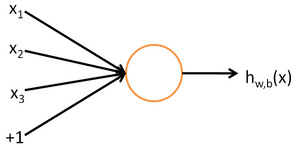
# **[反向传播神经网络](http://www.hankcs.com/ml/back-propagation-neural-network.html)**

**单个神经元**

神经网络是多个“神经元”（感知机）的带权级联，神经网络算法可以提供非线性的复杂模型，它有两个参数：权值矩阵{Wl}和偏置向量{bl}，不同于感知机的单一向量形式，{Wl}是复数个矩阵，{bl}是复数个向量，其中的元素分别属于单个层，而每个层的组成单元，就是神经元。

**神经元**

神经网络是由多个“神经元”（感知机）组成的，每个神经元图示如下：

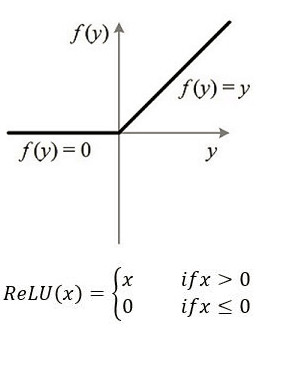


其实就是一个单层感知机，其输入是由x1，x2，x3和+1组成的向量，其输出为，即输出等于输入与系统函数的卷积。对于矩阵来说，卷积就是相乘。其中f是一个激活函数（输出函数），主要目的是让输出量在一定范围之内，方便计算。

**单极性Sigmoid激活函数**

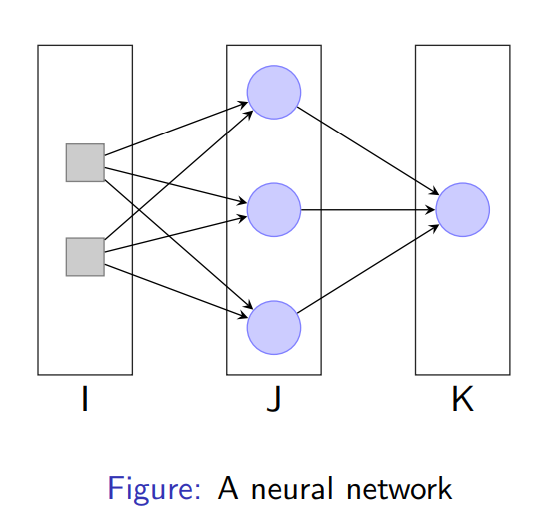


## **ReLU激活函数**



**神经网络模型**

神经网络就是多个神经元的级联，上一级神经元的输出是下一级神经元的输入，而且信号在两级的两个神经元之间传播的时候需要乘上这两个神经元对应的权值。例如，下图就是一个简单的神经网络：



### 设给定训练集M，对于其中任意一个样本m，输入是，输出是，期望是，训练次数为N。

### **前向传播**



### ****输出层权值调整****

对于每次训练，模型输出的输出误差定义为：



其实就是所有实例的方差的一半，训练的目标就是最小化该方差，每次训练更新一次、。根据梯度下降原理求E对的梯度：



编程代码：



更新输出层权重：



再求E对的梯度：



编程代码



更新隐藏层权重：



2 relu函数方式

前向传递



损失函数



梯度求导





3 推导公式及代数含义

# N is batch size; D\_in is input dimension;

# H is hidden dimension; D\_out is output dimension.

N, D\_in, H, D\_out

N:样本个数

D\_in:样本的维度

H:隐藏层维度

D\_out:输出结果的维度











sigmod函数BP代码

import numpy as np

N, D\_in, H, D\_out = 5, 5, 10, 5

# x = np.array([ [0,0],[0,1],[1,0],[1,1] ])

# y = np.array([[0,0,0,1]]).T

# x = np.random.randn(N, D\_in)

# y = np.random.randn(N, D\_out)

x = np.array([

[1,0,0,0,0],

[0,1,0,0,0],

[0,0,1,0,0],

[0,0,0,1,0],

[0,0,0,0,1],

])

y = x.copy()

# syn0 = np.random.random((2,4))

# syn1 = np.random.random((4,1))

syn0 = np.random.randn(D\_in, H)

syn1 = np.random.randn(H, D\_out)

lean\_rate=5

for j in range(1000):

l1 = 1/(1+np.exp(-(np.dot(x,syn0))))

l2 = 1/(1+np.exp(-(np.dot(l1,syn1))))

print(((l2-y)\*\*2).sum())

l2\_delta = (l2-y)\*(l2\*(1-l2))

l1\_delta = l2\_delta.dot(syn1.T) \* (l1 \* (1-l1))

syn1 -= lean\_rate\*l1.T.dot(l2\_delta)

syn0 -= lean\_rate\*x.T.dot(l1\_delta)

l1 = 1/(1+np.exp(-(np.dot(x,syn0))))

l2 = 1/(1+np.exp(-(np.dot(l1,syn1))))

print(l2)

ReLU函数的BP代码

# # -\*- coding: utf-8 -\*-

import numpy as np

def sigmoid(x, Derivative=False):

if not Derivative:

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

else:

out = sigmoid(x)

return out \* (1 - out)

# N is batch size; D\_in is input dimension;

# H is hidden dimension; D\_out is output dimension.

N, D\_in, H, D\_out = 4, 2, 4, 1

# Create random input and output data

# x = np.random.randn(N, D\_in)

# y = np.random.randn(N, D\_out)

x = np.array([[1,0],[0,1],[0,0],[1,1]])

y = np.array([[0],[0],[0],[1]])

# Randomly initialize weights

w1 = np.random.randn(D\_in, H)

w2 = np.random.randn(H, D\_out)

learning\_rate = 0.5

for t in range(1000):

# Forward pass: compute predicted y

h\_relu = sigmoid(x.dot(w1))

y\_pred = sigmoid(h\_relu.dot(w2))

# Compute and print loss

loss = np.square(y\_pred - y).sum()

print(t, loss)

# Backprop to compute gradients of w1 and w2 with respect to loss

grad\_y\_pred = y\_pred.dot((1-y\_pred).T).dot(y\_pred - y)

grad\_w2 = h\_relu.T.dot(grad\_y\_pred)

grad\_h\_relu = h\_relu.dot((1-h\_relu).T).dot(grad\_y\_pred).dot(w2.T)

grad\_w1 = x.T.dot(grad\_h\_relu)

# Update weights

w1 -= learning\_rate \* grad\_w1

w2 -= learning\_rate \* grad\_w2

h = x.dot(w1)

h\_relu = np.maximum(h, 0)

y\_pred = h\_relu.dot(w2)

print(y\_pred)