使用BP神经网络算法进行手写数字识别

1. 样本来源

样本源自THE MNIST DATABASE of handwritten digits(<http://yann.lecun.com/exdb/mnist/>)，这个样本是NIST的一个子库，在NIST的基础之上进行了居中和大小的统一。其中，里面包含有6万个训练样本和1万个测试样本。数字的具体样子如下图1所示。



图1　MNIST数据库数字样例图

可以看到，这些数字并不是规矩的，而是在不同角度进行旋转的，所以在识别之前可以进行一些旋转的预处理。本次实验直接采用的就是这些数字进行分类，并没有进行任何的预处理。

针对每个数字图像而言，每个数字就是一个28\*28像素的图片，使用一个28\*28大小的矩阵存储，矩阵的每个值代表的是当前像素点的亮度值，下图2所示的是一个数字8的具体显示方式：

左图：数字8的亮度图 右图：左图在右边对应的矩阵

图2　数字8的具体显示方式

1. 人工神经网络

人工神经网络是一种模仿生物神经网络的结构和功能的数学模型或计算模型。神经网络由大量的人工神经元联结进行计算。大多数情况下人工神经网络能在外界信息的基础上改变内部结构，是一种自适应系统。现代神经网络是一种非线性统计性数据建模工具，常用来对输入和输出间复杂的关系进行建模，或用来探索数据的模式。

神经网络是一种运算模型，由大量的节点（或称“神经元”，或“单元”）和之间相互联接构成。每个节点代表一种特定的输出函数，称为激励函数（activation function）。每两个节点间的连接都代表一个对于通过该连接信号的加权值，称之为权重（weight），这相当于人工神经网络的记忆。网络的输出则依网络的连接方式，权重值和激励函数的不同而不同。而网络自身通常都是对自然界某种算法或者函数的逼近，也可能是对一种逻辑策略的表达。

一种常见的多层结构的前馈网络（Multilayer Feedforward Network）由三部分组成，

输入层（Input layer），众多神经元（Neuron）接受大量非线形输入信息。输入的信息称为输入向量。输出层（Output layer），信息在神经元链接中传输、分析、权衡，形成输出结果。输出的信息称为输出向量。隐藏层（Hidden layer），简称“隐层”，是输入层和输出层之间众多神经元和链接组成的各个层面。隐层可以有多层，习惯上会用一层。隐层的节点（神经元）数目不定，但数目越多神经网络的非线性越显著，从而神经网络的强健性（robustness）（控制系统在一定结构、大小等的参数摄动下，维持某些性能的特性。）更显著。习惯上会选输入节点1.2至1.5倍的节点。

在本次实验中，由于有28\*28=784个输入，则我们有784个输入单元。由于电脑计算能力有限，我采取的隐含层层数只有50个，输出层是0-9这10个数字的置信度共计10个。下图3所示的就是我本次实验采用的神经网络示意图。在本次试验中，每个节点的输出函数并不是线性关系，而是sigmod函数，即，表示改与该节点相关的参数，即要训练的值。

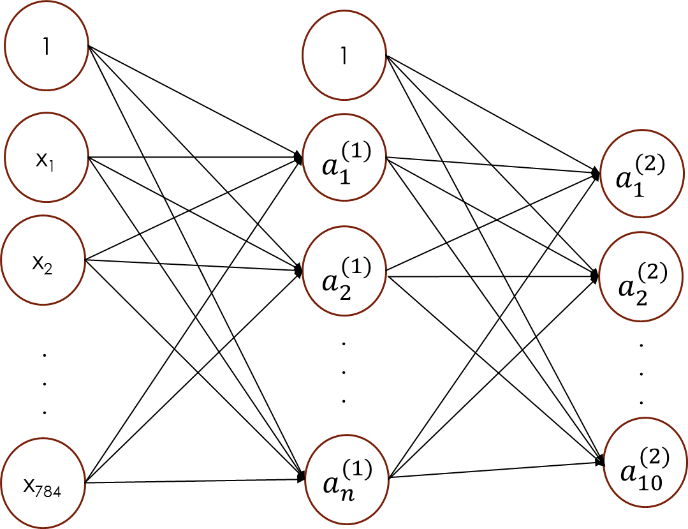


图3　本次实验所采用的神经网络示意图

1. 反向传播算法

反向传播算法（英：Backpropagation algorithm，简称：BP算法）是一种监督学习算法，常被用来训练多层感知机。 于1974年，Paul Werbos首次给出了如何训练一般网络的学习算法，而人工神经网络只是其中的特例。不巧的，在当时整个人工神经网络社群中却无人知晓Paul所提出的学习算法。直到80年代中期，BP算法才重新被David Rumelhart、Geoffrey Hinton及Ronald Williams、David Parker和Yann LeCun独立发现，并获得了广泛的注意，引起了人工神经网络领域研究的第二次热潮。

BP算法是Delta规则的推广，要求每个人工神经元（节点）所使用的激励函数必须是可微的。BP算法特别适合用来训练前向神经网络。

反向传播算法(BP算法)主要由两个环节(激励传播、权重更新)反复循环迭代，直到网络的对输入的响应达到预定的目标范围为止。其中，激励传播环节包含两步，第一步是将训练输入送入网络以获得激励响应，第二步时将激励响应同训练输入目标求差，从而得到隐含层和输入层的误差。权重更新时，首先将输入激励和响应误差相乘，从而获得权重的梯度，然后将这个梯度乘以上一个比例并取反后加到权重上来。

具体来说，假设输入层输入一个训练样例，将其放到神经网络中进行计算（即前向传播），在输出层得到输出，将其与真实的相减，得到结果为。将这个结果进行反向传播，得到隐含层和输出层的误差，为当前隐含层和输入层的参数，为sigmod激励的导数。同理也可以计算出输入层和隐含层之间的误差。这样就可以求出梯度下降算法的每一次更新值为，就能够用这个求出每次更新的值。

在更新值之前，需要计算出当前神经网络下的花费，用来衡量梯度下降算法迭代时的效率，并且保证每一次迭代都是向花费减少的方向迭代。

计算花费

在更新之前，需要首先随机化一下神经网络的权值，而且这个随机必须是正态分布的，并且在之间取值。在本例中，取0.12。

1. 关键代码实现

本次实验使用的是GUN Octave，一个类MATLAB的开源软件，能够运行在Linux操作系统中。

1. 初始化

初始化使用的是Rand函数，该函数能够在[0,1]之间按照正态分布的方式取出指定的个数，具体代码如下：

W = rand(L\_out, 1 + L\_in) \* 2 \* epsilon\_init - epsilon\_init;

L\_out和L\_in分别为需要生成的矩阵的行数和列数，L\_in + 1 是因为需要将偏置单元加上。

1. 计算花费和值

下面是计算花费J的代码，for循环中使用前向传播得到每一次迭代的误差值，然后进行加和计算，然后最后使用sum汇总。本例中，lamda参数的值为1。

for i=1:m

input = [ones(1,1) X(i,:)];

hidden = sigmoid(input \* Theta1');

hidden = [1 hidden];

output = sigmoid(hidden \*Theta2');

temp(i) = sum(Y(:,i)' .\* log(output) + (1-Y(:,i)') .\* log(1-output));

end;

J = -sum(temp) / m;

t = sum(sum(Theta1.^2)) + sum(sum(Theta2.^2)) - sum(Theta1(:,1).^2) - sum(Theta2(:,1).^2);

J = J + (lambda \*t )/(2\*m);

计算值的代码就比较复杂：

for i=1:m

input = [ones(1,1) X(i,:)];

hidden = sigmoid(input \* Theta1');

hidden = [1 hidden];

output = sigmoid(hidden \*Theta2');

delter\_3 = (output' - Y(:,i));

delter\_2 = [Theta2' \* delter\_3] .\* sigmoidGradient([1 input \* Theta1'])';

delter\_2 = delter\_2(2:end);

Theta1\_grad = Theta1\_grad + delter\_2 \* input;

Theta2\_grad = Theta2\_grad + delter\_3 \* hidden;

end;

Theta1\_grad = Theta1\_grad ./ m;

Theta2\_grad = Theta2\_grad ./ m;

这段代码中，同样是采用前向传播计算出误差，然后使用后向传播计算误差，在计算时要注意去除偏执单元引起的误差。

1. 梯度下降迭代

在Octave中，可以直接使用fmincg函数就能进行梯度下降算法的迭代。使用fmincg时，首先要指明了计算花费和的函数，然后指明初始调用的值，还有选项，就能调用梯度下降算法直接计算，不用再去写梯度下降算法的方法。

costFunction = @(p) nnCostFunction(p, ...

input\_layer\_size, ...

hidden\_layer\_size, ...

num\_labels, X\_tmp, y\_tmp, lambda);

options = optimset('MaxIter', 50);

[nn\_params, cost] = fmincg(costFunction, initial\_nn\_params, options);

1. 计算结果

再使用6万个训练样本进行训练后，再用1万个训练样本进行对比，输出计算的精确度。得到的精确度如下图4所示，可以看到计算精度大概能到90.74%。这样的一个精度主要是由于隐含层数目比较少的原因。由于我机器性能的原因，不能用更多的隐含层进行计算，如果能用更多的隐含层进行计算，就能够得到更多的值。

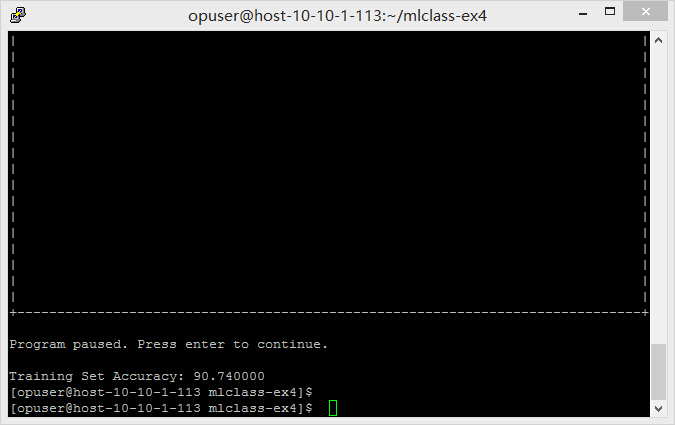


图4　计算结果

上图2的图片经过神经网络之后，能够得到如下图5的结果：

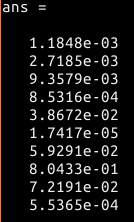


图5　神经网络输出结果

可以看到，经过神经网络计算，能够区分出来这个图片为数字8，而且置信度能达到0.8，能够很好的和其他数字进行区分。