机器学习实验七——深度学习实验报告

学号：1813055

姓名：赵书楠

本次实验需要对手写数字数据集mnist进行多分类，我使用numpy实现了卷积神经网络，并分别进行了三分类和十分类的任务，实验的细节和结果将在报告中给出。

基础要求

一、反向传播公式推导

神经网络的精髓就是利用每次前向传播得到的预测值和真实值之间的误差loss对于模型反过来不断进行调整，以得到更好的模型，这个过程就是反向传播，具体来说就是利用整个网络的loss对每层的输出Out的偏导（delta），通过链式法则求出loss对每层的输入In的偏导（next_delta），再作为前一层的输出偏导，不断迭代求下去，利用这些偏导求出参数的梯度，进行每层参数的更新。

$$\frac{\partial Loss}{\partial In} = \frac{\partial Loss}{\partial Out}\frac{\partial Out}{\partial In}$$

下面对于卷积神经网络的每层推导反向传播公式：

1.卷积层

设*代表卷积，以3x3的输入为例，卷积核为2x2。因为池化层没有激活函数，所以可得到下面的等式：

$$\begin{pmatrix} {a_{11}}^{l-1} & {a_{12}}^{l-1} & {a_{13}}^{l-1} \\ {a_{21}}^{l-1} & {a_{22}}^{l-1} & {a_{23}}^{l-1} \\ {a_{31}}^{l-1} & {a_{32}}^{l-1} & {a_{33}}^{l-1} \end{pmatrix} * \begin{pmatrix} {w_{11}}^{l} & {w_{12}}^{l} \\ {w_{21}}^{l} & {w_{22}}^{l} \end{pmatrix} = \begin{pmatrix} {z_{11}}^{l} & {z_{12}}^{l} \\ {z_{21}}^{l} & {z_{22}}^{l} \end{pmatrix}$$

$$\begin{aligned} {z_{11}}^{l} &= {a_{11}}^{l-1}{w_{22}}^{l} + {a_{12}}^{l-1}{w_{21}}^{l} + {a_{21}}^{l-1}{w_{12}}^{l} + {a_{22}}^{l-1}{w_{11}}^{l} \\ {z_{12}}^{l} &= {a_{12}}^{l-1}{w_{22}}^{l} + {a_{13}}^{l-1}{w_{21}}^{l} + {a_{22}}^{l-1}{w_{12}}^{l} + {a_{23}}^{l-1}{w_{11}}^{l} \\ {z_{21}}^{l} &= {a_{21}}^{l-1}{w_{22}}^{l} + {a_{22}}^{l-1}{w_{21}}^{l} + {a_{31}}^{l-1}{w_{12}}^{l} + {a_{32}}^{l-1}{w_{11}}^{l} \\ {z_{22}}^{l} &= {a_{22}}^{l-1}{w_{22}}^{l} + {a_{23}}^{l-1}{w_{21}}^{l} + {a_{32}}^{l-1}{w_{12}}^{l} + {a_{33}}^{l-1}{w_{11}}^{l} \end{aligned}$$

$$\delta^{l-1} = \frac{\partial J}{\partial z^{l-1}} = \frac{\partial J}{\partial a^{l-1}} \cdot \frac{\partial a^{l-1}}{\partial z^{l-1}} = \frac{\partial J}{\partial a^{l-1}} \cdot 1 = \frac{\partial J}{\partial z^{l}} \cdot \frac{\partial z^{l}}{\partial a^{l-1}} = \delta^{l} \cdot \frac{\partial z^{l}}{\partial a^{l-1}}$$

池化层有：$a^{l-1} = \sigma\left(z^{l-1}\right) = z^{l-1}$

于是根据next_delta与delta的关系可以推出每个位置上delta与卷积核w存在的数量关系：

$$\delta^{l-1} = \delta^{l} \cdot \frac{\partial z^{l}}{\partial a^{l-1}}$$

$${\delta_{11}}^{l-1} = {\delta_{11}}^{l}{w_{22}}^{l} \qquad {\delta_{12}}^{l-1} = {\delta_{11}}^{l}{w_{21}}^{l} + {\delta_{12}}^{l}{w_{22}}^{l} \qquad {\delta_{13}}^{l-1} = {\delta_{12}}^{l}{w_{21}}^{l}$$

$$\begin{aligned} {\delta_{21}}^{l-1} &= {\delta_{11}}^{l}{w_{12}}^{l} + {\delta_{21}}^{l}{w_{22}}^{l} \\ {\delta_{22}}^{l-1} &= {\delta_{11}}^{l}{w_{11}}^{l} + {\delta_{12}}^{l}{w_{12}}^{l} + {\delta_{22}}^{l}{w_{22}}^{l} \\ {\delta_{23}}^{l-1} &= {\delta_{12}}^{l}{w_{11}}^{l} + {\delta_{22}}^{l}{w_{21}}^{l} \end{aligned}$$

$${\delta_{31}}^{l-1} = {\delta_{21}}^{l}{w_{12}}^{l} \qquad {\delta_{32}}^{l-1} = {\delta_{21}}^{l}{w_{11}}^{l} + {\delta_{22}}^{l}{w_{12}}^{l} \qquad {\delta_{33}}^{l-1} = {\delta_{22}}^{l}{w_{11}}^{l}$$

写成卷积形式则有：

$$\begin{pmatrix} 0 & 0 & 0 & 0 \\ 0 & {\delta_{11}}^{l} & {\delta_{12}}^{l} & 0 \\ 0 & {\delta_{21}}^{l} & {\delta_{22}}^{l} & 0 \\ 0 & 0 & 0 & 0 \end{pmatrix} * rot180 \begin{pmatrix} {w_{11}}^{l} & {w_{12}}^{l} \\ {w_{21}}^{l} & {w_{22}}^{l} \end{pmatrix} = \begin{pmatrix} {\delta_{11}}^{l-1} & {\delta_{12}}^{l-1} & {\delta_{13}}^{l-1} \\ {\delta_{21}}^{l-1} & {\delta_{22}}^{l-1} & {\delta_{23}}^{l-1} \\ {\delta_{31}}^{l-1} & {\delta_{32}}^{l-1} & {\delta_{33}}^{l-1} \end{pmatrix}$$

最后可以得到：

$$\frac{\partial Loss}{\partial In} = conv(\frac{\partial Loss}{\partial Out}, rot180(W))$$

2.池化层

对于最大池化来说，输出就是输入矩阵在某些位置上元素的输出，所以next_delta就是delta扩充成输入形状。

$$\frac{\partial Loss}{\partial In} = upsample(\frac{\partial Loss}{\partial Out})$$

3.全连接层

全连接层中in与w矩阵相乘得到out

$$\begin{aligned}\frac{\partial Loss}{\partial In} &= \frac{\partial Loss}{\partial Out}\frac{\partial Out}{\partial In} \\ &= \frac{\partial Loss}{\partial Out}\frac{\partial In \cdot W}{\partial In} \\ &= \frac{\partial Loss}{\partial Out}W^T\end{aligned}$$

4.激活函数

Sigmoid：

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

$$Sigmoid'(x) = Sigmoid(x)(1 - Sigmoid(x))$$

$$\begin{aligned}\frac{\partial Loss}{\partial In} &= \frac{\partial Loss}{\partial Out}\frac{\partial Out}{\partial In} \\ &= \frac{\partial Loss}{\partial Out}\frac{\partial Sigmoid(In)}{\partial In} \\ &= \frac{\partial Loss}{\partial Out}Sigmoid'(x)\end{aligned}$$

ReLU：

$$ReLU(x) = \begin{cases} x & \text{x} \geq 0 \\ 0 & \text{x}<0 \end{cases}$$

$$\frac{\partial Loss}{\partial In} = \begin{cases} \frac{\partial Loss}{\partial Out} & \text{x} \geq 0 \\ 0 & \text{x}<0 \end{cases}$$

5.交叉熵代价函数softmax

设zi是输出向量中数字i的值，通过softmax得到i输出的概率：

$$Softmax(z_i) = \frac{exp(z_i)}{\sum_{j=1}^{n} exp(z_j)}$$

两边取对数：

$$-Loss = ln(Softmax(z_i)) = z_i - ln(\sum_{j=1}^{n} exp(z_j))$$

$$\frac{\partial Loss}{\partial z_i} = \frac{exp(z_i)}{\sum_{j=1}^{n} exp(z_j)} - \delta_{ij}$$

当i=j，delta_ij=1；当i≠j，delta_ij=0.

二、预处理mnist数据集

> 实验数据集为 MNIST - 0、1、2 三类手写数字识别数据集，从 tensor flow 上获取并对数据做预处理工作

首先通过`from keras.datasets import mnist`下载十分类数据集mnist.npz到本地，然后通过标签筛选出0,1,2的样本，得到了三分类数据集mnist_3_types.npz。

```
    data = np.load("mnist.npz")
    X_train, y_train, X_test, y_test = data['x_train'], data['y_train'], data['x_test'], data['y_test']
    n = len(X_train)
```

```
    x1, y1, x2, y2 = [], [], [], []
    for i in range(n):
        if y_train[i] < 3:
            x1.append(X_train[i])
            y1.append(y_train[i])
    m = len(X_test)
    for i in range(m):
        if y_test[i] < 3:
            x2.append(X_test[i])
            y2.append(y_test[i])
    np.savez("mnist_3_types.npz", x_train=x1, y_train=y1, x_test=x2,
y_test=y2)
```

十分类数据集的训练集大小为60000，测试集为10000。

三分类数据集的训练集大小为18623，测试集为3147。

三、实现卷积神经网络

> 对清洗后的数据集进行预测，划分训练集和测试集，计算并可视化每个 epoch 的训练误差、训练精度、测试误差和测试精度

首先，对于庞大规模的训练集，我们要将其划分为若干个大小为batch_size的批（batch），用每个batch里面的图片一起进行一次训练，这本质上是采用小批量批量梯度下降（mini-batch）的思想，相比每次只对一个样本进行训练，可以使计算出的梯度更好地代表样本整体，另外可以利用矩阵操作进行并行计算，大大减小训练时间。

1.卷积层

卷积层的工作就是让若干个卷积核和输入图像进行卷积运算，多个卷积核映射了多个特征，从而提取图像的邻域信息。

（1）卷积层的成员变量

input_shape：卷积层的输入每次是一个batch，输入形状input_shape=[batch_size, height, weight, input_channels]，分别是batch大小、图片的高、宽和输入图片的通道数。输入通道数为3指的是每张图片的每个像素是由3个数值表示的（比如rgb表示法），而mnist的图片是单通道的，但这并不意味着我们不需要这个维度，当我们的卷积神经网络有多个卷积层时，第二个卷积层及其之后的卷积层的输入是池化层的输出，此时的输入必然是多通道的。

output_channels：卷积核数，也称输出通道数

weights：若干个卷积核组成的数组。该数组的形状是[kernel_size, kernel_size, self.input_channels, self.output_channels]，即卷积核的高、卷积核的宽、输入通道数、卷积核数。注意如果输入是3通道的，那么卷积核也应该是3通道的，也就是说每个卷积核其实是有3个子卷积核。

bias：卷积运算结果的偏置量

delta：整个卷积神经网络的loss对该卷积层输出的偏导

w_grad：weights的梯度

b_grad：bias的梯度

output_shape：卷积层输出的形状

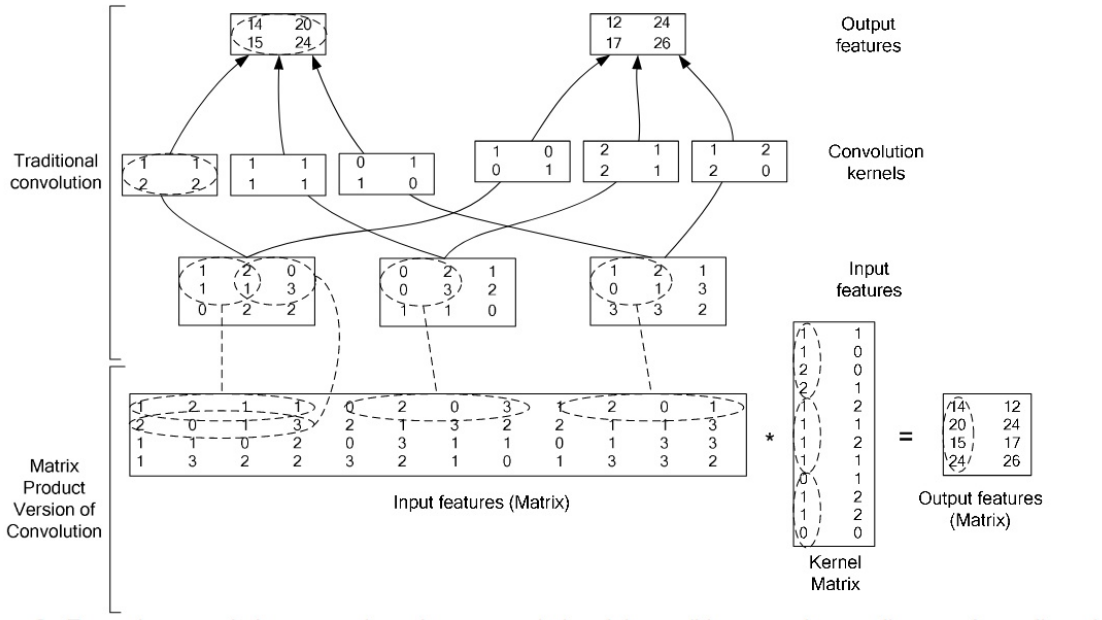
```
# 卷积层
class ConvolutionLayer(object):
```

```
    def __init__(self, input_shape, output_channels, kernel_size=3,
stride=1):  # 3*3的卷积核
        self.input_shape = input_shape  # batch_size*w*h*通道数
        self.output_channels = output_channels  # 卷积核数
        self.input_channels = input_shape[-1]  # 输入图片的通道数，注意如果输入是
3通道的，那么卷积核也应该是三通道的（3倍的参数）
        self.batch_size = input_shape[0]
        self.stride = stride
        self.kernel_size = kernel_size

        weights_scale = np.sqrt(reduce(lambda x, y: x * y, input_shape) /
self.output_channels)
        # 卷积核大小*图片通道数*卷积核数
        self.weights = np.random.standard_normal(
            (kernel_size, kernel_size, self.input_channels,
self.output_channels)) / weights_scale
        # b的数量=卷积核数
        self.bias = np.random.standard_normal(self.output_channels) /
weights_scale
        # 输出形状：batch*h*w*卷积核数
        self.delta = np.zeros((self.batch_size, (input_shape[1] -
kernel_size + 1) // stride,
                               (input_shape[2] - kernel_size + 1) // stride,
self.output_channels))
        self.w_grad = np.zeros(self.weights.shape)
        self.b_grad = np.zeros(self.bias.shape)
        self.output_shape = self.delta.shape
```

（2）卷积层的前向传播

如何让图片和卷积核做卷积呢？假设每张图片是28x28的，卷积核是5x5的，那么首先让图片左上角的5x5子矩阵的元素和卷积核的元素对应相乘，放入输出矩阵的左上角。不断移动卷积核这个“框”，对图片每个5x5的子矩阵都进行卷积，即可得到一个(28-5+1)x(28-5+1)，即24x24的输出矩阵。这个过程也叫做**权值共享**，即用一个卷积核对整张图片不同部分进行卷积。

事实上，卷积运算可以转为矩阵相乘的形式，如图所示。**im2col**函数将输入图片变换为一个矩阵，以下面一张三通道的3x3图片为例，将每个通道上被卷积核框起的2x2部分化为一个行向量，并不断向下拼接形成一个4x4的矩阵，各通道变形后的矩阵横向拼接，得到一个4x12的矩阵。每个卷积核变换为一个列向量，由卷积核每个通道纵向拼接，不同卷积核之间横向拼接，得到一个12x2的矩阵。这两个矩阵相乘后，再reshape成两个2x2的矩阵，就是卷积层需要的输出了。




```
    # 把图片中每个卷积核对应的子矩阵转成行向量，拼成一个新的矩阵
    def im2col(self, img):
        col_img = []
        for i in range(0, img.shape[1] - self.kernel_size + 1, self.stride):
            for j in range(0, img.shape[2] - self.kernel_size + 1,
self.stride):
                col = img[:, i:i + self.kernel_size, j:j + self.kernel_size,
:].reshape(-1)  # 图片是四维的，只截取2,3维长和宽
                col_img.append(col)
        col_img = np.array(col_img)
        return col_img

    def forward(self, input_array):
        # 把卷积核转为列向量，列数为卷积核数，行数是通道数*卷积核大小
        col_weights = self.weights.reshape((-1, self.output_channels))
        output = np.zeros(self.output_shape)
        self.col_img = []

        for i in range(self.batch_size):
            img = input_array[i][np.newaxis, :]  # 在最前面增加一个维度（即对于一
张图batchsize=1）
            # 对于每张图，都通过im2col变形为：子矩阵数*卷积核hw
            col_img_i = self.im2col(img)
            self.col_img.append(col_img_i)
            # 变形后的img和列卷积核相乘，结果转为原来形状。output=conv(img,kernel)
            output[i] = np.reshape(np.dot(col_img_i, col_weights) +
self.bias, self.delta[0].shape)
        self.col_img = np.array(self.col_img)
        return output
```

（3）卷积层的反向传播

w的梯度+=输入图片im2col后转置，与该图片的delta相乘。因为前向计算out时，每个位置卷积后都加上了b，所以b的梯度+=每张图片delta的和。

next_delta是delta和翻转180度后的卷积核进行卷积的结果。而为了让next_delta和img形状相同，需要将delta用pad扩充，使得结果符合形状，然后才能进行卷积。

```
    def backward(self, delta):
        self.delta = delta
        # 将delta转为batch*输出的hw*卷积核数的形状，而hw=子矩阵数
        col_delta = np.reshape(delta, (self.batch_size, -1,
self.output_channels))
        # 分别计算w和b的梯度
        for i in range(self.batch_size):
            self.w_grad += np.dot(self.col_img[i].T,
col_delta[i]).reshape(self.weights.shape)  # 输入与delta做卷积
        self.b_grad += np.sum(col_delta, axis=(0, 1))  # 每列的和，拼成一个向量

        # 计算向前传播的next_delta=conv(delta,np.rot90(kernel,2))，为了让
next_delta和img形状相同，需要将delta用pad扩充，使得结果符合形状
        pad_delta = np.pad(array=self.delta,
                           pad_width=((0, 0), (self.kernel_size - 1,
self.kernel_size - 1),
                                      (self.kernel_size - 1,
self.kernel_size - 1), (0, 0)),
                           mode='constant',
                           constant_values=0)
        col_pad_delta = np.array([self.im2col(pad_delta[i][np.newaxis, :])
for i in range(self.batch_size)])
        # 将卷积核翻转180度
        flip_weights = np.flipud(np.fliplr(self.weights))  # fliplr左右翻转，
flipud上下翻转
        flip_weights = flip_weights.swapaxes(2, 3)  # 交换2,3维
        col_flip_weights = flip_weights.reshape([-1, self.input_channels])
        next_delta = np.dot(col_pad_delta, col_flip_weights)
        next_delta = np.reshape(next_delta, self.input_shape)
        return next_delta
```

更新参数：将w和b减去学习率与各自梯度的乘积

```
    def update(self, learning_rate):
        self.weights -= learning_rate * self.w_grad
        self.bias -= learning_rate * self.b_grad

        self.w_grad = np.zeros(self.w_grad.shape)
        self.b_grad = np.zeros(self.b_grad.shape)
```

2.池化层

池化层的目的是降低图片维度，降低卷积层对位置的过度敏感。池化方法有两种：max pooling和average pooling，本次实验采用前者。

（1）池化层的成员变量

```python
# 池化层
class PoolingLayer(object):
    def __init__(self, input_shape, kernel_size=2, stride=2):
        self.input_shape = input_shape
        self.batch_size = input_shape[0]
        self.kernel_size = kernel_size
        self.stride = stride

        self.output_channels = input_shape[-1]
        # 注意如果除不开的话，输出时直接舍掉后面的行和列
        self.output_shape = [self.batch_size, input_shape[1] // stride,
input_shape[2] // stride, self.output_channels]
        # 存储池化位置
        self.index = np.zeros(self.input_shape)
```

（2）池化层的前向传播

max pooling前向传播的做法和卷积层类似，不过是在每个子矩阵中找到最大值，填入输出矩阵的相应位置。需要使用self.index来记录下来最大值的位置，通过将这个位置进行标记来实现，这将被在反向传播中使用。

另外需要注意的是，输出矩阵的边长是输入矩阵的边长除以步长，如果除不开的话就舍弃输入矩阵的剩余部分。

```python
    # 输入(batch_size,h,w,kernel_num)
    def forward(self, input_array):
        output = np.zeros(self.output_shape)
        for b in range(self.batch_size):
            for k in range(self.output_channels):
                for i in range(0, input_array.shape[1], self.stride):
                    """注意选定子矩阵时，如果i + self.kernel_size超出了输入尺寸，也
会被算进来，所以要提前排除这种情况"""
                    if i + self.kernel_size > input_array.shape[1]:
                        break
                    for j in range(0, input_array.shape[2], self.stride):
                        if j + self.kernel_size > input_array.shape[2]:
                            break
                        # 选定子矩阵
                        subarray = input_array[b, i:i + self.kernel_size,
j:j + self.kernel_size, k]
                        # 把最大值存入output相应位置
                        output[b, i // self.stride, j // self.stride, k] =
subarray.max()
                        index = np.argmax(subarray)  # 最大值的位置，注意是x*y
                        self.index[b, i + index // self.stride, j + j %
self.stride, k] = 1
                        # 除以步长，并取模得到坐标x和y.将坐标标记为1，后面反向传播时
用
        return output
```

（3）池化层的反向传播

将delta扩充成输入矩阵的形状，并将index没有记录的位置置为0，这个过程被称为upsample。

```
    def backward(self, delta):
        next_delta = np.repeat(np.repeat(delta, self.stride, axis=1),
self.stride, axis=2) * self.index
        return next_delta
```

由于池化层没有参数，所以无需进行更新操作。

3.全连接层

池化层的输出依然是矩阵形式，而我们如果需要十分类，就需要得到一个10维的向量，哪个维度的结果最大就分入哪类，这就是全连接层的功能，通过输入矩阵与一个w矩阵相乘来变形。

（1）全连接层层的成员变量

output_num：分类的类数

input_len：每张图片需要转为一个列向量，长度为宽x高x通道数

self.weights：是一个(input_len, output_num)形状的矩阵，输入矩阵右乘它即可得到一个output_num维的向量

```
# 全连接层
class FullyConnectedLayer(object):
    def __init__(self, input_shape, output_num):
        self.input_shape = input_shape
        self.batch_size = self.input_shape[0]
        self.output_shape = (self.batch_size, output_num)

        input_len = reduce(lambda x, y: x * y, input_shape[1:])
        self.weights = np.random.standard_normal((input_len, output_num)) /
100
        self.bias = np.random.standard_normal(output_num) / 100

        self.w_grad = np.zeros(self.weights.shape)
        self.b_grad = np.zeros(self.bias.shape)
```

（2）全连接层的前向传播

首先将一个batch中每个输入矩阵转为input_len长度的行向量，这个过程叫做压平（flatten）。然后将这个行向量与self.weights矩阵相乘，加上偏置即可得到output_num维的输出。

```
    def forward(self, input_array):
        self.flatten_x = input_array.reshape((self.batch_size, -1))  # 将输入
flatten
        output = np.dot(self.flatten_x, self.weights) + self.bias
        return output
```

（3）全连接层的反向传播

首先把flatten后的行向量转为(input_shape,1)形状的列向量，与(1,output_num)形状的输出相乘，得到w的梯度。b的梯度就是delta。

```
    def backward(self, delta):
        for i in range(self.batch_size):
            col_x = self.flatten_x[i][:, np.newaxis]  # 把行向量转为(input_shape,1)形状的列向量
            delta_i = delta[i][:, np.newaxis].T  # (1,output_num)
            self.w_grad += np.dot(col_x, delta_i)
            self.b_grad += delta_i.reshape(self.bias.shape)

        next_delta = np.dot(delta, self.weights.T)
        next_delta = np.reshape(next_delta, self.input_shape)
        return next_delta
```

更新参数：

```
    def update(self, learning_rate):
        self.weights -= learning_rate * self.w_grad
        self.bias -= learning_rate * self.b_grad

        self.w_grad = np.zeros(self.w_grad.shape)
        self.b_grad = np.zeros(self.b_grad.shape)
```

4.激活函数sigmoid和relu

本次实验中我尝试采用了两种激活函数sigmoid和relu。relu结果见基础要求，sigmoid结果见提高要求。

sigmoid

```
class Sigmoid(object):
    def __init__(self, input_shape):
        self.output_shape = input_shape
        self.output = np.zeros(self.output_shape)
        self.delta = np.zeros(input_shape)
        self.x = np.zeros(input_shape)

    def forward(self, x):
        self.x = x
        self.output = 1 / (1 + np.exp(-self.x))
        return self.output

    def backward(self, delta):
        self.delta = delta * self.output * (1 - self.output)
        return self.delta
```

relu

```
class Relu(object):
    def __init__(self, shape):
        self.delta = np.zeros(shape)
        self.x = np.zeros(shape)
        self.output_shape = shape

    def forward(self, x):
        self.x = x
        return np.maximum(x, 0)  # relu就是把数组中负数都变成0
```

```
    def backward(self, delta):
        self.delta = delta
        self.delta[self.x < 0] = 0  # x中小于0元素对应的项的delta=0
        return self.delta
```

5.交叉熵代价函数softmax

loss：计算本次训练的误差，作为评估标准

predict：将全连接层的线性输出转为[0,1]上的概率分布

backward：反向传播，为全连接层提供delta

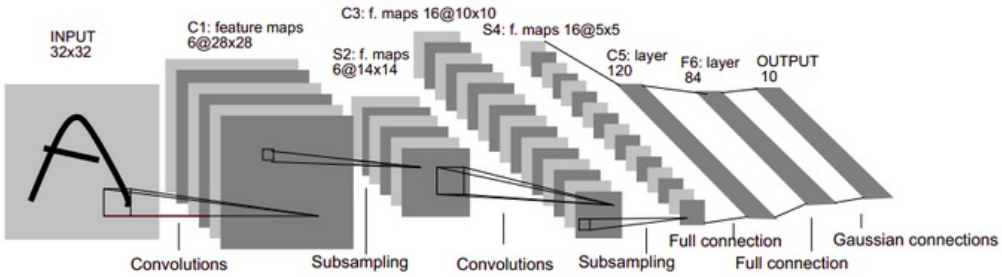
注意计算幂时，先减去最大值，否则计算exp时会越界

对于一个分类任务，zi的理想概率非0即1，因此loss=(0或者1)-ln(Softmax)

```
class Softmax(object):
    def __init__(self, input_shape):
        self.softmax = np.zeros(input_shape)
        self.delta = np.zeros(input_shape)
        self.batch_size = input_shape[0]

    # 计算评估值loss
    def loss(self, output, labels):
        self.labels = labels
        self.predict(output)
        loss = 0
        for i in range(self.batch_size):
            loss += np.log(np.sum(np.exp(output[i]))) - output[i][
                self.labels[i]]  # 设labels[i]=2，那么如果output[i][2]=1，则loss=0，否则=1
        return loss

    # 将全连接层的线性输出转为概率分布
    def predict(self, output):
        self.softmax = np.zeros(output.shape)
        exp_output = np.zeros(output.shape)
        for i in range(self.batch_size):
            output[i:, ] -= np.max(output[i])  # 先减去最大值，否则计算exp时会越界
            exp_output[i] = np.exp(output[i])
            self.softmax[i] = exp_output[i] / np.sum(exp_output[i])
        return self.softmax

    # 生成反向传播的delta
    def backward(self):
        self.delta = self.softmax.copy()
        for i in range(self.batch_size):
            self.delta[i][self.labels[i]] -= 1
        return self.delta
```

6.训练

我的卷积神经网络结构是：输入-卷积层1-激活函数层1-池化层1-卷积层2-激活函数层2-池化层2-全连接层-softmax层，如图所示：



设置batch_size=50，学习率=1e-4。第一个卷积层采用3个5x5的卷积核，池化层采用2x2的核，第二个卷积层采用3个3x3的卷积核。

```
def train(X_train, y_train, X_test, y_test):
    output_types = 3  # 分类数
    batch_size = 50
    learning_rate = 1e-4

    train_batch_num = len(y_train) // batch_size  # 训练集batch数量
    test_batch_num = len(y_test) // batch_size
    print("train_batch_num:", train_batch_num)
    print("test_batch_num:", test_batch_num)

    """准备模型"""
    conv1 = ConvolutionLayer((batch_size, 28, 28, 1), output_channels=3, kernel_size=5)
    relu1 = Relu(conv1.output_shape)
    pool1 = PoolingLayer(relu1.output_shape)

    conv2 = ConvolutionLayer(pool1.output_shape, output_channels=3, kernel_size=3)
    relu2 = Relu(conv2.output_shape)
    pool2 = PoolingLayer(relu2.output_shape)

    full = FullyConnectedLayer(pool2.output_shape, output_num=output_types)
    soft = Softmax(full.output_shape)

    train_loss = []
    train_accuracy = []
    test_loss = []
    test_accuracy = []

    start = time.time()
    for epoch in range(5):
        """开始训练"""
        epoch_train_loss = 0  # 这一epoch的平均loss
        epoch_train_accuracy = 0  # 这一epoch的平均accuracy
        for batch in tqdm(range(train_batch_num)):
            imgs = X_train[batch * batch_size:(batch + 1) * batch_size].reshape((batch_size, 28, 28, 1))
            labels = y_train[batch * batch_size:(batch + 1) * batch_size]
            # 前向传播
            conv1_out = conv1.forward(imgs)
            relu1_out = relu1.forward(conv1_out)
```

```
            pool1_out = pool1.forward(relu1_out)

            conv2_out = conv2.forward(pool1_out)
            relu2_out = relu2.forward(conv2_out)
            pool2_out = pool2.forward(relu2_out)
            full_out = full.forward(pool2_out)
            epoch_train_loss += soft.loss(full_out, labels)

            """训练集acc"""
            cnt = 0
            for j in range(batch_size):
                if labels[j] == np.argmax(full_out[j]):
                    cnt += 1
            accuracy = cnt / batch_size
            epoch_train_accuracy += accuracy
            # if batch % 10 == 0:
            #     print("train batch %d, batch_acc:%t"%(batch, accuracy))

            # 反向传播
            soft.backward()
            conv1.backward(
                relu1.backward(
                    pool1.backward(
                        conv2.backward(
                            relu2.backward(
                                pool2.backward(
                                    full.backward(soft.delta)))))))
            full.update(learning_rate)
            conv2.update(learning_rate)
            conv1.update(learning_rate)

        end = time.time()
        train_loss.append(epoch_train_loss / train_batch_num)
        train_accuracy.append(epoch_train_accuracy / train_batch_num)
```

测试：依然是把测试集按照batch输入网络，比对结果和labels的差异

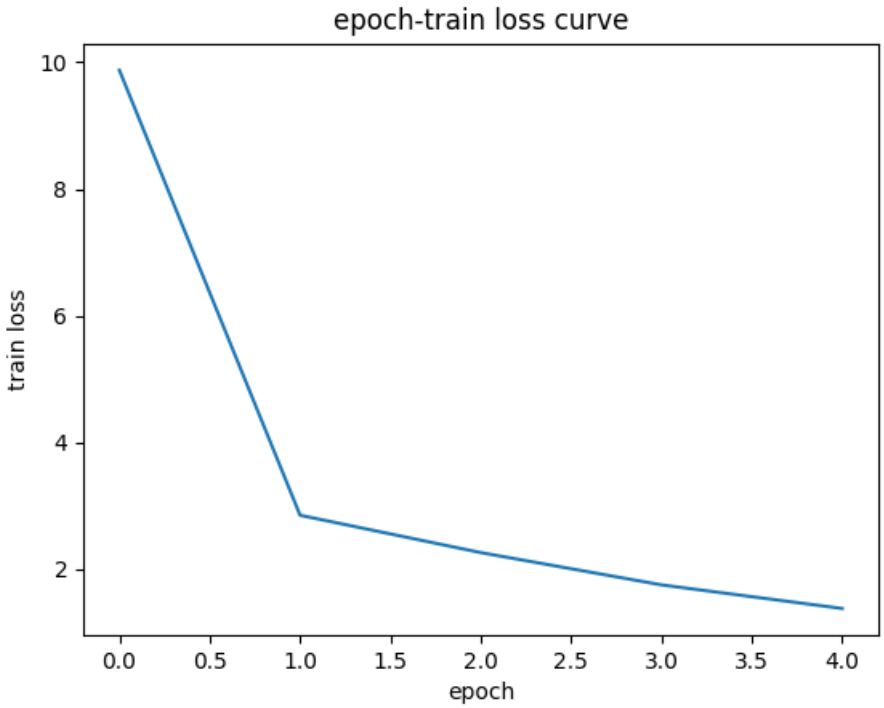
```
        """开始测试"""
        epoch_test_loss = 0
        epoch_test_accuracy = 0
        for batch in range(test_batch_num):
            imgs = X_test[batch * batch_size:(batch + 1) *
batch_size].reshape((batch_size, 28, 28, 1))
            labels = y_test[batch * batch_size:(batch + 1) * batch_size]

            conv1_out = conv1.forward(imgs)
            relu1_out = relu1.forward(conv1_out)
            pool1_out = pool1.forward(relu1_out)

            conv2_out = conv2.forward(pool1_out)
            relu2_out = relu2.forward(conv2_out)
            pool2_out = pool2.forward(relu2_out)
            full_out = full.forward(pool2_out)
            epoch_test_loss += soft.loss(full_out, labels)
            """测试集acc"""
            cnt = 0
            for j in range(batch_size):
```

```
                if labels[j] == np.argmax(full_out[j]):
                    cnt += 1
            accuracy = cnt / batch_size
            epoch_test_accuracy += accuracy
            print("test batch %d accuracy:%f" % (batch, accuracy))

        test_loss.append(epoch_test_loss / test_batch_num)
        test_accuracy.append(epoch_test_accuracy / test_batch_num)
        print(end - start)

    return train_loss, train_accuracy, test_loss, test_accuracy
```

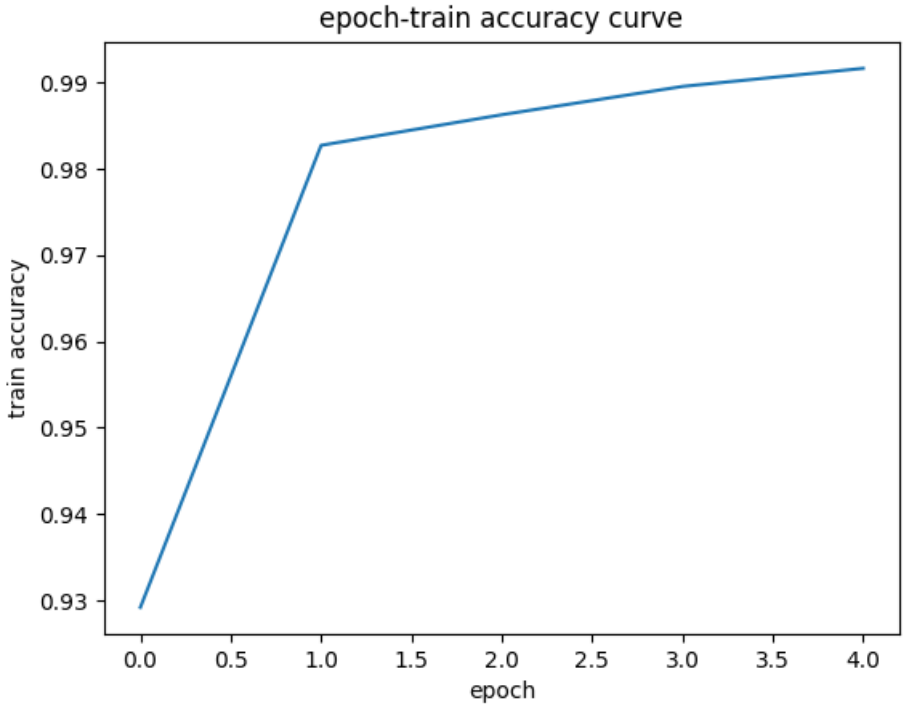
四、实验结果

训练误差、训练精度、测试误差和测试精度曲线

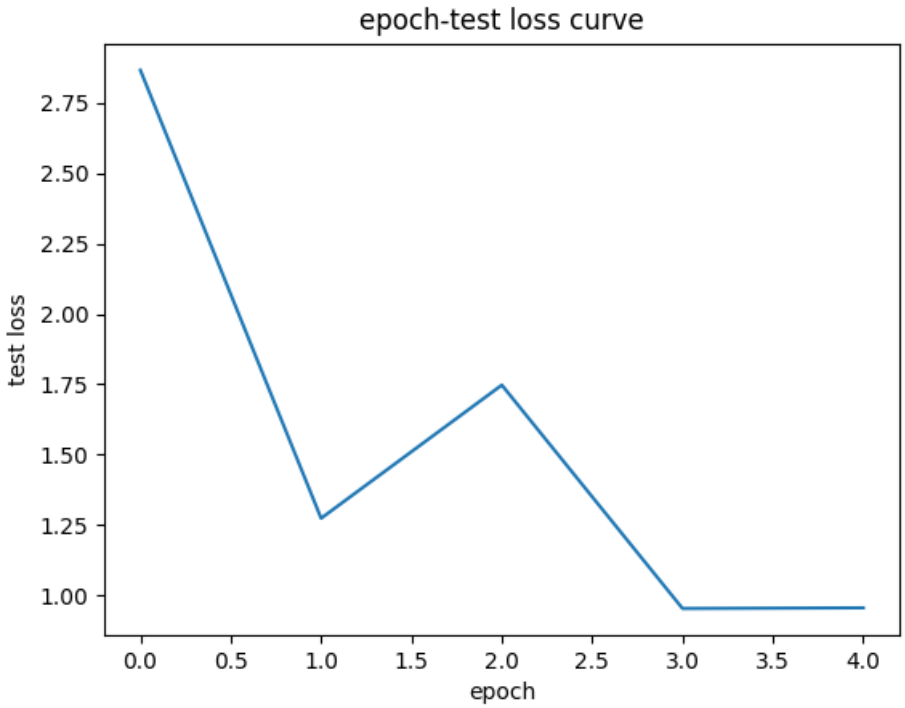
共进行5个epoch，训练误差分别为：[9.874170738832447, 2.849579035983718, 2.26006752359774, 1.750511236442889, 1.3779564118476204]



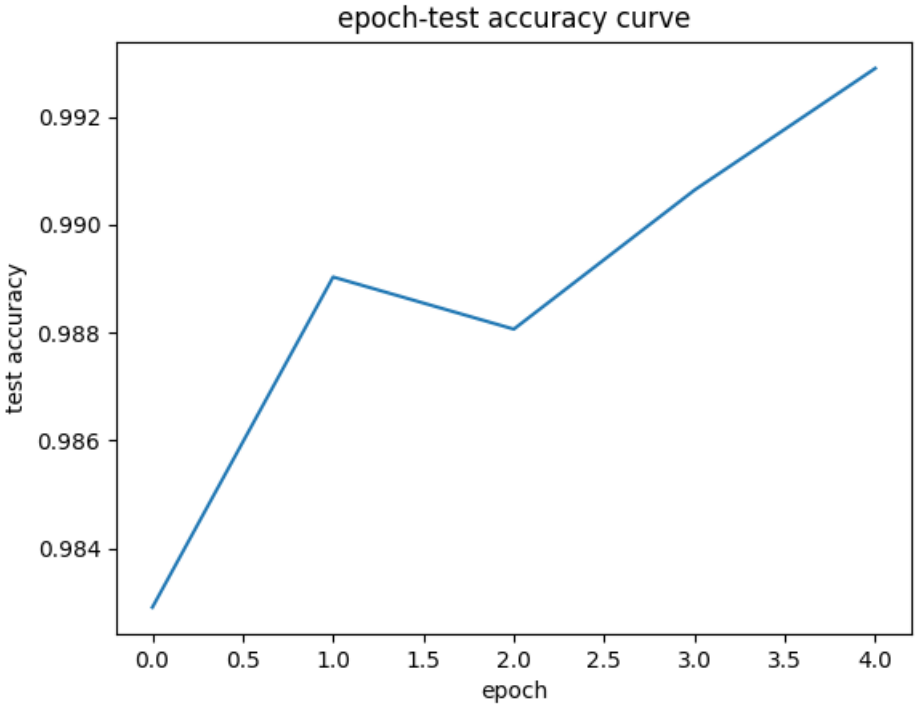
训练精度：[0.9291935483870959, 0.9827419354838706, 0.9862903225806445, 0.9895698924731179, 0.9916666666666666]



测试误差：[2.867351117030974, 1.2735433016310198, 1.7475114435036039, 0.9527523161843718, 0.9550976619867988]



测试精度：[0.9829032258064513, 0.9890322580645158, 0.9880645161290322, 0.9906451612903223, 0.9929032258064514]



之后，我又尝试用该模型进行10分类测试，在10个epoch后，准确度超过94%。

[0.8793000000000001, 0.9155000000000001, 0.9440000000000001, 0.9390999999999999, 0.9437000000000001, 0.936, 0.9503, 0.9478, 0.9459000000000001, 0.9437000000000001]

中级要求

> 对实验过程中可能出现的过拟合和欠拟合现象进行分析

一般来说，模型过于复杂，参数太多，会导致过拟合；过于简单则会导致欠拟合。卷积神经网络的构建中有若干超参数，比如迭代次数epoch，网络输入每组大小batch_size，卷积核数量、大小，池化窗口大小，网络层数等。

如果数据存在噪声，训练的时候把没有代表性的特征也作为训练的样本，那么结果很可能偏离真实值，造成过拟合。这种问题可以通过控制迭代次数、卷积核数量、减少网络层数来解决，因为上面三点都是为了获得图片的特征，并根据特征来调整参数。

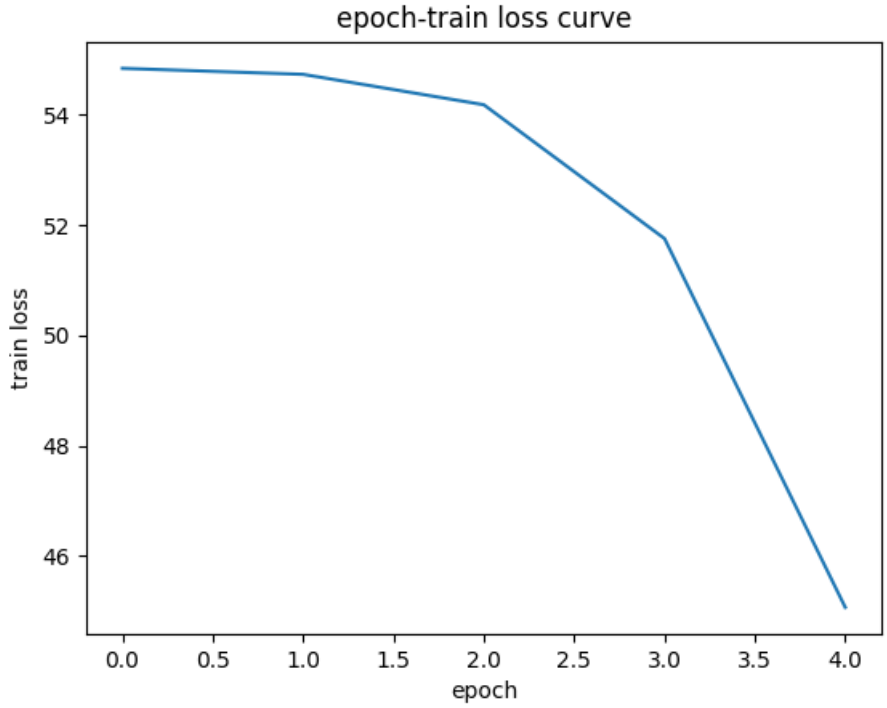
如果无论迭代多少次，loss都无法大幅下降，基本上是出现了欠拟合，增加网络层数可以改善欠拟合。我刚开始训练也经常出现这种情况，终是归结于方向传播算法的实现有问题，改正后两层的卷积神经网络分类能力已经满足一定需要了。

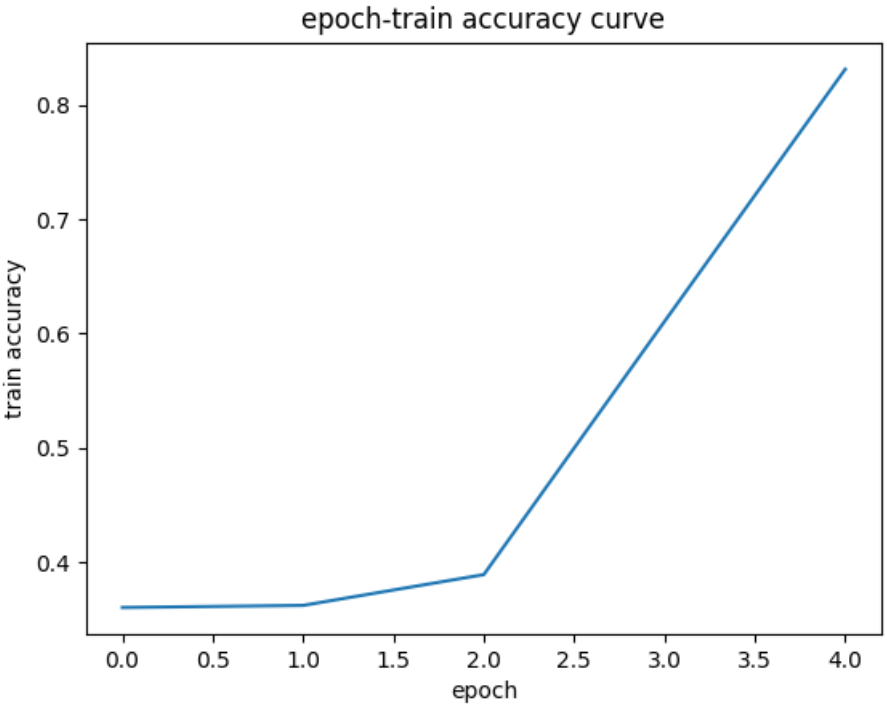
提高要求

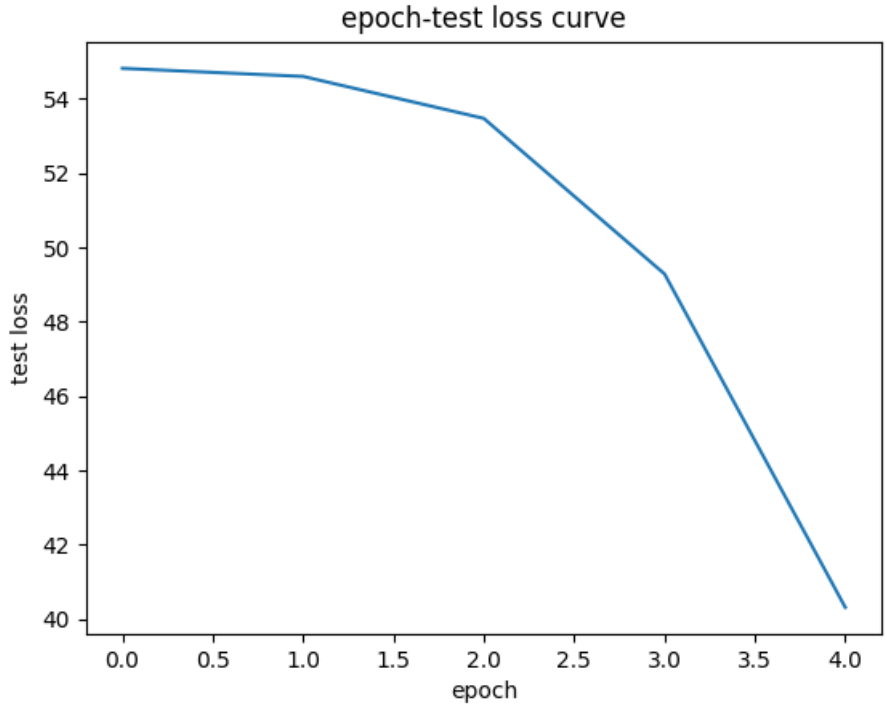
> 尝试更换其他激活函数，对比分析结果

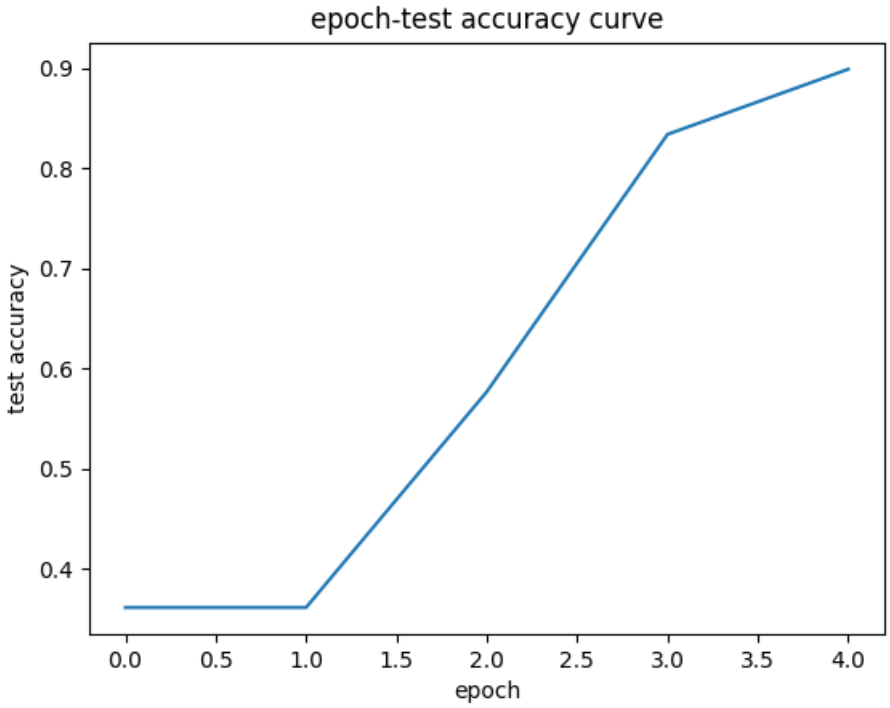
之前基础要求把relu作为激活函数，现在使用sigmoid。进行三分类，可以看到sigmoid的收敛速度明显低于relu，5个epoch之后sigmoid的精度刚刚超过90%，而relu可达99%以上。

另外，可以看到sigmoid的loss下降曲线是凸函数，而relu是凹函数，更加凸显了收敛速度的差异。


epoch-train loss curve
train loss
54
52
50
48
46
0.0
0.5
1.0
1.5
2.0
2.5
3.0
3.5
4.0
epoch


epoch-train accuracy curve
train accuracy
0.8
0.7
0.6
0.5
0.4
0.0
0.5
1.0
1.5
2.0
2.5
3.0
3.5
4.0
epoch





写在最后

因为本实验只由我一人完成，所以我就给自己打10分了（笑）。但实际上我参考了许多文章才得以完成本次作业，包括知乎用户Wziheng的文章，以及各种反向传播的公式推导的博客等，还有和同学之间的交流也让我发现许多问题从而改进。机器学习及应用这门课让我从只是粗浅了解分类、聚类、回归等名词，到动手实现这些经典的算法，这个过程中我对于机器学习的认识加深了许多，希望以后也能够更加深入理解其数学原理。最后感谢老师和助教们一学期的付出！