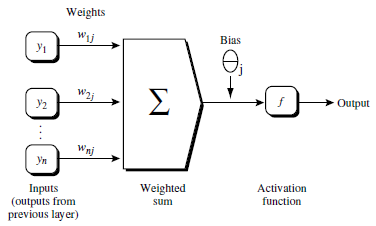
[**神经网络NN算法(应用篇)**](http://www.hongweipeng.com/index.php/archives/1420/)

起步

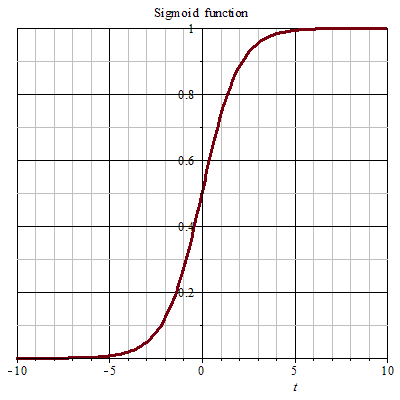
在神经网络的理论篇中，神经元在对权重进行求和后，需要进行一个非线性转化，即作为参数传入激活函数去。这个激活函数是一个 S 型函数( Sigmoid )。



S 函数：

* 双曲函数（ tanh ）；
* 逻辑函数（logistic function ）

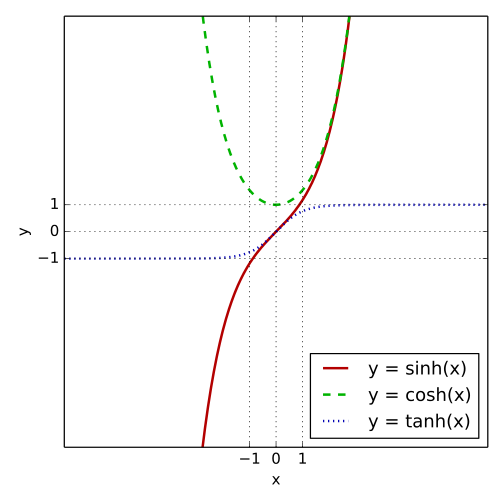
Sigmoid 函数

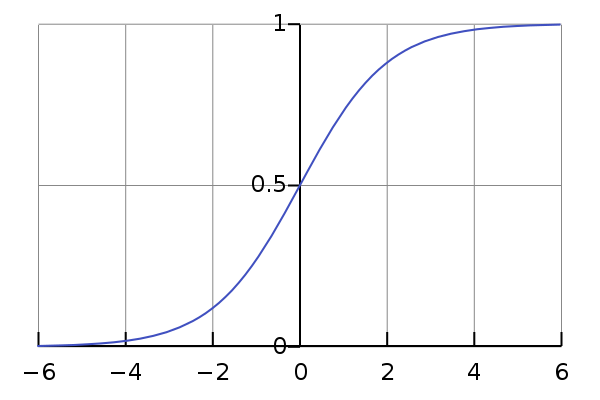


S 曲线函数可以将一个数值转为值域在 0 到 1 之间，广义上S 函数是满足y值在某个值域范围，渐变的一个曲线就可以了。通常情况下使用什么样的 Sigmoid 函数呢？有两种。

**双曲函数（ tanh ）** 双曲函数是一类与常见的三角函数（也叫圆函数）类似的函数。详情看维基：

http://latex.codecogs.com/png.latex?tanhx%20=%20\frac%7bsinhx%7d%7bcoshx%7d

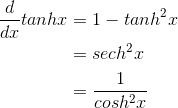


**逻辑函数（logistic function ）** 逻辑函数或逻辑曲线是一种常见的S函数

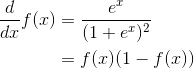
一个简单的逻辑函数可用公式表示:

http://latex.codecogs.com/png.latex?P(t)%20=%20\frac1%7b1+e%5e%7b-t%7d%7d

从百科中可以得到它们的求导，双曲函数的导数：



逻辑函数的导数：



实现神经网络

知道神经网络的工作原理，现在可以着手实现一个神经网络了。

基础函数

根据上面的sigmoid 函数以及其求导得到对应的python代码：

# coding: utf-8

import numpy as np

def tanh(x):

return np.tanh(x)

def tanh\_deriv(x):

"""

tanh的导数

"""

return 1.0 - np.tanh(x) \* np.tanh(x)

def logistic(x):

return 1.0 / (1 + np.exp(-x))

def logistic\_deriv(x):

"""

逻辑函数的导数

"""

fx = logistic(x)

return fx \* (1 - fx)

设计神经网络的类结构

类的结构大致如下：

class NeuralNetwork(object):

def \_\_init\_\_(self, layers, activation='tanh'):

pass

def fit(self, X, Y, learning\_rate=0.2, epochs=10000):

pass

def predict(self, x);

pass

构造函数

在构造函数中，需要确定神经网络的层数，每层的个数，从而确定单元间的权重规格和单元的偏向。

def \_\_init\_\_(self, layers, activation='logistic'):

"""

:param layers: 层数，如[4, 3, 2] 表示两层len(list)-1,(因为第一层是输入层，，有4个单元)，

第一层有3个单元，第二层有2个单元

:param activation:

"""

if activation == 'tanh':

self.activation = tanh

self.activation\_deriv = tanh\_deriv

elif activation == 'logistic':

self.activation = logistic

self.activation\_deriv = logistic\_deriv

# 初始化随机权重

self.weights = []

for i in range(len(layers) - 1):

tmp = (np.random.random([layers[i], layers[i + 1]]) \* 2 - 1) \* 0.25

self.weights.append(tmp)

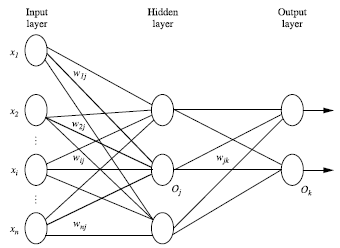
# 偏向随机初始化

self.bias = []

for i in range(1, len(layers)):

self.bias.append((np.random.random(layers[i]) \* 2 - 1) \* 0.25)

其中，layers 参数表示神经网络层数以及各个层单元的数量，例如下图这个模型：



这个模型就对应了 layers = [4, 3, 2] 。这是一个 2 层，即 len(layers) - 1 层的神经网络。输入层 4 个单元，其他依次是 3 ，2 。权重是表示层之间单元与单元之间的连接。因此权重也有 2 层。第一层是一个 4 x 3 的矩阵，即 layers[0] x layers[1]。同理，偏向也有 2 层，第一层个数 3 ，即leyers[1] 。

利用 np.random.random([m, n]) 来创建一个 m x n 的矩阵。在这个神经网络的类中，初始化都随机 -0.25 到 0.25 之间的数。

训练函数

在神经网络的训练中，需要先设定一个训练的终止条件，在理论篇介绍了3种停止条件，这边使用的是 **达到预设一定的循环次数** 。每次训练是从样本中随机挑选一个实例进行训练，将这个实例的预测结果和真实结果进行对比，再进行反向传播得到各层的误差，然后再更新权重和偏向：

def fit(self, X, y, learning\_rate=0.2, epochs=10000):

X = np.atleast\_2d(X)

y = np.array(y)

# 随即梯度

for k in range(epochs):

i = np.random.randint(X.shape[0])

a = [X[i]] # 随即取某一条实例

for j in range(len(self.weights)):

a.append(self.activation(np.dot(a[j], self.weights[j]) + self.bias[j] ))

errors = y[i] - a[-1]

deltas = [errors \* self.activation\_deriv(a[-1]) ,] # 输出层的误差

# 反向传播，对于隐藏层的误差

for j in range(len(a) - 2, 0, -1):

tmp = np.dot(deltas[-1], self.weights[j].T) \* self.activation\_deriv(a[j])

deltas.append(tmp)

deltas.reverse()

# 更新权重

for j in range(len(self.weights)):

layer = np.atleast\_2d(a[j])

delta = np.atleast\_2d(deltas[j])

self.weights[j] += learning\_rate \* np.dot(layer.T, delta)

# 更新偏向

for j in range(len(self.bias)):

self.bias[j] += learning\_rate \* deltas[j]

参数 learning\_rate 表示学习率， epochs 表示设定的循环次数。

预测

预测就是将测试实例从输入层传入，通过正向传播，最后返回输出层的值即可：

def predict(self, row):

a = np.array(row) # 确保是 ndarray 对象

for i in range(len(self.weights)):

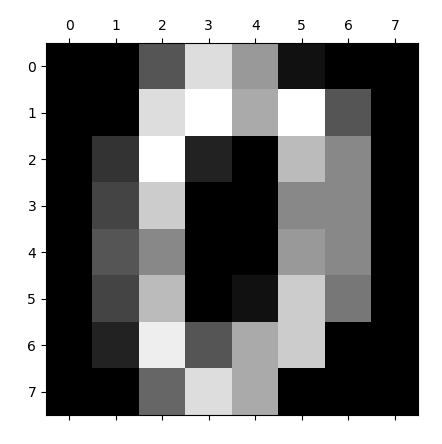
a = self.activation(np.dot(a, self.weights[i]) + self.bias[i])

return a

这个神经网络只用了 50 多行的代码了。来应用看看实际效果吧。

手写数字识别

手写数字数据集来自 sklearn ，其中由1797个图像组成，其中每个图像是表示手写数字的 8x8 像素图像：



可以推出，这个神经网络的输入层将有 64 个输入单元，分类结果是 0~9 ，因此输出层有10个单元，构造为：

nn = NeuralNetwork(layers=[64, 100, 10])

载入数据集

from sklearn import datasets

digits = datasets.load\_digits()

X = digits.data

y = digits.target

拆分成训练集和数据集，分类结果离散化：

from sklearn.model*\_selection import train\_*test\_split

from sklearn.preprocessing import LabelBinarizer

![20171219170850.png][10]

# 拆分为训练集和测试集

X*\_train, X\_*test, y*\_train, y\_*test = train*\_test\_*split(X, y)

# 分类结果离散化

labels*\_train = LabelBinarizer().fit\_*transform(y\_train)

labels*\_test = LabelBinarizer().fit\_*transform(y\_test)

训练

取训练集进行训练：

nn.fit(X\_train, labels\_train)

测试模型

# 收集测试结果

predictions = []

for i in range(X\_test.shape[0]):

o = nn.predict(X\_test[i] )

predictions.append(np.argmax(o))

# 打印对比结果

from sklearn.metrics import confusion\_matrix, classification\_report

print (confusion\_matrix(y\_test, predictions) )

print (classification\_report(y\_test, predictions))

利用测试集对模型进行测试，得到 confusion\_matrix 的打印结果：

[[51 0 0 0 1 0 0 0 0 0]

[ 0 44 0 1 0 0 0 0 0 1]

[ 0 0 34 0 0 0 0 0 0 0]

[ 0 0 0 41 0 0 0 0 0 0]

[ 0 0 0 0 39 0 0 0 0 0]

[ 0 0 0 0 0 50 0 0 0 0]

[ 0 1 0 0 0 0 42 0 1 0]

[ 0 0 0 0 0 0 0 59 0 0]

[ 0 4 1 3 0 0 0 0 31 0]

[ 1 0 0 0 0 1 0 0 0 44]]

行列中，表示预测值与真实值的情况，比方预测值为0，真实值也为0，那么就在 [0][0] 计数 1。因此这个对角线计数越大表示预测越准确。

另一个报告 classification\_report :

precision recall f1-score support

0 0.98 0.98 0.98 52

1 0.90 0.96 0.93 46

2 0.97 1.00 0.99 34

3 0.91 1.00 0.95 41

4 0.97 1.00 0.99 39

5 0.98 1.00 0.99 50

6 1.00 0.95 0.98 44

7 1.00 1.00 1.00 59

8 0.97 0.79 0.87 39

9 0.98 0.96 0.97 46

avg / total 0.97 0.97 0.97 450

正确率在 97% ，相当不错的。

附上本次实验代码：<https://github.com/hongweipeng/learn_ai_example/blob/master/nn_example.py>