

## 1 编程任务

在不使用现成框架的条件下，实现书写体数字识别。满足以下要求：

1. 使用两层或者以上的全连接层组成的 MLP
2. 激活函数使用 ReLU
3. MLP 模型参数更新使用随机梯度更新 (SGD)

## 2 编程思路及代码实现

在本次编程作业中，使用了两层隐藏层，每一层隐藏层含有 256 个神经元。隐藏层的激活函数为 ReLU 函数，输出层使用 softmax 函数，损失函数使用交叉熵损失函数。

### 2.1 数据的读取

本次作业使用的数据集为 MNIST 手写体数字数据集，下载自数据集官网：<https://yann.lecun.com/exdb/mnist/>  
MNIST 数据集包含四个压缩包文件，分别为

- train-images-idx3-ubyte.gz：训练集图片，共 60,000 张 0~9 手写体数字图片。
- train-labels-idx1-ubyte.gz：训练集图片对应的标签，即实际数字
- t10k-images-idx3-ubyte .gz：测试集图片，10,000 张手写体数字图片
- t10k-labels-idx1-ubyte.gz：测试集图片对应的标签

在完成解压之后，对于图像数据集文件，文件的开始使用 16 个字节存放 4 个 32bit 的 int 类型的数，分别为：magic number、图片数量、图片的行数和图片的列数，在文件中以大端的格式存放，然后存放的才是图片。对于标签数据集文件，文件的开始使用 8 个字节存放 2 个 32bit 的 int 类型的数，分别为：magic number、标签的数量，在这之后存放标签数据。

我们根据文件的结构对数据集中的图像和标签进行了读取，对于图像数据，我们进行了归一化操作，即将图像中 0~255 的数据映射到 0~1，然后将图像数据转换成长度为 784 的向量，并将标签进行 one-hot 编码，得到长度为 10 的向量。

数据读取部分的代码如下所示

```
1 # coding:utf-8
2 import numpy as np
3 import gzip
4 from struct import unpack
5
6 #读取图像
7 def read_image(path):
8     with gzip.open(path, 'rb') as f:
9         magic, num, rows, cols = unpack('>4I', f.read(16))
```

```

10     img=np.frombuffer(f.read(), dtype=np.uint8).reshape(num, 28*28)
11     return img
12
13 #读取标签
14 def read_label(path):
15     with gzip.open(path, 'rb') as f:
16         magic, num = unpack('>2I', f.read(8))
17         lab = np.frombuffer(f.read(), dtype=np.uint8)
18         # print(lab[1])
19     return lab
20
21 #将图像信息正则化, 即0-255 --> 0-1
22 def normalize_image(image):
23     img = image.astype(np.float32) / 255.0
24     return img
25
26 #将标签进行one-hot编码, 如数字标签5转换为[0,0,0,0,0,1,0,0,0,0,0]
27 def one_hot_label(label):
28     lab = np.zeros((label.size, 10))
29     for i, row in enumerate(lab):
30         row[label[i]] = 1
31     return lab
32
33 def loadMnist(train_data_path, train_labels_path, test_data_path, test_labels_path, normalize
    = 1, one_hot = 1):
34     image = {
35         'train' : read_image(train_data_path),
36         'test' : read_image(test_data_path)
37     }
38     label = {
39         'train' : read_label(train_labels_path),
40         'test' : read_label(test_labels_path)
41     }
42     if normalize:
43         for type in ('train', 'test'):
44             image[type] = normalize_image(image[type])
45     if one_hot:
46         for type in ('train', 'test'):
47             label[type] = one_hot_label(label[type])
48     return (image['train'], label['train']), (image['test'], label['test'])
49
50 #导入数据
51 train_images_path = './dataset/train-images-idx3-ubyte.gz'
52 train_labels_path = './dataset/train-labels-idx1-ubyte.gz'
53 test_images_path = './dataset/t10k-images-idx3-ubyte.gz'

```

```

54 test_labels_path = './dataset/t10k-labels-idx1-ubyte.gz'
55 (train_image, train_label), (test_image, test_label) = loadMnist(train_images_path,
    train_labels_path, test_images_path, test_labels_path)

```

## 2.2 参数初始化

如下，是参数初始化的代码：

```

1 def __init__(self, layer_size, learning_rate=0.001, batch_size=64, max_epoch=40):
2     self.layer_size = layer_size #各层神经元个数，如：[784, 256, 256, 10]
3     self.learning_rate = learning_rate
4     self.batch_size = batch_size
5     self.max_epoch = max_epoch #训练的轮数
6
7     self.layer_num = len(layer_size) #隐藏层数量+输入层+输出层
8     self.weights = [] #各层（输入层、隐层）输出的权重
9     self.bias = [] #各层（输入层、隐层）输出的偏
10    for i in range(1, self.layer_num):
11        #由于使用relu激活函数，因此使用He初始化
12        self.weights.append(
13            np.random.randn(self.layer_size[i-1], self.layer_size[i]) * np.sqrt(2 / self.
14                layer_size[i-1])
15        )
16        self.bias.append(np.random.randn(self.layer_size[i]))
17
18    self.delta_w = [] #损失函数关于各层（输入层、隐层）输出的权重weight的偏导
19    self.delta_b = [] #损失函数关于各层（输入层、隐层）输出的偏移bias的偏导
20    self.input_net = [] #各层的输入
21    self.output_net = [] #各层的输出
22    for i in range(self.layer_num):
23        self.input_net.append(np.zeros(self.layer_size[i]))
24        self.output_net.append(np.zeros(self.layer_size[i]))

```

为了代码编写的方便，在本次作业中，将多层感知机模型作为一个类进行编写，上面这部分内容是对模型参数进行初始化，各参数对作用见代码注释。主要说明对是，为了搭配 ReLU 函数，网络权重的初始化采用了何恺明提出的 He 初始化。

## 2.3 前向传输

```

1 def forward(self, data):
2     #前向传输，data为输入层的输入数据
3     self.output_net[0] = data
4     for j in range(1, self.layer_num-1):
5         #求隐藏层的输入与输出
6         self.input_net[j] = np.dot(self.output_net[j-1], self.weights[j-1]) + self.bias[j-1]
7         self.output_net[j] = ReLU(self.input_net[j])

```

```

8      #求输出层的输入与输出
9      self.input_net[-1] = np.dot(self.output_net[-2], self.weights[-1]) + self.bias[-1]
10     self.output_net[-1] = softmax(self.input_net[-1])
11     y_hat = self.output_net[-1]
12     return y_hat

```

以上是前向传播部分的代码，将图像数据作为输入和输入层的输出，前向传播过程中，更新隐藏层和输出层中各层神经元的输入矩阵和输出矩阵，更新公式如下所示：

$$input_i = output_{i-1} * W_i + B_i \quad (1)$$

对于隐藏层，输出为：

$$output_i = ReLU(input_i) \quad (2)$$

对于输出层，输出为：

$$output_i = softmax(input_i) \quad (3)$$

最后返回输出层的输出。

## 2.4 反向传输

```

1  def backward(self, y_hat, y):
2      self.delta_b = []
3      self.delta_w = []
4      delta_output = y_hat - y #输出层的梯度，即损失函数关于输出层输入的导数
5      layer_index = self.layer_num - 1
6      while layer_index > 0:
7          self.delta_b.append(np.sum(delta_output, axis=0))
8          self.delta_w.append(np.dot(self.output_net[layer_index-1].T, delta_output))
9          if layer_index > 1:
10             #计算前一层的梯度
11             delta_output = np.dot(delta_output, self.weights[layer_index-1].T) * deReLU(self.
                input_net[layer_index-1])
12         layer_index -= 1

```

以上为反向传播部分的代码，反向传播的过程中，主要在求损失函数关于各层输入的梯度。将损失记为  $loss$ ，输出层输出记为  $\hat{y}$ ，正确的输出记为  $y$ ，有：

$$loss = - \sum_k y_k \ln \hat{y}_k \quad (4)$$

设输出层的输入为  $z$ ，损失函数关于输出层输出的偏导数有：

$$\frac{\partial loss}{\partial \hat{y}_k} = \begin{cases} -\frac{y_j}{\hat{y}_j}, & k = j \\ 0, & \text{else} \end{cases} \quad (5)$$

$j$  为输入图像所对应的标签（数字）。当  $i = j$  时，

$$\frac{\partial \hat{y}_j}{\partial z_i} = \frac{\partial (\frac{e^{z_i}}{\sum_k e^{z_k}})}{\partial z_i} = \hat{y}_j (1 - \hat{y}_j) \quad (6)$$

当  $i \neq j$  时，

$$\frac{\partial \hat{y}_j}{\partial z_i} = \frac{\partial (\frac{e^{z_j}}{\sum_k e^{z_k}})}{\partial z_i} = -\hat{y}_j \hat{y}_i \quad (7)$$

综上，损失函数对于输出层输入的偏导数为

$$\frac{\partial loss}{\partial z_i} = \sum_k \left( \frac{\partial loss}{\partial \hat{y}_k} \frac{\partial \hat{y}_k}{\partial z_i} \right) = \hat{y}_i - y_i \quad (8)$$

将隐藏层的输入和输出分别记为  $input_i, output_i, i = 1, 2$ ，输入层的输入和输出分别记为  $input_0, output_0$ ，有

$$z = W_2 * output_2 + B_2 \quad (9)$$

$$input_2 = W_1 * output_1 + B_1 \quad (10)$$

$$input_1 = W_0 * output_0 + B_0 \quad (11)$$

根据式子 (2) (9-11) 损失函数关于隐藏层输入的梯度为

$$\begin{aligned} \frac{d loss}{d input_2} &= \frac{d loss}{dz} \frac{dz}{d output_2} \frac{d output_2}{d input_2} \\ &= \frac{d loss}{dz} * W_2^T \odot ReLU'(input_2) \end{aligned} \quad (12)$$

$$\begin{aligned} \frac{d loss}{d input_1} &= \frac{d loss}{d input_2} \frac{d input_2}{d output_1} \frac{d output_1}{d input_1} \\ &= \frac{d loss}{d input_2} * W_1^T \odot ReLU'(input_1) \end{aligned} \quad (13)$$

借此，我们可以求得损失函数关于各权重和偏移的偏导数，对输出层，有

$$\frac{\partial loss}{\partial W_2} = output_2^T * \frac{d loss}{dz} \quad (14)$$

$$\frac{\partial loss}{\partial B_2} = \frac{d loss}{dz} \quad (15)$$

对隐藏层，有

$$\frac{\partial loss}{\partial W_i} = output_i^T * \frac{d loss}{d input_{i+1}} \quad (16)$$

$$\frac{\partial loss}{\partial B_i} = \frac{d loss}{d input_{i+1}} \quad (17)$$

其中， $i = 0, 1$ 。将损失函数关于各权重和偏移的偏导数存放在列表  $\delta_{w_i}$  和  $\delta_{b_i}$  中。

## 2.5 训练过程

为了使用随机梯度下降法对模型参数进行更新，每次训练前，先从训练集中随机抽取 64 个样本组成 batch 进行训练，每个样本数据为一张图像和一个标签，借助它们的梯度均值对模型参数进行更新。在训练过程中，我们一共训练 40 轮，每一轮训练次数训练集样本数除以 batch 大小。每次训练主要分为四个步骤：① 随机选取样本；② 前向传播；③ 反向传播；④ 更新模型参数。训练过程的代码如下所示：

```
1 def train(self, inputs, y):
2     if self.weights == []:
3         for i in range(1, self.layer_num):
4             # 由于使用relu激活函数，因此使用He初始化
5             self.weights.append(
6                 np.random.randn(self.layer_size[i-1], self.layer_size[i]) * np.sqrt(2 / self.
7                     layer_size[i-1])
8             )
9         if self.bias == []:
10             for i in range(1, self.layer_num):
```

```

10         self.bias.append(np.random.randn(self.layer_size[i]))
11     n_sample = inputs.shape[0] #输入样本数
12     iter_num = n_sample // self.batch_size #训练一轮所需迭代次数
13     if n_sample % self.batch_size: iter_num += 1
14     for epoch_index in range(self.max_epoch):
15         for iter_index in range(iter_num):
16             sample_index_array = []
17             for i in range(self.batch_size):
18                 sample_index = np.random.randint((n_sample))
19                 sample_index_array.append(sample_index)
20             input_batch = inputs[sample_index_array]
21             y_batch = y[sample_index_array]
22             # input_batch = inputs[sample_index * self.batch_size: min((sample_index + 1)
23             # * self.batch_size, n_sample)] #输入样本batch
24             # y_batch = y[sample_index * self.batch_size: min((sample_index + 1) * self.
25             # batch_size, n_sample)]#label batch
26             yhat_batch = self.forward(input_batch)#前向传播
27             self.backward(yhat_batch, y_batch)#反向传播
28
29             for i in range(len(self.weights)):
30                 #调整参数
31                 self.weights[i] -= self.learning_rate * self.delta_w[self.layer_num-2-i]
32                 self.bias[i] -= self.learning_rate * self.delta_b[self.layer_num-2-i]
33             if epoch_index < 10:
34                 self.learning_rate -= 0.0001 * self.learning_rate
35             #计算损失与准确度
36             y_hat = self.forward(inputs)
37             loss = cross_entropy_loss(y_hat, y)
38             accur = accuracy(y, y_hat)
39             print('Epoch: {} Loss: {} accuracy: {}'.format(epoch_index, loss, accur))

```

为了实现更好的训练效果, 随着时间的推移, 我们将学习率逐渐调小, 第一轮训练时学习率为 0.003, 经过十轮训练后, 学习率下降为 0.002, 而后保持不变。

### 3 实验结果

在经过训练以后, 本次作业所搭建的 MLP 在 MNIST 数据集的表现令人满意, 对于训练集的预测准确率达到了 99%, 对于测试集的预测准确率也达到了 98%。