机器学习实验报告 (四)

2018/12/16

目录

1 实验环境

操作系统	win10
编程语言	python3
编程环境	Jupyter Notebook
报告编写	latex

2 问题描述与分析

问题描述：实现 *bp* 神经网络来测试 *MNIST* 手写数字训练集，并在不同方面对 *bp* 模型做性能对比。

2.1 MNIST 数据集

MNIST 是一个简单的视觉计算数据集，其中包括类似如下的手写数字图片信息（60000 张训练集与 10000 张测试集）：

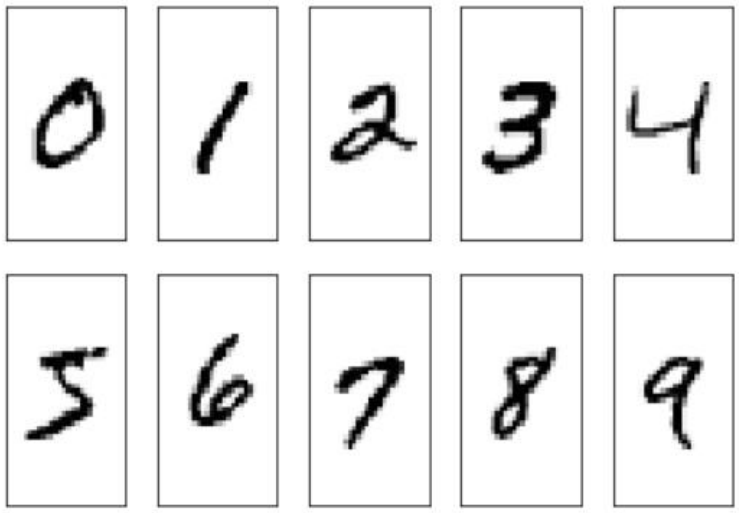

图 1　MNIST 数据集示例

每张图片都是 28*28 大小，除了包含的像素信息以外，每张图片还有一个标签标识所代表的数字。因此我们的首先从处理该数据集入手。

2.2 BP 神经网络

BP 神经网络，是 1986 年由 Rumelhart 和 McClelland 为首的科学家提出的概念，是一种按照误差逆向传播算法训练的多层前馈神经网络。BP 算法是一种有效的多层神经网络学习方法，通过向信号前向输出层传递，误差向输入层反馈，不断调节网络权重值，使得网络的最终输出与期望输出尽可能接近，以达到训练的目的。一般来说，一个多层神经网络由输入层，若干隐含层，输出层构成。

我们定义下标 i，j，k，其中 j 为隐含层神经元，i 与 k 分别为其左右相邻的神经元下标。$e_j(n)$ 为第 j 个神经元在第 n 次迭代的误差，$d_j(n)$ 为第 j 个神经元在第 n 次迭代的期望输出值，$y_j(n)$ 为第 j 个神经元在第 n 次迭代的实际输出值，则的输出层的误差函数如下：

$$e_j(n) = d_j(n) - y_j(n) \tag{1}$$

瞬时误差 $E(n)$ 定义为

$$E(n)=\frac{1}{2}\sum_{j\in C}e_j^2(n) \tag{2}$$

可以知道，每个神经元的输入，等于其上一层各个神经元输出与权重的乘积和，经过激活函数的值即

$$y_j(n)=\varphi_j(v_j(n)) \tag{3}$$

其中

$$v_j(n)=\sum_{i=0}^{p}w_{ji}(n)\,y_i(n) \tag{4}$$

为了调整权重降低误差，我们需要求出误差 $E(n)$ 对于权重的偏导 $\dfrac{\partial E(n)}{\partial w_{ji}(n)}$，结合 (1)(2) 式，由链式法则可得到

$$\frac{\partial E(n)}{\partial w_{ji}(n)}=\frac{\partial E}{\partial v_j}\frac{\partial v_j}{\partial w_{ji}(n)} \tag{5}$$

又因为 (4) 式，我们可以得到：

$$\frac{\partial v_j(n)}{\partial w_{ji}(n)}=\frac{\partial\left[\sum_{i=0}^{p}w_{ji}(n)\,y_i(n)\right]}{\partial w_{ji}(n)}=y_i(n) \tag{6}$$

其中 p 是上一层连接该神经元的个数。定义：

$$\delta_j(n)=-\frac{\partial E(n)}{\partial v_j(n)} \tag{7}$$

当 j 为输出层时，结合 (1)(2)(3) 式可求出：

$$\delta_j(n)=(d_j(n)-y_j(n))\,\varphi_j'(v_j(n)) \tag{8}$$

当 j 为隐含层时，结合 (3) 式与链式法则可得：

$$\delta_j(n)=-\frac{\partial E(n)}{\partial y_j(n)}\frac{\partial y_j(n)}{\partial v_j(n)}=-\frac{\partial E(n)}{\partial y_j(n)}\varphi'(v_j(n)) \tag{9}$$

因为 (4)(7) 式与链式法则，得到：

$$\frac{\partial E(n)}{\partial y_i(n)} = \sum_k \frac{\partial E(n)}{\partial v_k(n)} \frac{\partial v_k(n)}{\partial y_j(n)} = \sum_k \frac{\partial E(n)}{\partial v_k(n)} w_{kj}(n) = -\sum_k \delta_k(n) w_{kj}(n) \tag{10}$$

因此，我们可以得到 $\delta_j(n)$ 表达式为：

$$\delta_j(n) = \begin{cases} (d_j(n) - y_j(n)) \varphi_j'(v_j(n)) & if\ j\ is\ output\ layer \\ \varphi_j'(v_j(n)) \sum_k \delta_k(n) w_{kj}(n) & others \end{cases} \tag{11}$$

综上所述，权重的变化 Δw 可通过下式求出：

$$\Delta w_{ji}(n) = \eta \delta_j(n) y_j(n) \tag{12}$$

其中 η 为学习率

2.3 激活函数

激活函数将非线性特性引入到我们的网络中，没有激活函数的每层都相当于矩阵相乘，叠加了若干层之后实质只是矩阵相乘。常用激活函数有 $Sigmoid, Tanh, ReLU$ 函数等，在此次试验中我们选用了 $Sigmoid$ 和 $Tanh$ 函数进行性能的测试。

2.3.1 Sigmoid 函数

$Sigmoid$ 的公式形式：

$$f(x) = \frac{1}{1 + e^{-x}} \tag{13}$$

其导数形式为 $f'(x) = f(x)(1 - f(x))$。该函数的取值在 0-1 之间，当 x 趋近于负无穷时，y 趋近于 0；当 x 趋近于正无穷时，y 趋近于 1；当 x= 0 时，y=0.5。在 0.5 处为中心对称，并且越靠近 x=0 的取值斜率越大。也由于其导数的性质，很容易就会出现梯度消失的情况，从而无法完成深层网络的训练。

2.3.2 Tanh 函数

$Tanh$ 的公式形式：

$$f(x) = \frac{e^x - e^{-x}}{e^x + e^{-x}} \tag{14}$$

其导数形式为 $f'(x) = 1 - f(x)^2$。$Tanh$ 函数已成为双切正切函数，取值范围为 [-1,1]，关于原点中心对称。tanh 在特征相差明显时的效果会很好，在循环过程中会不断扩大特征效果。与 $Sigmoid$ 相对来说，$Tanh$ 是以 0 为均值的，在实际应用中效果更好一点。

2.4 目标

本次实验准备实现一个较为通用的 bp 神经网络，层数及每层的神经元的个数可以随意设置。通过改变层数，改变神经元个数，改变激活函数的方面来比较该网络的性能。

3 具体实现过程

3.1 MNIST 数据集的导入

在导入所需要的库依赖之后，我们首先要做的就是讲 MNIST 数据集读入。通过参考网上相关资料，我在这定义了 $load_mnist_train(path)$ 和 $load_mnist_test(path)$ 两个方法，分别用于加载训练集与测试集

```
# 加载数据
def load_mnist_train(path):
    labels_path = os.path.join(path,'train-labels.idx1-ubyte')
    images_path = os.path.join(path, 'train-images.idx3-ubyte')
    with open(labels_path, 'rb') as lbpath:
        magic, n = struct.unpack('>II',lbpath.read(8))
        labels = np.fromfile(lbpath, dtype=np.uint8)
```

```
8       with open(images_path, 'rb') as imgpath:
9           magic, num, rows, cols = struct.unpack('>IIII',
10          imgpath.read(16))
11          images = np.fromfile(imgpath,
12          dtype=np.uint8).reshape(len(labels), 784)
13      return images, labels
14  def load_mnist_test(path):
15      labels_path = os.path.join(path,'t10k-labels.idx1-ubyte')
16      images_path = os.path.join(path, 't10k-images.idx3-ubyte')
17      with open(labels_path, 'rb') as lbpath:
18          magic, n = struct.unpack('>II',lbpath.read(8))
19          labels = np.fromfile(lbpath, dtype=np.uint8)
20      with open(images_path, 'rb') as imgpath:
21          magic, num, rows, cols = struct.unpack('>IIII',
22          imgpath.read(16))
23          images = np.fromfile(imgpath,
24          dtype=np.uint8).reshape(len(labels), 784)
25      return images, labels
26  X_train,Y_train = load_mnist_train('E:\学习\大三\机器学习\MNIST')
27  X_test,Y_test = load_mnist_test('E:\学习\大三\机器学习\MNIST')
```

3.2 bp 神经网络的实现

首先我们先定义激活函数及其导数，为接下来做准备。首先定义 $Tanh$ 函数及导数

```
1   def tanh(x):
2       return (np.exp(x)-np.exp(-1*x))/(np.exp(x)+np.exp(-1*x))
3   def dtanh(y):
4       return 1.0 - y*y
```

同时我们定义 $Sigmoid$ 函数及导数，对比使用。

```
def sigmoid(x):
     return  1.0 / (1.0 + np.exp(-1*x))
def dsigmoid(y):
    return y*(1-y)
```

然后定义 $trans(li)$ 函数，输入为具体数字，便于将 $lable$ 转换为我们将要用到的 10 x1 维的向量。

```
def trans(li):
    re_list =[0.0]*10
    re_list[li]=1
    return re_list
```

接下来定义神经网络类 NN，在第一次实现中，权重的更新等操作均采用的 for 循环来实现的，发现运算速度很慢，于是在第二次实现中将所有显示 for 循环均替换成矩阵运算，大大提高了运算速度，以下是替换后的第二版本。

```
class NN:
    def __init__(self, ni, nh_li, no):
        # 输入层、隐藏层列表、输出层
        self.ni = ni
        self.nh_li = nh_li
        self.nh_len = len(nh_li)
        self.no = no
        #输入层到第一层隐含层，。。。，最后一层隐含层到输出层
        #总长度为 nh_len+1
        self.w = []

        self.w.append(np.random.normal(0.0,
```

```
13          pow(self.nh_li[0],-0.5), (self.nh_li[0],self.ni)))
14          for i in range(self.nh_len-1):
15              self.w.append(np.random.normal(0.0,
16                  pow(self.nh_li[i+1],-0.5),
17                  (self.nh_li[i+1],self.nh_li[i])))
18          self.w.append(np.random.normal(0.0,
19                  pow(self.no,-0.5),
20                  (self.no,self.nh_li[self.nh_len-1])))
21
22      def oneRound(self, inputs,targets,N):
23          targets = np.array(targets,ndmin=2).T
24          # 激活输入层
25          inputs = np.array(inputs,ndmin=2).T/255
26          # 隐含层输出
27          # 输入层到第一个隐含层，。。。。，最后一个隐含层到输出层
28          #总长度为 nh_len+1
29          outputs = []
30          outputs.append(tanh(np.dot(self.w[0],inputs)))
31          for i in range(self.nh_len-1):
32              outputs.append(tanh(np.dot(self.w[i+1],outputs[i])))
33          # 输出层输出
34          outputs.append(tanh(np.dot(self.w[self.nh_len],
35          outputs[self.nh_len-1])))
36
37          #输出层，最后一个隐含层 。。。 第一个隐含层
38          #总长度为 nh_len+1
39          deltas = []
40          #计算输出层的误差
41          output_deltas = dtanh(outputs[self.nh_len])*
```

```
42                  (targets - outputs[self.nh_len])
43          deltas.append(output_deltas)
44          for i in range(self.nh_len):
45              deltas.append(dtanh(outputs[self.nh_len-1-i])*
46                      np.dot(self.w[self.nh_len-i].T, deltas[i]))
47
48          for i in range(self.nh_len):
49              self.w[self.nh_len-i] += N*np.dot(deltas[i],
50                      np.transpose(outputs[self.nh_len-i-1]))
51          self.w[0] += N*np.dot(deltas[self.nh_len],
52                  np.transpose(inputs))
53
54          return outputs[self.nh_len]
55
56      def test(self, inputs_list,Y):
57          le = len(Y)
58          ans = 0
59          for i in range(le):
60              inputs = np.array(inputs_list[i],ndmin=2).T/255
61              # 输入层到第一个隐含层，。。。。，最后一个隐含层到输出层
62              #总长度为 nh_len+1
63              outputs = []
64              outputs.append(tanh(np.dot(self.w[0],inputs)))
65              for j in range(self.nh_len-1):
66                  outputs.append(tanh(np.dot(self.w[j+1],outputs[j])))
67              # 输出层输出
68              outputs.append(tanh(np.dot(self.w[self.nh_len],
69                      outputs[self.nh_len-1])))
70              if np.argmax(outputs[self.nh_len])==Y[i]:
```

```
71                  ans+=1
72          print('准确率',ans/le)
73          return ans/le
74
75      def weights(self):
76          return self.w
77
78      def train(self, X, Y,iterations=100, N=0.01):
79          len_total = len(Y)
80          for i in range(iterations):
81              print("第",i,"轮")
82              for p in range(len_total):
83                  inputs = X[p]
84                  target = Y[p]
85                  targets = trans(target)
86                  self.oneRound(inputs,targets,N)
```

在实现多层神经网络时，我采用列表来存储隐含层的信息，包括权重、误差、权重变化值等。$self.w = []$ 存储各节点间的权重，$outputs = []$ 存储各节点的输出，$deltas = []$ 存储变化值。

在初始函数中，传入参数为输入层、隐含层、输出层神经元个数，其中隐含层为列表的形式。我先将各个层所包含的神经元个数记录下来，并使用 $np.random.normal$ 方法对权重初始化。

$oneRound(self, inputs, targets, N)$ 函数为一次权重更新的完整过程。首先将输入 $inputs$ 与 $targets$ 转换为我们需要的格式，通过前面推导的 (3) 式求出每个神经元的输出，存储到 $outputs$ 中，通过 (11) 式求出 δ 存入到 $deltas$ 中，通过 (12) 式求出 Δw 并更新权值。

$test(self, inputs_list, Y)$ 方法通过输入测试集与对应标签，计算输出，最终返回准确率

$weights(self)$ 方法较为简单，将权重返回

$train(self, X, Y, iterations = 100, N = 0.01)$ 方法用于训练神经网络，调用了 $oneRound$ 方法。这里采用 $Stochasticon - line$ 方法进行权重更新。具体的调用方法示例如下 (两层隐含层)：

```
num=60000
n = NN(784, [100,40], 10)
n.train(X_train[:num],Y_train[:num],iterations=10)
n.test(X_test,Y_test)
```

4 结果分析

结果将从层数、神经元个数、激活函数、输入归一化、学习率等方面来进行对比。我们约定神经网络的层数为隐含层与输出层之和，即当只有输入层、一层隐含层、输出层时，我们说该神经网络的层数为 2。相关数据放在附录中

4.1 隐含层层数对比

我们尝试了当激活函数及学习率相同时，层数和分别为 2(隐含层为 [90])、3(隐含层为 [100,40])、4(隐含层为 [100,50,20]) 层时，训练一轮（即 60000 个数据）后，在测试集上的准确率，整理得到如下图像：

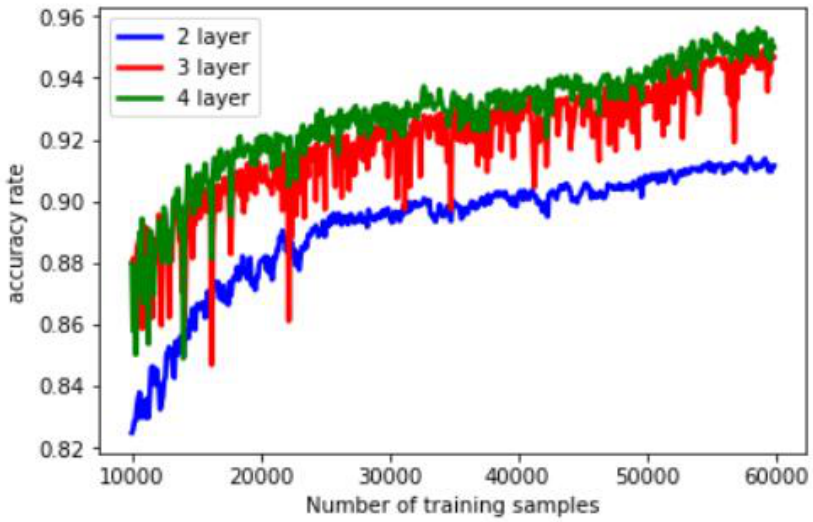

图 2　神经网络为 2、3、4 层，训练准确率对比

为便于观察，这里我们下标定为从 10000 开始。可以很清晰的看到，当我们的训练次数增大时，我们的准确率在波动提高；并且在训练次数相同时，层数大的神经网络得到的准确率更高一些。因为有三层网络时准确率已经很高，当我们增加到四层时提高的幅度不是很大。不难理解，当层数较大时，网络更加复杂，能学习到的东西也更多。因训练次数较少，并没有出现过拟合现象。

4.2　隐含层神经元个数对比

在此我们选择了单层隐含层时且学习率相同时，300，89，30 个隐含神经元，训练一轮（即 60000 个数据）后，在测试集上的准确率，整理得到如下图像：

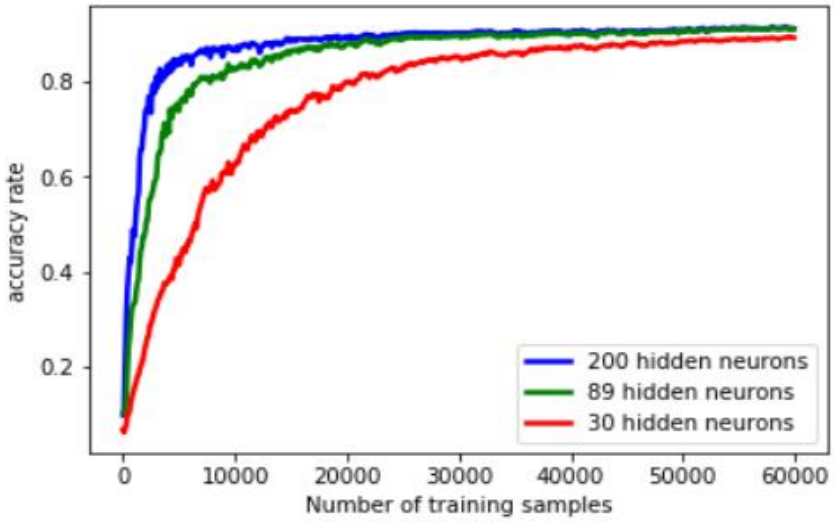

图 2　　单层隐含层为 300，89,30 时，训练准确率对比

很明显可以看出，在 $[0, 5000]$ 区间内，随着神经元个数的提高，模型的学习的效率也在增加。在 200 个隐含神经元模型中，在训练 5000 个样本后准确率就达到 0.8 的水平，而在 30 个隐含神经元模型中需要训练 60000 个样本后才能到达。相对于学习速度，相同隐含层不同神经元模型接近稳定时的准确率差别不大。

4.3　激活函数对比

在此我们选择了单层隐含层时且学习率相同时，使用 $sigmoid$ 函数与 $tanh$ 函数作为激活函数，训练一轮（即 60000 个数据）后，在测试集上的准确率，整理得到如下图像：

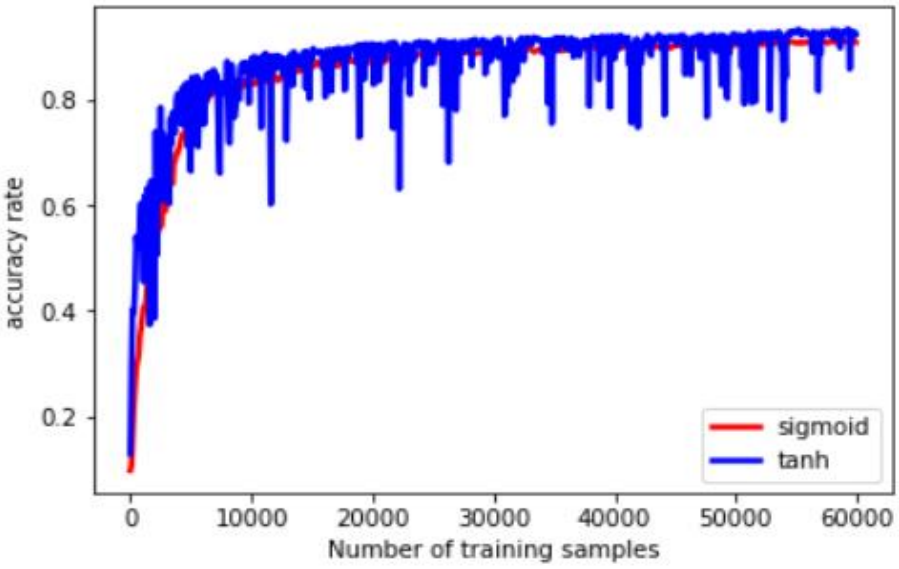

图 2　不同激活函数，训练准确率对比

通过上图可以看出，在测试集中，使用 $sigmoid$ 函数或 $tanh$ 函数作为激活函数求准确率效果比较接近，但可以注意到，相对 $sigmoid$ 函数来说，使用 $tanh$ 函数作为激活函数所求准确率波动较大，不稳定。经过搜查资料与思考，猜测是因为不同的激活函数对于学习率的要求也不同，样例中所设定的学习率 $\eta = 0.01$，这对 $sigmoid$ 函数似乎不错，但对 $tanh$ 函数来说可能偏大，于是接下来对用不同学习率对 $tanh$ 函数进行试验。

4.4　学习率对比

为了探究上面的疑问，我设置单层隐含层，激活函数为 $tanh$ 函数的模型在 $\eta = 0.01\ 0.005, 0.001$ 下，训练一轮（即 60000 个数据）后，在测试集上的准确率，整理得到如下图像：

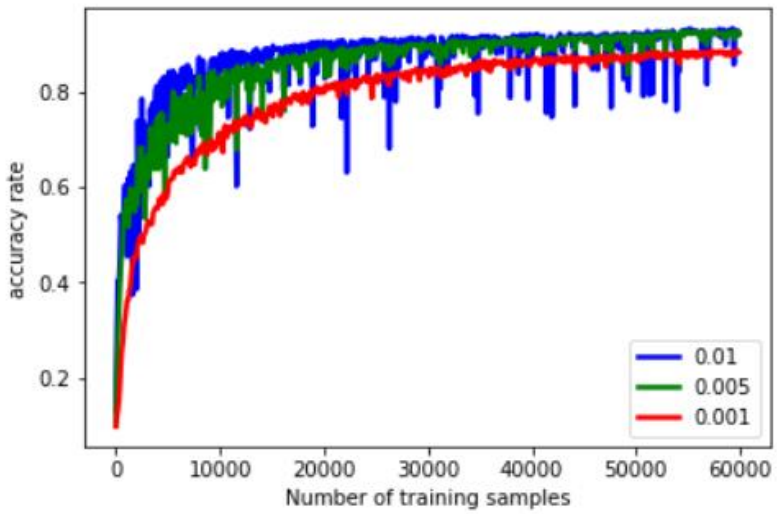

图 2　不同 η 下，训练准确率对比

通过上图可以发现，随着学习率的降低，得到的准确率的图像也越来越平滑，准确率也越来越稳定。不过也可以发现，当准确率较大时，我们可以更快的到达较高的准确率水平，因此，我们可以通过动态改变学习率来提高模型的性能，比如随着准确度的提高而减小学习率的值，让其更快的稳定下来。

4.5　正则化对比

可以注意到，我在处理输入层时，对输入进行了如下处理，以提高整个模型的性能。

```
1 inputs = np.array(inputs,ndmin=2).T/255
```

其中 255 为输入值得最大值，这样将输入规范到 $[0,1]$。下图为是否进行这样规范处理，训练一轮（即 60000 个数据）后，在测试集上的准确率，整理得到如下图像：

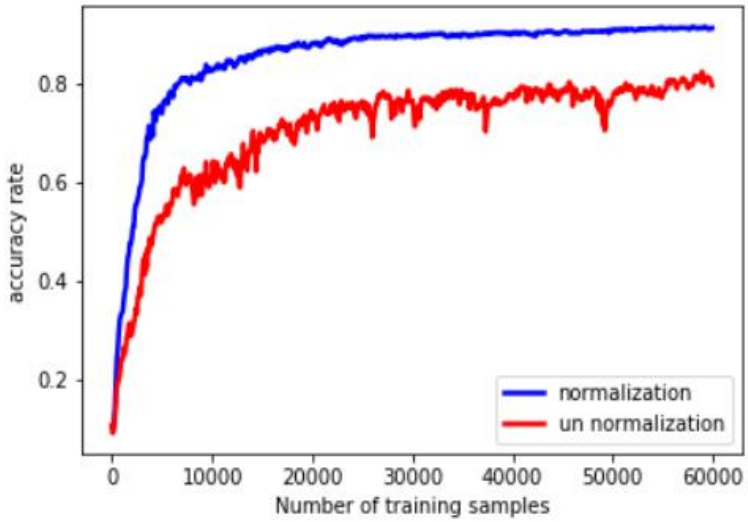

图 2 是否规范化输入，训练准确率对比

可以看到，当将输入规范以后，在收敛速度，稳定性，准确率方面都有着很好的表现。因为我们激活函数使用的为 $sigmoid$ 函数与 $tanh$ 函数，当输入值过大时，输出值非常接近于 1，并且导数值非常接近于 0，出现梯度消失现象。这非常不利于我们之后的梯度下降法来更新权重。

4.6 效果

综和以上发现的问题，最终使用输入层 784 个神经元，三层隐含层 [100,50,20] 个神经元, 输出层 10 个神经元，学习率 $\eta = 0.005$，$tanh$ 函数作为激活函数, 训练 200 轮左右可以达到 0.99 的准确率。

4.7 不足

经过查资料，发现对于多分类问题一般使用 $softmax$ 函数，本次试验并未使用。同时，训练方法使用的 $Stochasticon - line$ 方法，可以在后续工作中尝试使用 $minibatch$ 等方法。激活函数尝试使用其他函数。

5 总结与收获

通过这次实验，我对神经网络有了更加深刻地认识，对各种优化方法也有了一些自己的见解。手写数字识别在之前看来是一件非常复杂而高深的事情，然而在自己实现并的到较好的结果后发现其实原理并不复杂，也增强了自己的信心。