损失函数、进行预测等后续操作。

两个隐藏层

```
self.W1 = np.random.randn(input_size, hidden_size_1) * 0.01   # （784, hid_size_1)
self.b1 = np.zeros((1, hidden_size_1))                        # (1, hid_size_1)
self.W2 = np.random.randn(hidden_size_1, hidden_size_2) * 0.01  # (hid_size_1, h2)
self.b2 = np.zeros((1, hidden_size_2))                          # (1, h2)
self.W3 = np.random.randn(hidden_size_2, output_size) * 0.01    # (h2, 10)
self.b3 = np.zeros((1, output_size))                            # (1, 10)
```

```
def forward(self, X):
    self.z1 = np.dot(X, self.W1) + self.b1      #z1:   (m, h1) x:(m, 784)
    self.a1 = self.activate(self.z1)            # 激活函数 (m, h1)
    self.z2 = np.dot(self.a1, self.W2) + self.b2    # z2:   (m, h2)
    self.a2 = self.activate(self.z2)            # (m, h2)
    self.z3 = np.dot(self.a2, self.W3) + self.b3    # (m, 10)

    # softmax 对输出进行归一化，输出分类概率
    self.probs = softmax(self.z3)
```

在前向传播过程中，假设输入数据`X`的维度为`(m, 784)`，其中`m`是样本数量，`784`是输入特征的维度。具体步骤如下：

1. 线性变换1：计算第一个隐藏层的线性变换，将输入数据`X`与第一个隐藏层的权重`self.W1`相乘并加上偏置`self.b1`，得到第一个隐藏层的加权输入`self.z1`
2. 激活函数1：对第一个隐藏层的加权输入`self.z1`应用激活函数，得到第一个隐藏层的激活输出`self.a1`
3. 线性变换2：计算第二个隐藏层的线性变换，将第一个隐藏层的激活输出`self.a1`与第二个隐藏层的权重`self.W2`相乘并加上偏置`self.b2`，得到第二个隐藏层的加权输入`self.z2`
4. 激活函数2：对第二个隐藏层的加权输入`self.z2`应用激活函数，得到第二个隐藏层的激活输出`self.a2`
5. 线性变换3：计算输出层的线性变换，将第二个隐藏层的激活输出`self.a2`与输出层的权重`self.W3`相乘并加上偏置`self.b3`，得到输出层的加权输入`self.z3`
6. Softmax函数：对输出层的加权输入`self.z3`应用Softmax函数，将其归一化为概率分布，得到输出层的分类概率`self.probs`。这可以通过调用一个Softmax函数来实现

最终，`self.probs`表示输出层的分类概率，维度为`(m, 10)`，其中`m`是样本数量，`10`是输出层的类别数量。

反向传播

一个隐藏层

```
def backward(self, X, y, learning_rate):
    # X是训练数据(60000, 784) y是训练标签[5 0 4 ... 5 6 8]

    d = self.probs                      # 将模型的输出self.probs保存到变量d中

    if self.lossmodel == 1:
        # softmax激活函数 交叉熵损失函数
        d1 = soft_cross_gradient(d, y)      # (m, 10) 用简单方法求的梯度
    elif self.lossmodel == 2:
        # softmax激活函数 均方误差损失函数
        d1 = soft_mean_gradient(d, y)

    db2 = np.sum(d1, axis=0)
    dW2 = np.dot(d1.T, self.a1).T       # 计算输出层权重self.W2的梯度 (hid_size, 10)

    d2 = np.dot(d1, self.W2.T)          # (m, hid_size)
    d3 = d2 * self.activate_grad(self.z1)
    db1 = np.sum(d3, axis=0)            # 对矩阵的列求平均值 (1, hid_size)

    self.W1 -= learning_rate * dW1
    self.b1 -= learning_rate * db1
    self.W2 -= learning_rate * dW2
    self.b2 -= learning_rate * db2
```

首先，给定训练数据`X`和训练标签`y`，将模型的输出保存到变量`d`中。`self.probs`是模型的输出，表示预测的类别概率。

然后，根据`self.lossmodel`的取值，选择不同的损失函数类型进行梯度计算。

- 如果`self.lossmodel`等于1，表示使用交叉熵损失函数，使用`soft_cross_gradient(d, y)`计算梯度`d1`。在这个函数中，根据模型的输出`d`和训练标签`y`计算交叉熵损失函数关于模型输出的梯度。
- 如果`self.lossmodel`等于2，表示使用均方误差损失函数，使用`soft_mean_gradient(d, y)`计算梯度`d1`。在这个函数中，根据模型的输出`d`和训练标签`y`计算均方误差损失函数关于模型输出的梯度。

接下来，根据梯度`d1`计算输出层权重的梯度。`db2`表示输出层偏置`self.b2`的梯度，计算为对梯度`d1`按列求和。`dW2`表示输出层权重`self.W2`的梯度，计算为将梯度`d1`与隐藏层输出`self.a1`进行矩阵乘法，并转置结果。

然后，计算第二个隐藏层的梯度。将梯度`d1`与第二个隐藏层的权重`self.W2`进行矩阵乘法得到`d2`。然后，根据第一个隐藏层的加权输入`self.z1`的激活函数导数，将`d2`乘以`self.activate_grad(self.z1)`得到`d3`。最后，对矩阵`d3`的列求和得到第一个隐藏层的偏置`self.b1`的梯度`db1`。

最后，根据学习率`learning_rate`和对应的梯度，更新模型的权重和偏置。使用梯度下降的方法，将权重和偏置按照学习率乘以对应的梯度进行更新。

两个隐藏层

```
def backward(self, X, y, learning_rate):
        # X是训练数据(60000, 784) y是训练标签[5 0 4 ... 5 6 8]
        d = self.probs                      # 将模型的输出self.probs保存到变量d中

        if self.lossmodel == 1:
            # softmax激活函数 交叉熵损失函数
            d1 = soft_cross_gradient(d, y)
        elif self.lossmodel == 2:
            # softmax激活函数 均方误差损失函数
            d1 = soft_mean_gradient(d, y, self.z2)

        db3 = np.sum(d1, axis=0)
        dW3 = np.dot(self.a2.T, d1)         # 计算输出层权重self.W2的梯度 (10, h2)

        d2 = np.dot(d1, self.W3.T)          # (m, h2)
        d3 = d2 * self.activate_grad(self.z2)
        db2 = np.sum(d3, axis=0)            # 对矩阵的列求平均值 (1, h2)
        dW2 = np.dot(self.a1.T, d3)
        d4 = np.dot(d3, self.W2.T)
        d5 = d4 * self.activate_grad(self.z1)
        db1 = np.sum(d5, axis=0)

        dW1 = np.dot(X.T, d5)               # 计算输入层权重self.W1的梯度，即输入层误差delta2与输入数据X的转置矩阵的乘积

        self.W3 -= learning_rate * dW3
        self.b3 -= learning_rate * db3
        self.W1 -= learning_rate * dW1
        self.b1 -= learning_rate * db1
        self.W2 -= learning_rate * dW2
        self.b2 -= learning_rate * db2
```

这段代码实现了一个三层神经网络模型的反向传播过程。

首先，给定训练数据`X`和训练标签`y`，将模型的输出保存到变量`d`中。`self.probs`表示模型的输出，即预测的类别概率。

接下来，根据`self.lossmodel`的取值选择不同的损失函数类型进行梯度计算。

- 如果`self.lossmodel`等于1，表示使用交叉熵损失函数，使用`soft_cross_gradient(d, y)`计算梯度`d1`。这个函数根据模型输出`d`和训练标签`y`计算交叉熵损失函数关于模型输出的梯度。
- 如果`self.lossmodel`等于2，表示使用均方误差损失函数，使用`soft_mean_gradient(d, y, self.z2)`计算梯度`d1`。这个函数根据模型输出`d`、训练标签`y`和第二个隐藏层的加权输入`self.z2`计算均方误差损失函数关于模型输出的梯度。

然后，根据梯度`d1`计算输出层权重`self.W3`的梯度。`db3`表示输出层偏置`self.b3`的梯度，计算为对梯度`d1`按列求和。`dW3`表示输出层权重`self.W3`的梯度，计算为隐藏层输出`self.a2`的转置与梯度`d1`的矩阵乘法。

接着，计算第二个隐藏层的梯度。将梯度`d1`与第二个隐藏层的权重`self.W3`进行矩阵乘法得到`d2`。然后，根据第二个隐藏层的加权输入`self.z2`的激活函数导数，将`d2`乘以`self.activate_grad(self.z2)`得到`d3`。最后，对矩阵`d3`的列求和得到第二个隐藏层的偏置`self.b2`的梯度`db2`，并计算隐藏层之间权重`self.W2`的梯度`dW2`，即隐藏层输出`self.a1`的转置与`d3`的矩阵乘法。

最后，计算第一个隐藏层的梯度。将梯度`d3`与第一个隐藏层的权重`self.W2`进行矩阵乘法得到`d4`。然后，根据第一个隐藏层的加权输入`self.z1`的激活函数导数，将`d4`乘以`self.activate_grad(self.z1)`得到`d5`。最后，对矩阵`d5`的列求和得到第一个隐藏层的偏置`self.b1`的梯度`db1`，并计算输入层与第一个隐藏层之间权重`self.W1`的梯度`dW1`，即输入数据`X`的转置与`d5`的矩阵乘法。

最后，根据学习率`learning_rate`和对应的梯度，更新模型的权重和偏置。使用梯度下降的方法，将权重和偏置按照学习率乘以对应的梯度进行更新。

正则化方法

正则化方法在机器学习和统计建模中起着重要的作用，它的主要目的是控制模型的复杂度，以避免过拟合（Overfitting）问题。

当模型过度拟合训练数据时，它会过多地关注训练数据的细节和噪声，导致在新的未见过的数据上表现较差。正则化方法通过在损失函数中引入正则化项，可以对模型进行约束，防止模型过度拟合，并提高模型的泛化能力。

正则化方法通常会在损失函数中添加一个正则化项，该项用于衡量模型的复杂度。常见的正则化方法包括L1正则化和L2正则化。

1. L1正则化（L1 Regularization）：L1正则化通过在损失函数中加入模型参数的L1范数（参数绝对值之和）作为正则化项，可以促使模型参数稀疏化。L1正则化可以将不相关或冗余的特征的权重降低为零，从而使模型更加简洁。
2. L2正则化（L2 Regularization）：L2正则化通过在损失函数中加入模型参数的L2范数（参数平方和的平方根）作为正则化项，可以限制模型参数的大小。L2正则化对异常值不敏感，并可以平滑模型的权重，避免参数过大。

正则化方法的意义在于：

1. 控制过拟合：正则化方法通过约束模型的复杂度，避免模型过度拟合训练数据，从而提高模型在新数据上的泛化能力。
2. 特征选择：正则化方法在L1正则化中可以将不相关或冗余的特征的权重降低为零，从而实现特征选择，提高模型的解释性和可解释性。
3. 模型简化：正则化方法可以约束模型的复杂度，使得模型更加简洁和可解释。通过限制模型参数的大小或权重稀疏化，可以减少模型的复杂性。

L^2正则化

L^2正则化项的数学表示为：

$$\Omega\left(w\right)=\frac{1}{2}\parallel w\parallel_{2}^{2}=\frac{1}{2}\sum_{i}w_{i}^{2}$$

L^2正则化后的目标函数为

$$\tilde{L}=L+\frac{\theta}{2}\parallel w\parallel^{2}$$

目标函数对w求偏导得到

$$\bigtriangledown_{w}\tilde{L}=\bigtriangledown_{w}L+\theta w$$

以η 为步长，单步梯度更新权重为

$$w \leftarrow w-\eta\left(\bigtriangledown_w \tilde{L}+\theta w\right)$$

梯度更新中增加了权重衰减项θ。通过L^2正则化后，权重w成为梯度的一部分。权重w的绝对值变小，拟合的曲线就会变平滑，数据拟合得更好。

L^1正则化

L^1正则化的数学表示为：

$$\Omega\left(w\right)=\parallel w \parallel_1=\sum_i \mid w_i \mid$$

L^1正则化后的目标函数为

$$\tilde{L}=L+\theta \parallel w \parallel_1$$

目标函数对w求偏导得到

$$\bigtriangledown_w \tilde{L}=\bigtriangledown_w L+\theta sign\left(w\right)$$

以η 为步长，单步梯度更新权重为

$$w \leftarrow w-\eta\left(\bigtriangledown_w \tilde{L}+\theta sign\left(w\right)\ \right)$$

L^1正则化在梯度中加入了一个符号函数，当w为正数时，更新后的w会更小；当w为负数时，更新后的w会变大。因此正则化的效果是使w更接近0，即神经网络中的权重接近于0，从而减少过拟合。

结合两个正则化的损失函数

下面是修改后的损失函数：

```
# 损失函数正则化
def cross_entropy_loss_with_regularization(reg, Y_pred, Y_true, lambda_, *args):
    # 交叉熵损失函数（带正则化）
    m = Y_pred.shape[0]     # 样本数量
    log_probs = -np.log(Y_pred + 1e-10) * Y_true    # 修正概率矩阵
    cross_entropy_loss = np.sum(log_probs) / m

    regularization_term = 0
    if reg == 'L2':
        # 计算L2正则化项
        for weight_matrix in args:
            regularization_term += np.sum(np.square(weight_matrix))
    elif reg == 'L1':
        # 计算L2正则化项
        for weight_matrix in args:
            regularization_term += np.sum(np.abs(weight_matrix))
    else:
        regularization_term = 0

    # 添加正则化项到损失函数中
    loss = cross_entropy_loss + (lambda_ / (2 * m)) * regularization_term

    return loss
```

1. 函数定义：给定参数reg（正则化类型）、Y_pred（模型的预测值）、Y_true（真实标签）、lambda_（正则化参数）以及可变数量的*args参数（用于接收权重矩阵），定义了一个名为cross_entropy_loss_with_regularization的函数。
2. 样本数量：通过Y_pred的shape属性获取样本数量m。
3. 修正概率矩阵：为了避免计算log(0)的错误，将Y_pred加上一个小的常数（1e-10），然后将其与Y_true相乘，并取其负对数，得到修正后的概率矩阵log_probs。
4. 交叉熵损失函数：将修正后的概率矩阵log_probs的所有元素求和，并除以样本数量m，得到交叉熵损失cross_entropy_loss。
5. 正则化项初始化：初始化正则化项regularization_term为0。
6. 正则化类型判断：根据reg的取值（'L2'或'L1'），分别计算L2正则化项和L1正则化项。
7. L2正则化项计算：对于每个权重矩阵weight_matrix，将其平方后求和，将结果累加到regularization_term中。
8. L1正则化项计算：对于每个权重矩阵weight_matrix，将其绝对值求和，将结果累加到regularization_term中。
9. 正则化项添加：根据正则化类型，将正则化项添加到损失函数中。使用lambda_/(2*m)乘以正则化项，其中lambda_是正则化参数。
10. 返回结果：将交叉熵损失和正则化项相加得到最终的损失函数loss，并将其返回。

这段代码允许在计算交叉熵损失函数时添加L2或L1正则化项，以控制模型的复杂度并防止过拟合。正则化参数lambda_用于调节正则化的强度。

反向传播中需要添加与正则化对应的修改：

```
# 正则化
        if self.reg == 'L2':
            dW1 = (dW1 + self.lambda_* self.W1)
            dW2 = (dW2 + self.lambda_* self.W2)
        elif self.reg == 'L1':
            dW1 = (dW1 + self.lambda_* np.sign(self.W1))
            dW2 = (dW2 + self.lambda_* np.sign(self.W2))
```

这段代码实现了在反向传播过程中对权重矩阵进行正则化的操作。

1. 正则化类型判断：通过判断self.reg的取值（'L2'或'L1'），确定要应用的正则化类型。
2. L2正则化：如果self.reg为'L2'，则对权重矩阵进行L2正则化。对于权重矩阵W1，将其与self.lambda_（正则化参数）和self.W1相乘，得到正则化项self.lambda_* self.W1，然后将其加到对应的梯度dW1上。同样地，对于权重矩阵W2，将其与self.lambda_和self.W2相乘，得到正则化项self.lambda_* self.W2，然后将其加到对应的梯度dW2上。
3. L1正则化：如果self.reg为'L1'，则对权重矩阵进行L1正则化。对于权重矩阵W1，将其与self.lambda_和np.sign(self.W1)（W1的元素的符号）相乘，得到正则化项self.lambda_* np.sign(self.W1)，然后将其加到对应的梯度dW1上。同样地，对于权重矩阵W2，将其与self.lambda_和np.sign(self.W2)相乘，得到正则化项self.lambda_* np.sign(self.W2)，然后将其加到对应的梯度dW2上。

优化算法

反向传播优化算法，是在基础的梯度下降算法上进行改进，优化梯度下降过程中的步长以及梯度，以达到增加反向传播效率的算法

梯度下降法（Gradient Descent）

梯度下降是一种基本的优化算法，根据损失函数关于参数的梯度方向来更新参数。它可以分为批量梯度下降（Batch Gradient Descent）、随机梯度下降（Stochastic Gradient Descent）和小批量梯度下降（Mini-Batch Gradient Descent）等不同的变体

批量梯度下降（Batch Gradient Descent）

批量梯度下降是梯度下降算法的一种形式，它在每一次参数更新时使用全部训练数据计算损失函数关于参数的梯度。下面是批量梯度下降的主要特点和步骤：

1. 特点：
    - 使用全部训练数据：批量梯度下降在每一次参数更新时，使用全部的训练数据计算损失函数关于参数的梯度，因此需要将所有训练样本一起输入模型进行前向传播和反向传播计算。
    - 精确的梯度：由于使用了全部训练数据，计算得到的梯度是精确的，没有采样误差。
    - 计算开销大：由于涉及到所有训练样本的计算，批量梯度下降的计算开销通常较大，特别是在处理大规模数据集时。
2. 步骤：
    - 初始化参数：随机初始化模型的参数。
    - 前向传播：使用当前参数对所有训练样本进行前向传播，计算模型的输出。
    - 计算损失函数：根据模型的输出和训练样本的真实标签，计算损失函数的值。
    - 反向传播：通过反向传播算法计算损失函数对所有参数的梯度。
    - 参数更新：使用梯度信息按照梯度下降的规则更新模型的参数。
    - 重复迭代：重复执行前面的步骤，直到达到停止条件（如达到最大迭代次数或损失函数收敛）。

批量梯度下降的主要优点是可以获得全局最优解（如果存在），并且在参数更新时具有确定性。然而，由于需要计算所有训练样本的梯度，批量梯度下降的计算开销较大，尤其是在处理大规模数据集时。此外，批量梯度下降对于处理非凸优化问题时可能会陷入局部最优解

```
def BGD(self, X, y, num_epochs, learning_rate):
        '''批量梯度下降 Batch Gradient Descent'''
        for epoch in range(num_epochs):
            self.forward(X)
            self.update_loss_epochs(y)
            self.backward(X, y, learning_rate)
            self.epoch += 1
```

小批量梯度下降（Mini-Batch Gradient Descent）

小批量梯度下降（Mini-Batch Gradient Descent）是梯度下降算法的一种变体，它在每次参数更新时使用一小批（mini-batch）训练数据来计算损失函数关于参数的梯度。下面是小批量梯度下降的主要特点和步骤：

特点：

- 使用小批量数据：小批量梯度下降在每一次参数更新时，使用一个小批量的训练数据计算损失函数关于参数的梯度。小批量大小通常是一个可调节的超参数，一般选择的大小在几十到几千之间。
- 近似的梯度：由于使用了一个小批量的数据来计算梯度，所得到的梯度是对整体样本的梯度的近似。这可以降低计算开销，同时仍然能够提供较好的梯度估计。

```
def MBGD(self, X, y, num_epochs, learning_rate, batch_):
        '''小批量梯度下降 Mini-Batch Gradient Descent MBGD'''
        self.start = time.time()
        # 将训练集随机打乱顺序
```

```
        permutation = np.random.permutation(X.shape[0])
        X_train_shuffled = X[permutation]
        y_train_shuffled = y[permutation]

        # 分割训练集为小批量 batch = 1为随机梯度下降SGD
        batch = batch_
        for epoch in range(num_epochs):
            for i in range(0, X.shape[0], batch):
                # 获取当前批量的输入和标签
                X_batch = X_train_shuffled[i:i + batch]
                y_batch = y_train_shuffled[i:i + batch]
                self.forward(X_batch)
                self.backward(X_batch, y_batch, learning_rate)
```

1. 定义函数：
    - MBGD(self, X, y, num_epochs, learning_rate, batch_): 定义一个名为MBGD的方法，接受输入数据X和对应的标签y，指定训练的轮数（num_epochs）、学习率（learning_rate）和小批量大小（batch_）。
2. 随机打乱训练数据：
    - permutation = np.random.permutation(X.shape[0]): 生成一个随机的排列索引，用于将训练数据的顺序打乱。
    - X_train_shuffled = X[permutation] 和 y_train_shuffled = y[permutation]: 根据生成的随机索引，将输入数据和标签按照同样的顺序进行打乱。
3. 小批量梯度下降迭代训练：
    - for epoch in range(num_epochs): 对于指定的训练轮数，进行迭代。
    - for i in range(0, X.shape[0], batch): 对于每个小批量数据的起始索引，进行迭代。
    - X_batch = X_train_shuffled[i:i + batch] 和 y_batch = y_train_shuffled[i:i + batch]: 根据起始索引，获取当前批量的输入数据和标签。
    - self.forward(X_batch): 调用模型的前向传播方法，计算当前批量数据的输出。
    - self.backward(X_batch, y_batch, learning_rate): 调用模型的反向传播方法，计算当前批量数据的梯度并更新模型的参数，使用给定的学习率进行参数更新。

随机梯度下降（Stochastic Gradient Descent）

随机梯度下降（Stochastic Gradient Descent，SGD）是梯度下降算法的一种变体，它在每次参数更新时使用单个样本来计算损失函数关于参数的梯度。与批量梯度下降和小批量梯度下降不同，SGD不使用批量或小批量的数据，而是选择一个样本来计算梯度。下面是随机梯度下降的主要特点和步骤：

特点：

- 使用单个样本：随机梯度下降在每一次参数更新时，使用一个单独的训练样本来计算损失函数关于参数的梯度。
- 随机性：由于每次只使用一个样本，所得到的梯度是对整体样本梯度的估计。这种随机性使得SGD在参数更新时具有一定的不确定性，有助于跳出局部最优解，但也可能引入一些噪音。
- 计算开销较小：由于只使用一个样本，SGD的计算开销相对较小，尤其是在处理大规模数据集时。

随机梯度下降的代码实现与上面相同，batch_=1即可

动量法（Momentum）

动量法在梯度下降的基础上引入了动量的概念，通过累积之前的梯度信息来决定参数更新的方向和幅度。它可以加速收敛速度，并且对于参数空间中存在的局部最优解有一定的跳出能力。

1. 特点：

- 考虑历史梯度：动量法不仅使用当前的梯度信息，还考虑了之前的梯度信息。它引入了一个动量项，用于累积历史梯度的影响。
- 平滑参数更新：通过动量项的引入，参数更新的方向更加稳定，并且在梯度方向改变时，具有一定的惯性，从而减少参数更新的震荡。
- 加速收敛：动量法可以加速模型的收敛速度，特别是在存在平坦区域或局部最优解附近时，可以帮助跳出局部最优解。

2. 步骤：
   - 初始化参数：随机初始化模型的参数。
   - 初始化动量：设置初始动量为0。
   - 迭代更新：对于每个训练样本，进行以下步骤：
     - 前向传播：使用当前参数对该训练样本进行前向传播，计算模型的输出。
     - 计算损失函数：根据模型的输出和训练样本的真实标签，计算损失函数的值。
     - 反向传播：通过反向传播算法计算损失函数对参数的梯度。
     - 更新动量：根据当前梯度和之前的动量，更新动量的值。一般使用指数加权平均来计算动量项。
     - 参数更新：使用动量信息按照梯度下降的规则更新模型的参数。参数更新的大小由学习率和动量共同决定。
   - 重复迭代：重复执行前面的步骤，直到达到停止条件（如达到最大迭代次数或损失函数收敛）。

动量法通过累积历史梯度的影响，使得参数更新具有惯性和稳定性。在训练过程中，动量法能够在参数更新时减少震荡，更快地向全局最优解或局部最优解移动，从而加速模型的收敛。动量法是一种常用的优化算法，在深度学习中广泛应用。

$$
\begin{aligned}
v_t &= \eta v_{t-1} + \eta \nabla_\theta F(\theta) \\
\theta &\leftarrow \theta - v_t
\end{aligned}
$$

η是动量因子，一般性的经验值为0.9

```
# 定义动量变量
self.velocity_W1 = np.zeros_like(self.W1)
self.velocity_b1 = np.zeros_like(self.b1)
self.velocity_W2 = np.zeros_like(self.W2)
self.velocity_b2 = np.zeros_like(self.b2)
# 动量值 momentum=0 时无优化
self.momentum = momentum
```

```
if self.momentum != 0:
    # momentum 优化
    # 更新动量变量
    self.velocity_W1 = self.momentum * self.velocity_W1 - learning_rate * dW1
    self.velocity_b1 = self.momentum * self.velocity_b1 - learning_rate * db1
    self.velocity_W2 = self.momentum * self.velocity_W2 - learning_rate * dW2
    self.velocity_b2 = self.momentum * self.velocity_b2 - learning_rate * db2

    self.W1 += self.velocity_W1
    self.b1 += self.velocity_b1
    self.W2 += self.velocity_W2
    self.b2 += self.velocity_b2
```

在这段代码中，`self.momentum`是动量因子（momentum factor），它决定了历史梯度对参数更新的影响程度。下面是对代码进行逐行解释：

1. 更新动量变量：
   - `self.velocity_W1 = self.momentum * self.velocity_W1 - learning_rate * dW1`：根据动量法的更新规则，计算参数W1的动量变量。该变量是之前动量变量的衰减（乘以动量因子），再加上当前梯度乘以学习率的结果。

- 同样地，对于b1、W2和b2也进行类似的动量变量更新。

2. 参数更新：
   - `self.W1 += self.velocity_W1`: 使用动量变量更新参数W1。将当前的参数W1与对应的动量变量相加，得到新的参数值。
   - 同样地，对b1、W2和b2进行相应的参数更新。

自适应学习率算法（Adaptive Learning Rate）

自适应学习率算法根据参数的历史梯度信息自适应地调整学习率。常见的算法包括Adagrad、RMSprop和Adam等。这些算法能够根据参数更新的频率和梯度的大小自动调整学习率，从而更好地适应不同的参数和问题。

Adagrad

Adagrad是一种自适应学习率优化算法，用于梯度下降的改进。它根据参数的历史梯度信息来调整学习率，使得在参数更新中对于频繁出现的稀疏特征有较大的学习率，对于不频繁出现的特征有较小的学习率。下面是Adagrad算法的主要特点和步骤：

1. 特点：
   - 自适应学习率：Adagrad使用自适应的学习率，根据参数的历史梯度信息来动态调整学习率的大小。
   - 稀疏特征关注：Adagrad能够对频繁出现的稀疏特征提供较大的学习率，使得模型能够更快地学习到这些特征的重要性。
   - 参数特定的学习率：Adagrad为每个参数维护一个独立的学习率，使得不同参数可以具有不同的学习速度。
2. 步骤：
   - 初始化参数：随机初始化模型的参数。
   - 初始化历史梯度累积：为每个参数维度初始化一个累积的历史梯度平方和，初始值为0。
   - 迭代更新：对于每个训练样本，进行以下步骤：
     - 前向传播：使用当前参数对该训练样本进行前向传播，计算模型的输出。
     - 计算损失函数：根据模型的输出和训练样本的真实标签，计算损失函数的值。
     - 反向传播：通过反向传播算法计算损失函数对参数的梯度。
     - 更新历史梯度累积：对于每个参数维度，将当前梯度的平方累加到历史梯度累积中。
     - 计算学习率：根据历史梯度累积，计算学习率。通常使用一个小的常数（如1e-8）加在分母上，以避免除以0的情况。
     - 参数更新：使用学习率按照梯度下降的规则更新模型的参数。除以学习率可以使得较大的历史梯度累积对应较小的学习率，较小的历史梯度累积对应较大的学习率。
   - 重复迭代：重复执行前面的步骤，直到达到停止条件（如达到最大迭代次数或损失函数收敛）。

Adagrad算法根据参数的历史梯度累积为每个参数维度提供不同的学习率，使得在参数更新时能够更好地适应不同特征的重要性。相对于固定学习率的梯度下降算法，Adagrad能够更快地收敛并提高模型的性能。然而，Adagrad的一个缺点是历史梯度累积的平方和可能会不断增加，导致后期学习率过小，参数更新变得缓慢。为了解决这个问题，后续的优化算法如Adadelta和Adam进行了改进。

$$\theta_{t+1} \leftarrow \theta_t - \frac{\eta}{\sqrt{G_t + \varepsilon}} \odot g_t$$

式中ε是小正实数，η常取0.01，$\odot$表示向量的元素级乘法

```
self.G_W1 = np.zeros_like(self.W1)
self.G_b1 = np.zeros_like(self.b1)
```

```
self.G_W2 = np.zeros_like(self.W2)
self.G_b2 = np.zeros_like(self.b2)
```

```
elif self.Adagrad == 1:
    epsilon = 1e-8  # 为了数值稳定性而添加的小常数
    # 更新累积梯度
    self.G_W1 += dW1 ** 2
    self.G_b1 += db1 ** 2
    self.G_W2 += dW2 ** 2
    self.G_b2 += db2 ** 2

    # 更新参数
    self.W1 -= learning_rate * dW1 / (np.sqrt(self.G_W1) + epsilon)
    self.b1 -= learning_rate * db1 / (np.sqrt(self.G_b1) + epsilon)
    self.W2 -= learning_rate * dW2 / (np.sqrt(self.G_W2) + epsilon)
    self.b2 -= learning_rate * db2 / (np.sqrt(self.G_b2) + epsilon)

            '''步长会慢慢减小失去调节作用'''
```

1. 初始化常数：
   - `epsilon = 1e-8`: 为了数值稳定性而添加的小常数。它被加在分母上，以避免除以0的情况。
2. 更新累积梯度：
   - `self.G_W1 += dW1 ** 2`: 将当前梯度的平方累加到参数W1的累积梯度`self.G_W1`中。
   - 同样地，对b1、W2和b2也进行类似的累积梯度更新。
3. 参数更新：
   - `self.W1 -= learning_rate * dW1 / (np.sqrt(self.G_W1) + epsilon)`: 使用Adagrad算法更新参数W1。计算学习率时，将当前梯度除以参数W1的累积梯度的平方根加上一个小常数epsilon，再乘以学习率，得到参数W1的更新量。然后将参数W1减去该更新量，完成参数更新。
   - 同样地，对b1、W2和b2进行相应的参数更新。

RMSprop

RMSprop（Root Mean Square Propagation）是一种自适应学习率优化算法，用于梯度下降的改进。它是对Adagrad算法的改进，针对Adagrad中学习率过早降低的问题进行了调整。RMSprop通过引入指数加权移动平均来平滑历史梯度累积的影响，以便更好地适应不同参数的学习率。以下是RMSprop算法的主要特点和步骤：

1. 特点：
   - 自适应学习率：RMSprop使用自适应的学习率，根据参数的历史梯度信息来动态调整学习率的大小。
   - 平滑历史梯度累积：RMSprop通过指数加权移动平均对历史梯度累积进行平滑处理，使得旧的梯度信息对学习率的影响逐渐减小。
   - 参数特定的学习率：RMSprop为每个参数维度维护一个独立的学习率，使得不同参数可以具有不同的学习速度。
2. 步骤：

   与Adagrad相似

RMSprop算法通过平滑历史梯度累积的影响，使得学习率能够更好地适应参数的更新需求。相比于Adagrad算法，RMSprop能够在训练过程中更好地平衡历史梯度的影响和学习率的大小，从而提高模型的收敛速度和性能。然而，RMSprop仍然具有一些限制，如对学习率的选择敏感，可能需要手动调节参数。为了进一步改进，后续的优化算法如Adam提供了更全面的自适应学习率调整和动量更新。

$$
\begin{aligned}
G_t &= \gamma\, G_{t-1} + (1-\gamma)\, g_t^2 \\
\theta_{t+1} &\leftarrow \theta_t - \frac{\eta}{\sqrt{G_t + \varepsilon}} \odot g_t
\end{aligned}
$$

RMSprop的提出者Hinton建议设定平衡因子γ为0.9，学习率η为0.001

```
elif self.RMSProp == 1:
    p = 0.999
    epsilon = 1e-8
    self.G_W1 = p * self.G_W1 + (1-p) * (dW1 ** 2)
    self.G_b1 = p * self.G_b1 + (1-p) * (db1 ** 2)
    self.G_W2 = p * self.G_W2 + (1-p) * (dW2 ** 2)
    self.G_b2 = p * self.G_b2 + (1-p) * (db2 ** 2)

    # 更新参数
    self.W1 -= learning_rate * dW1 / (np.sqrt(self.G_W1) + epsilon)
    self.b1 -= learning_rate * db1 / (np.sqrt(self.G_b1) + epsilon)
    self.W2 -= learning_rate * dW2 / (np.sqrt(self.G_W2) + epsilon)
    self.b2 -= learning_rate * db2 / (np.sqrt(self.G_b2) + epsilon)
```

1. 初始化常数：
   - `p = 0.999`：指数加权移动平均系数，用于平滑历史梯度累积。它控制了新梯度和旧梯度对平均值的相对贡献。
   - `epsilon = 1e-8`：为了数值稳定性而添加的小常数，与RMSprop的分母相关。避免除以0的情况。
2. 更新累积梯度：
   - `self.G_W1 = p * self.G_W1 + (1-p) * (dW1 ** 2)`：使用指数加权移动平均更新参数W1的平方梯度累积。它将当前梯度的平方乘以(1-p)作为新的累积梯度，再将历史累积梯度乘以p与新的累积梯度相加，得到参数W1的新累积梯度。这样做可以平滑历史梯度累积的影响，使得旧的梯度信息逐渐减小。
   - 对b1、W2和b2也进行类似的累积梯度更新。
3. 参数更新：
   - `self.W1 -= learning_rate * dW1 / (np.sqrt(self.G_W1) + epsilon)`：使用RMSprop算法更新参数W1。计算学习率时，将当前梯度除以参数W1的平方梯度累积的平方根加上一个小常数epsilon，再乘以学习率，得到参数W1的更新量。然后将参数W1减去该更新量，完成参数更新。
   - 对b1、W2和b2进行相应的参数更新。

Adam

Adam（Adaptive Moment Estimation）是一种自适应学习率优化算法，结合了动量（momentum）和自适应学习率的特性。它是一种常用的优化算法，被广泛应用于深度学习中。

Adam算法的主要思想是在梯度下降的基础上引入动量和自适应学习率。

以下是Adam算法的主要特点和步骤：

1. 特点：
   - 自适应学习率：Adam算法使用自适应的学习率，通过根据每个参数的历史梯度信息来动态调整学习率的大小。
   - 动量更新：Adam算法引入了动量的概念，以平滑参数更新的方向，加速收敛过程。
   - 参数特定的学习率：Adam为每个参数维度维护一个独立的学习率，使得不同参数可以具有不同的学习速度。
   - 自适应调整学习率：Adam算法通过估计梯度的一阶矩估计（均值）和二阶矩估计（方差）来自适应地调整学习率。
2. 步骤：

   与Adagrad相似

Adam算法通过自适应调整学习率和动量更新的方式，能够更好地适应不同参数的更新需求，同时加速收敛过程。相比于传统的梯度下降和基于动量的优化算法，Adam算法通常能够更快地达到较低的损失值，提高模型的性能。

$$
\begin{aligned}
m_t &= \gamma_1 \ m_{t-1} + (1-\gamma_1)\, g_t \\
v_t &= \gamma_2 \ v_{t-1} + (1-\gamma_2)\, g_t^2
\end{aligned}
$$

偏差校正：

$$\hat{m}_t = \frac{m_t}{1-\gamma_1^t}$$
$$\hat{v}_t = \frac{v_t}{1-\gamma_2^t}$$
$$\theta_{t+1} = \theta_t - \frac{\eta}{\sqrt{\hat{v}_t}+\varepsilon} \odot \hat{m}$$

超参数的建议值为γ_1=0.9，γ_2=0.999，$\varepsilon = 1 \times 10^{-8}$

```
elif self.Adam == 1:
    '''
    取消了adam的偏差校正环节 loss收敛的更快更好

    取消偏差校正可能使得算法在训练初期对学习率的调整更加敏感
    从而有助于更快地找到合适的学习率
    '''
    p = 0.9999
    q = 0.9
    epsilon = 1e-8
    self.G_W1 = (p * self.G_W1 + (1-p) * (dW1 ** 2))
    self.G_b1 = (p * self.G_b1 + (1-p) * (db1 ** 2))
    self.G_W2 = (p * self.G_W2 + (1-p) * (dW2 ** 2))
    self.G_b2 = (p * self.G_b2 + (1-p) * (db2 ** 2))

    self.velocity_W1 = (q * self.velocity_W1 + (1-q) * dW1)
    self.velocity_b1 = (q * self.velocity_b1 + (1-q) * db1)
    self.velocity_W2 = (q * self.velocity_W2 + (1-q) * dW2)
    self.velocity_b2 = (q * self.velocity_b2 + (1-q) * db2)
    # 更新参数
    self.W1 -= learning_rate * self.velocity_W1 / (np.sqrt(self.G_W1) + epsilon)
    self.b1 -= learning_rate * self.velocity_b1 / (np.sqrt(self.G_b1) + epsilon)
    self.W2 -= learning_rate * self.velocity_W2 / (np.sqrt(self.G_W2) + epsilon)
    self.b2 -= learning_rate * self.velocity_b2 / (np.sqrt(self.G_b2) + epsilon)
```

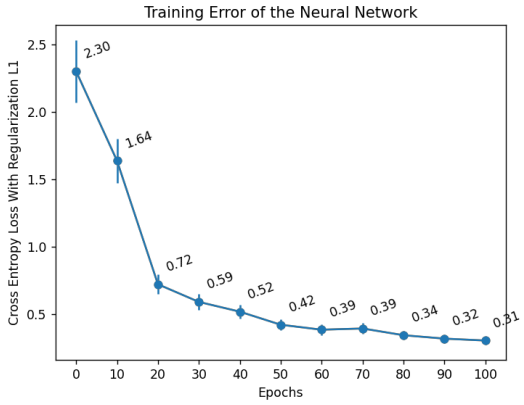
实验数据

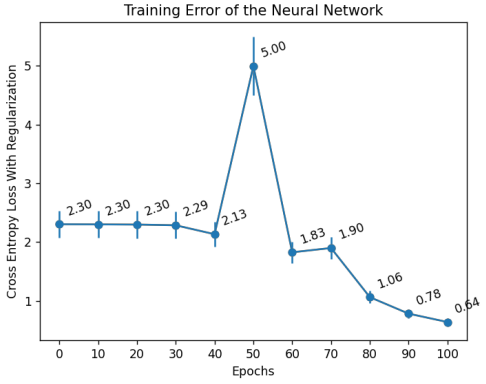
一层隐藏层与两层隐藏层

对比一层隐藏层的神经网络与两层隐藏层的神经网络在MNIST数据集上的表现，两个网络参数如下：

	第一层神经元数量	第二层神经元数量	步长	网络训练次数	耗时	正确率
一层隐藏层	256		0.00001	100	1min 19.15msc	91.76%
两层隐藏层	256	128	0.00001	100	1min 52.29msc	79.24%

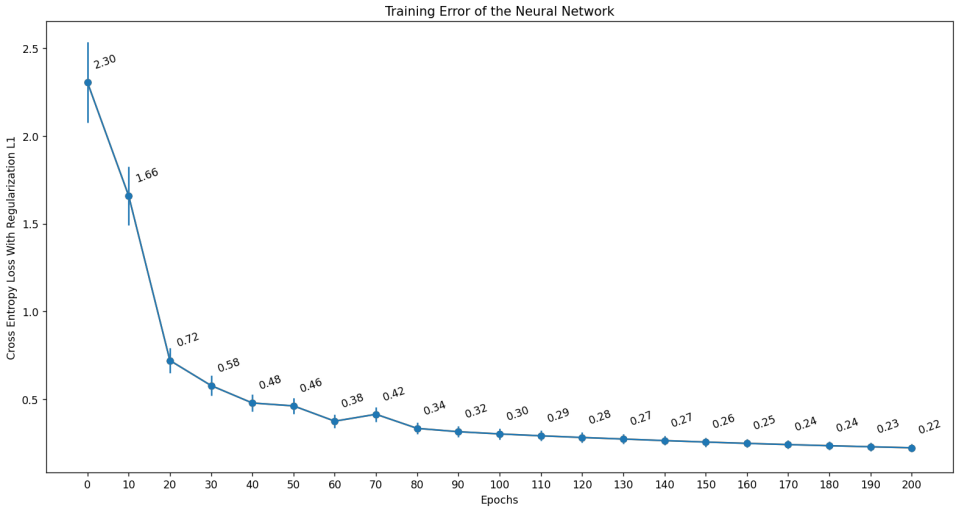
一层隐藏层Loss随训练次数的变化　　*两层隐藏层Loss随训练次数的变化*



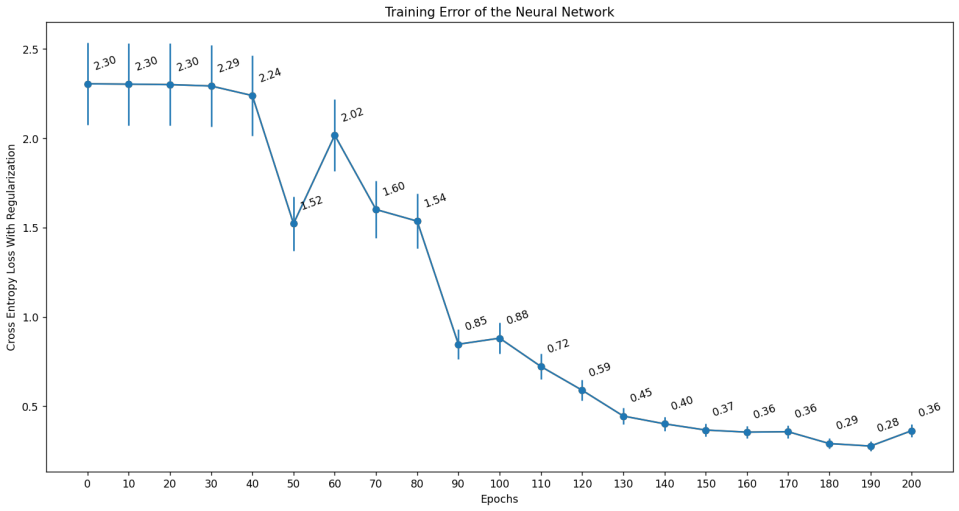


	第一层神经元数量	第二层神经元数量	步长	网络训练次数	耗时	正确率
一层隐藏层	256		0.00001	200	2min 42.64msc	93.74%
两层隐藏层	256	128	0.00001	200	4min 4.77msc	89.01%

一层隐藏层Loss随训练次数的变化



两层隐藏层Loss随训练次数的变化



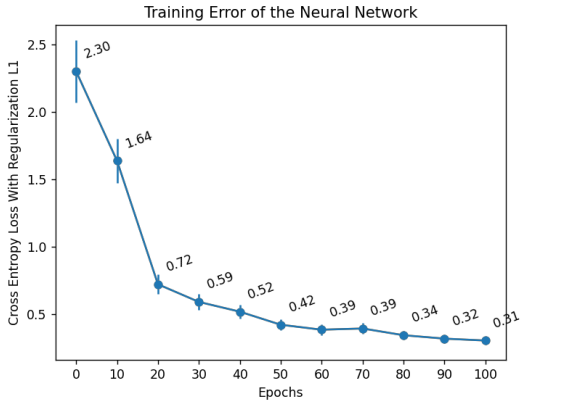
由这4组数据可以看出，用MNIST数据集测试神经网络时，**含两层隐藏层的网络速度更慢，准确率更低，loss波动大。这可能是因为，隐藏层会增加网络的复杂度和参数量，这可能导致网络更难以训练和优化。更复杂的网络需要更多的计算资源和时间来进行训练。**

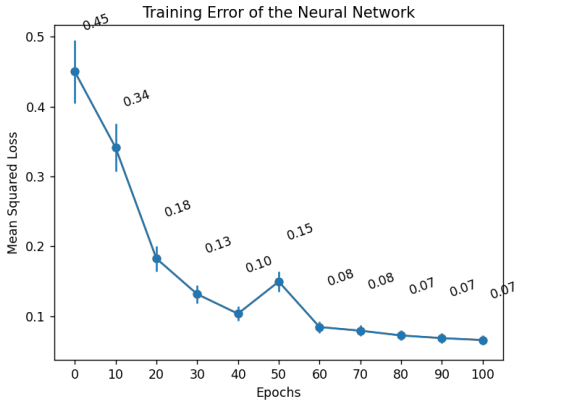
所以，对于MNIST这种简单的数据集，一层隐藏层的简单神经网络就已经足够满足需求，两层神经网络反而会因为复杂的拓扑结构引起波动，并耗费更多的时间。

后面的实验统一使用单层神经网络

交叉熵与均方误差

左边是一层隐藏层的神经网络的交叉熵损失变化图，右边是均方误差损失变化图





由此可见，相对于均方误差，交叉熵对离散概率分布的变化更加敏感。在分类任务中，交叉熵对于错误分类的惩罚更加严厉，因此可以促使模型更快地收敛。

激活函数对比

激活函数	步长	神经元数量	训练次数	耗时	测试集正确率	训练集正确率
relu	0.00001	256	100	1min 26.94msc	91.83%	91.61%

激活函数	步长	神经元数量	训练次数	耗时	测试集正确率	训练集正确率
leaky_relu	0.00001	256	100	1min 35.32msc	91.85%	92.37%
tanh	0.00001	256	100	2min 46.32msc	91.27%	91.05%
sigmoid	0.00001	256	100	2min 37.22msc	78.12%	77.61%
relu	0.00001	256	200	2min 58.42msc	93.89%	93.65%
leaky_relu	0.00001	256	200	3min 28.27msc	93.78%	93.67%
tanh	0.00001	256	200	5min 41.07msc	92.38%	92.41%
sigmoid	0.00001	256	200	5min 11.51msc	89.23%	87.96%

由此8组数据可知，**relu函数计算最快，tanh函数计算最复杂；relu以及leaky relu函数正确率最高，sigmoid函数正确率远低于其他激活函数，可见loss收敛较慢，需要更多次的训练**。sigmoid正确率低的可能原因是：Sigmoid函数的导数在接近饱和区域（0或1）时非常接近于零，导致反向传播时梯度逐渐减小，可能导致梯度消失问题，使得深层网络难以训练。

综合所有实验数据来看，**relu最适合MNIST数据集，接下来的实验都选择relu作为激活函数。**

各个优化算法对比

批量梯度下降 小批量梯度下降 随机梯度下降

注：小批量为每次**6000**个样本 所有样本训练一次才算一个训练次数

优化算法	步长	训练次数	耗时	测试集正确率	训练集正确率
BGD	0.00001	100	1min 26.94msc	91.83%	91.61%
MBGD	0.0001	100	1min 18.64msc	97.33%	98.09%
SGD	0.01	5	7min 34.93msc	97.78%	99.32%

由此可见三个算法中，**小批量梯度下降可以在较短的时间达到较大的正确率，也不存在严重的过拟合现象**。BGD算法在没有使用其他优化算法的情况下，单纯通过增加训练次数很难达到97的正确率，这说明**MBGD计算代价相对较低，参数更新速度较快**。

SGD算法的正确率仅在5次训练后就能达到97，参数更新速度最快，但是因为每个样本都单独训练所以效率很低，耗时太长

下面的实验默认采用MBGD算法

动量法 自适应学习率算法

优化算法	步长	训练次数	耗时	测试集正确率	训练集正确率
Momentum	0.00001	100	1min 11.06msc	95.67%	95.72%
Adagrad	0.1	100	1min 13.79msc	96.89%	99.08%
RMSprop	0.001	100	1min 16.46msc	97.63%	99.58%
Adam	0.00001	100	1min 15.90msc	97.33%	98.57%
无	0.0001	100	1min 11.03msc	97.21%	98.06%

从实验数据可以看出优化算法的计算效率都不错，正确率也都很高，但是**对于没有使用优化算法的MBGD算法，Momentum，Adagrad的正确率反而变低了**。

应用动量法后，在MNIST数据集上的正确率下降，可能有以下几个原因：

1. 不合适的动量系数：动量系数的选择对于算法的性能至关重要。如果选择了过大或过小的动量系数，都可能导致算法的性能下降。过大的动量系数可能会导致更新步骤在错误的方向上增加，而过小的动量系数可能会导致更新步骤受到过多的历史梯度影响，导致收敛速度变慢。因此，需要仔细选择和调整动量系数，以找到最佳的性能。

2. 数据集特点：MNIST数据集是一个相对简单的手写数字分类任务。由于数据集的相对简单性，应用动量法可能会导致算法在局部最小值附近震荡，而不是稳定地收敛。对于简单的数据集，简单的梯度下降算法可能已经足够好，而引入动量法可能会导致更多的噪音和不稳定性。

应用Adagrad后，在MNIST数据集上的正确率下降，可能有以下几个原因：

1. 学习率衰减过快：Adagrad算法会自适应地调整每个参数的学习率，根据参数的历史梯度进行缩放。然而，如果学习率衰减过快，可能导致在训练过程中学习率过小，使得模型无法充分学习和更新参数。这可能导致模型收敛速度变慢，正确率下降。
2. 稀疏梯度问题：Adagrad在处理稀疏梯度问题时可能表现不佳。对于MNIST数据集这样的图像分类任务，输入图像通常是稠密的，但是在某些情况下，可能会出现稀疏的梯度。Adagrad的学习率调整机制会导致对于稀疏梯度较大的参数进行较小的更新，这可能使得模型无法充分利用关键特征的梯度信息，从而影响正确率。
3. 参数更新过度：Adagrad根据参数的历史梯度平方和进行学习率调整。这意味着在训练过程中，参数的学习率会不断减小。在某些情况下，学习率的减小可能过于激进，导致参数更新过度。过度更新可能会使模型过于敏感，导致正确率下降。
4. 超参数调整：Adagrad还有其他需要调整的超参数，例如学习率和正则化项。不正确的超参数设置可能导致性能下降。因此，需要进行仔细的超参数调整，以找到最佳的组合。

正则化前后对比

正则化方式	步长	优化算法	训练次数	耗时	测试集正确率	训练集正确率
无正则化	0.1	Adagrad	100	1min 13.79msc	96.89%	99.08%
L^{1}正则化	0.1	Adagrad	100	1min 14.61msc	97.16%	98.04%
L^{2}正则化	0.1	Adagrad	100	1min 18.35msc	96.95%	98.14%
无正则化	0.001	RMSprop	100	1min 16.46msc	97.63%	99.58%
L^{1}正则化	0.001	RMSprop	100	1min 20.58msc	97.75%	99.14%
L^{2}正则化	0.001	RMSprop	100	1min 20.10msc	97.92%	99.67%

通过在损失函数中引入惩罚项，正则化能够使梯度更加平缓，避免参数更新过大，从而提高优化算法的稳定性。

1. 正则化的作用：正则化是一种用于缓解过拟合问题的技术。通过在损失函数中引入正则化项，可以对模型的复杂度进行约束，从而提高其泛化能力。
2. L1正则化与L2正则化：实验数据中使用了L1正则化和L2正则化两种常见的正则化方法。L1正则化通过加入权重的绝对值和一个正则化参数，促使一部分权重变为零，从而实现特征选择的效果。L2正则化通过加入权重的平方和一个正则化参数，使得权重趋向于较小的值，从而减小了权重之间的差异。
3. 效果对比：与无正则化相比，实验结果显示正则化方法能够在一定程度上提高测试准确率。具体而言，L1正则化在大部分设置下表现略低于无正则化，而L2正则化在某些设置下稍微超过无正则化。这可能是因为L1正则化在特征选择方面更为严格，可能剔除了一些对模型有贡献的特征，而L2正则化则对所有的特征进行了平衡。
4. 超参数选择：实验中还考察了不同的学习率和优化算法。结果显示，较小的学习率（0.001）和RMSprop优化算法在大多数情况下表现较好。这表明在正则化的情况下，使用较小的学习率和一种适应性优化算法可以更好地控制权重的更新，进而提高模型的性能。

所以在**使用正则化后，对于训练集的正确率大都略微下降，这是因为参数更新更加平缓；而对于测试集的正确率略微上升，这是因为正则化提高了模型的泛化能力。**