第四讲：多层感知机

南京大学人工智能学院

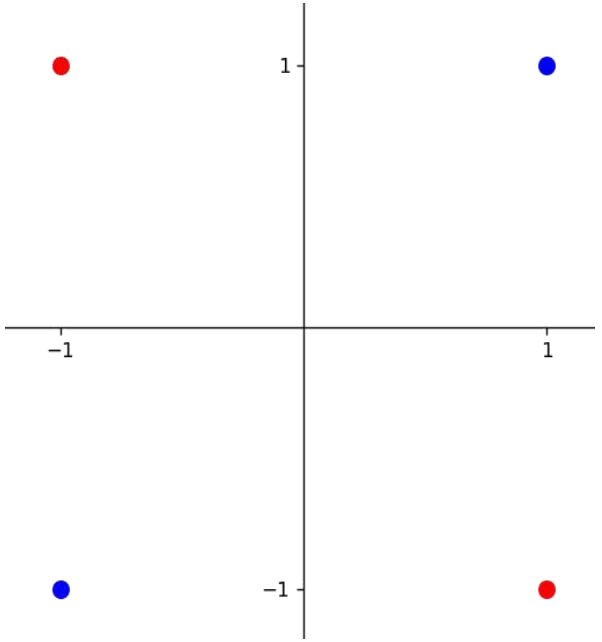
申富饶

目录

CONTENTS

01 单层感知机局限

单层感知机的局限

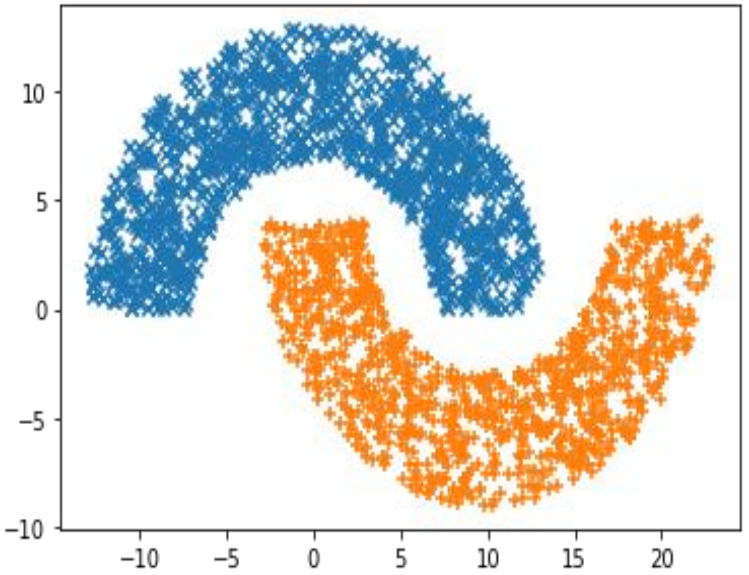


单层感知机：无法解决线性不可分问题，比如异或问题。

对于非线性问题，需要增加神经元，以拟合输入-输出关系。

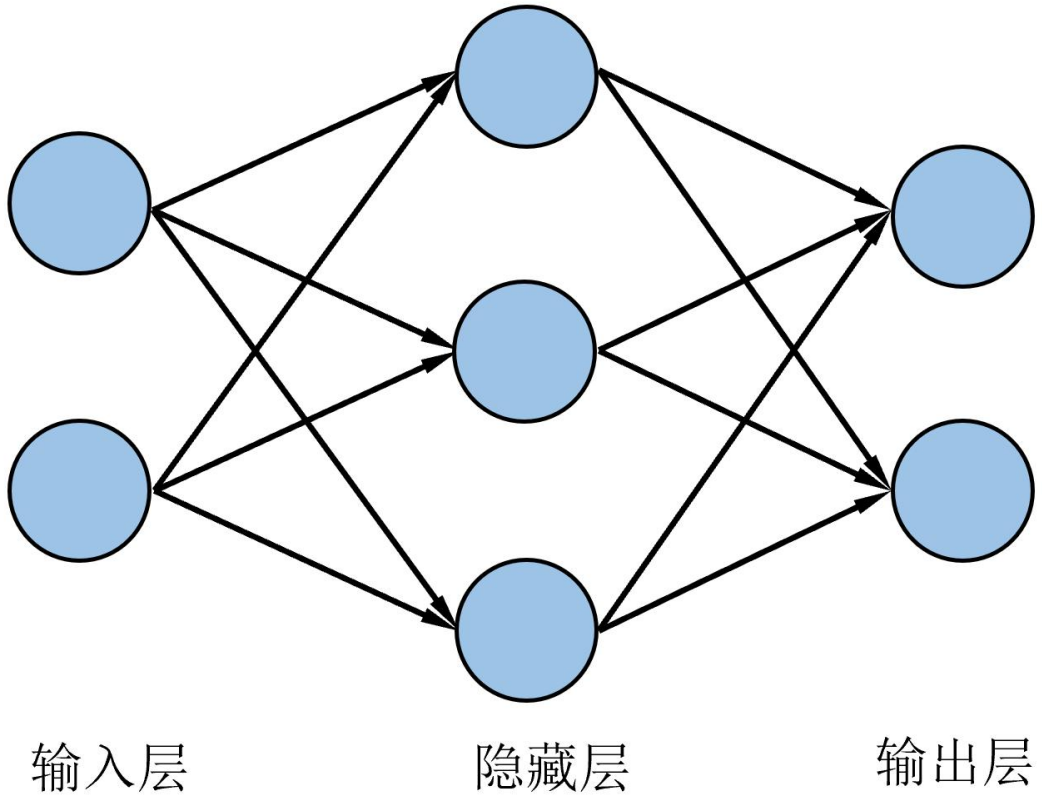
↓

增加网络层数不仅可以解决异或问题，而且具有很好的非线性分类效果。



单隐藏层神经网络

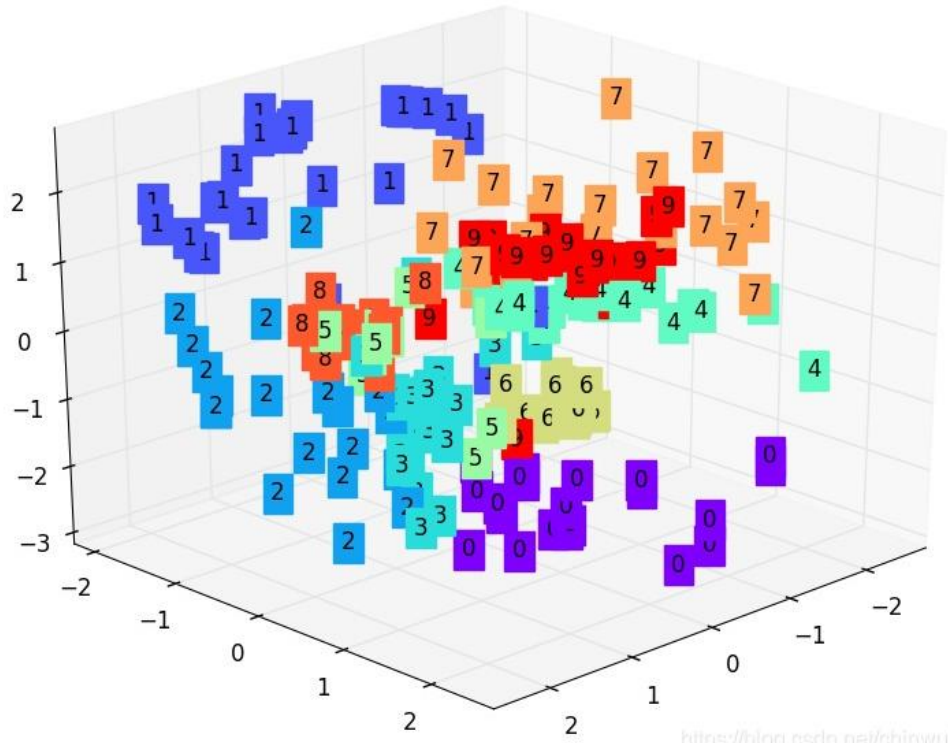
- 在单层感知机的基础上，增加一个中间层，称为**隐藏层**。



隐藏层

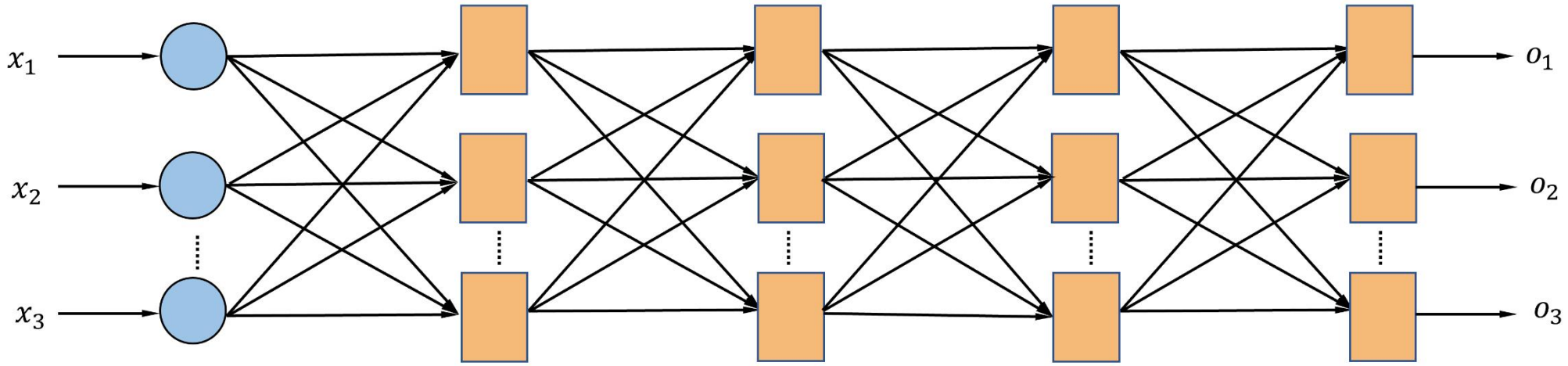
- 隐藏层的意义就是把输入数据的特征抽象到另一个维度空间，来展现其更抽象化的特征，这些特征能更好的进行线性划分。
- 以手写数字分类为例：

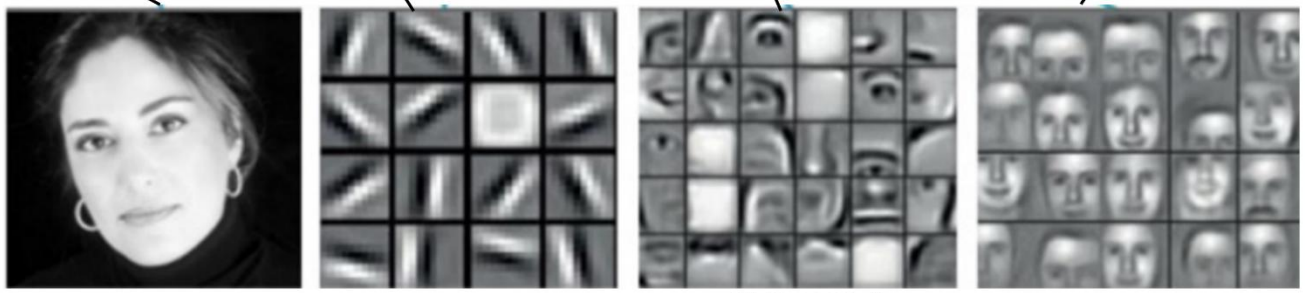
输入图片经过隐藏层加工，变成另一种特征代表 。如图所示，假如隐藏层中有3个神经元就可以输出3个特征，借助这3个特征可以将数字“1”与其他数字区分开来。



隐藏层

- 同理，如果有多个隐藏层就能对输入特征进行多层次的抽象。

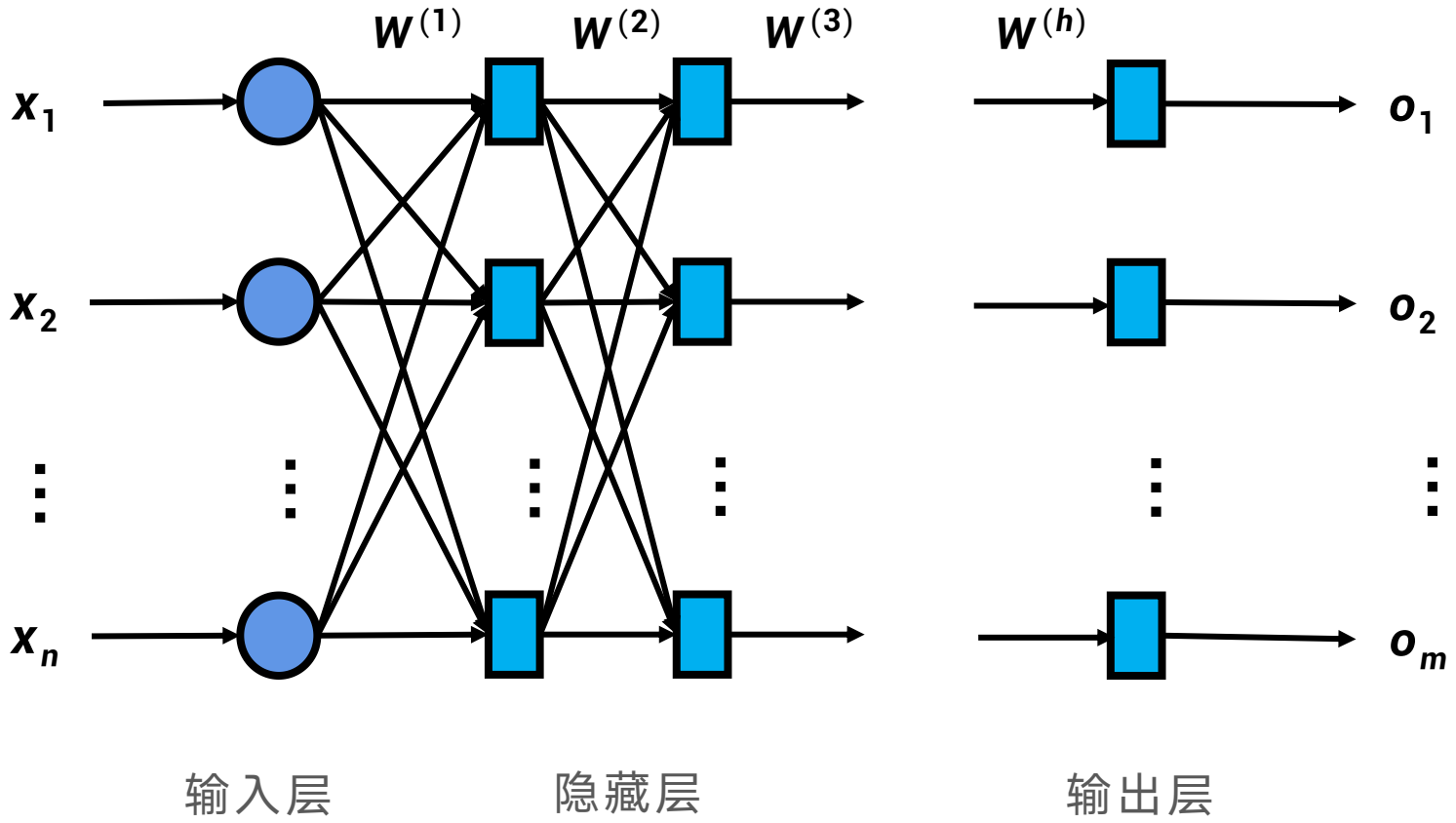




02 多层感知机基本概念

多层感知机

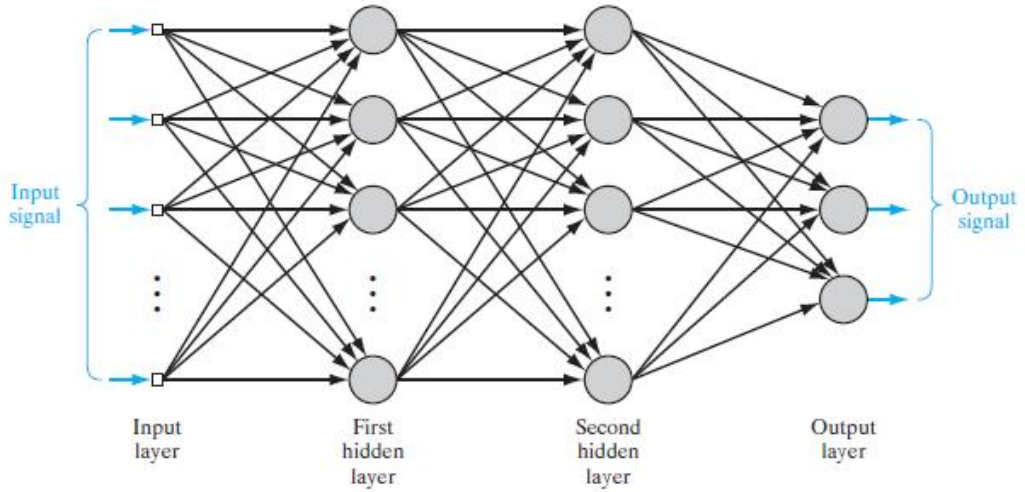
多层感知机（Multilayer Perceptron，MLP）是至少有一个隐藏层的神经网络，具有以任意精度逼近神经元输入层和输出层之间的任何非线性关系的能力。



多层感知机

模型特征

- 层级结构，各神经元分别属于不同的层，层内无连接，相邻两层之间的神经元全部两两连接。
- 多层感知机是单层感知机的加深结构，更深层的网络所表达的数学形式更复杂，能够拟合更复杂的分布。
- 网络中每个神经元模型包含一个可微的非线性激活函数。



多层感知机的数学表达

神经元的输入：

向量形式：

$$NET = WX$$

矩阵形式：

$$\begin{bmatrix} net_1 \\ \dots \\ net_n \end{bmatrix} = \begin{bmatrix} w_{11} & \dots & w_{1n} \\ \dots & \dots & \dots \\ w_{n1} & \dots & w_{nn} \end{bmatrix} \begin{bmatrix} x_1 \\ \dots \\ x_n \end{bmatrix}$$

分量形式：

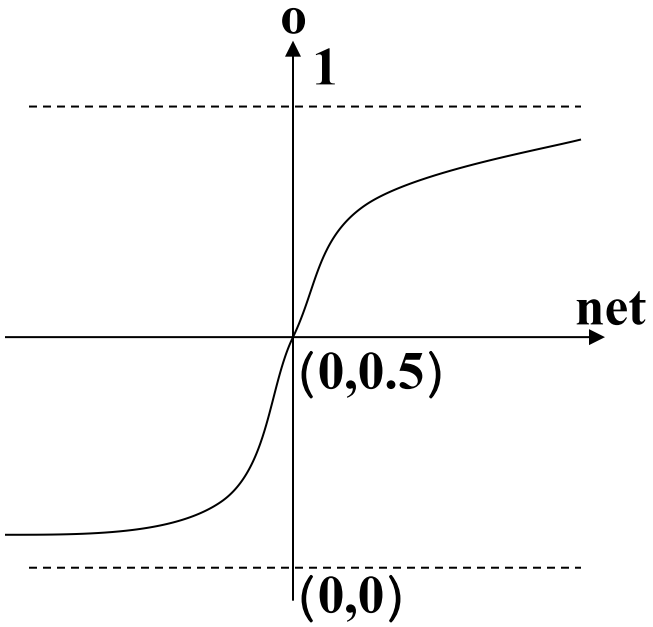
$$net_i = x_1 w_{1i} + x_2 w_{2i} + \dots + x_n w_{ni}$$

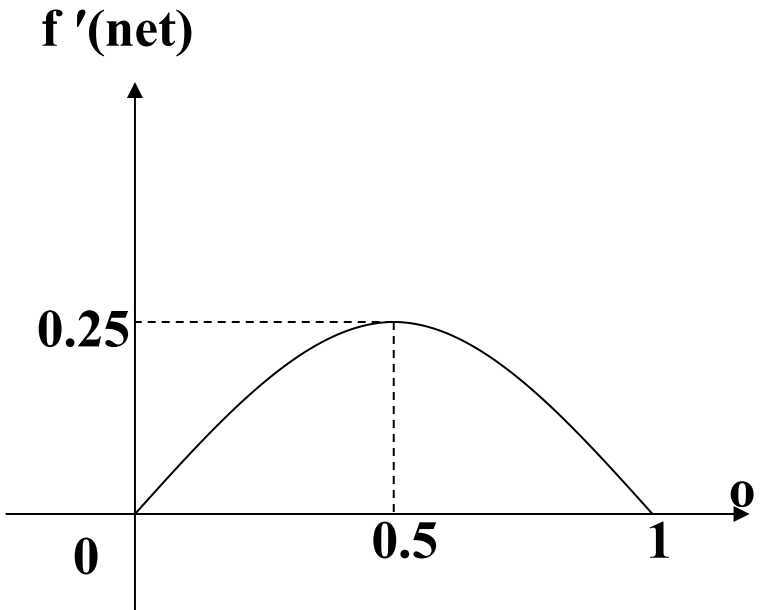
多层感知机的数学表达

神经元的输出：

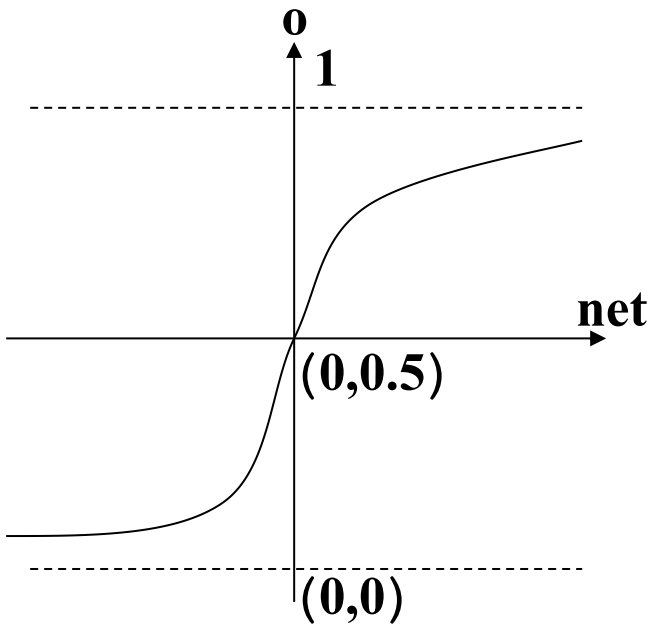
$$o = f(net) = \frac{1}{1+e^{-net}}$$

$$f'(net) = -\frac{1}{(1+e^{-net})^2}(-e^{-net}) = o - o^2 = o(1-o)$$

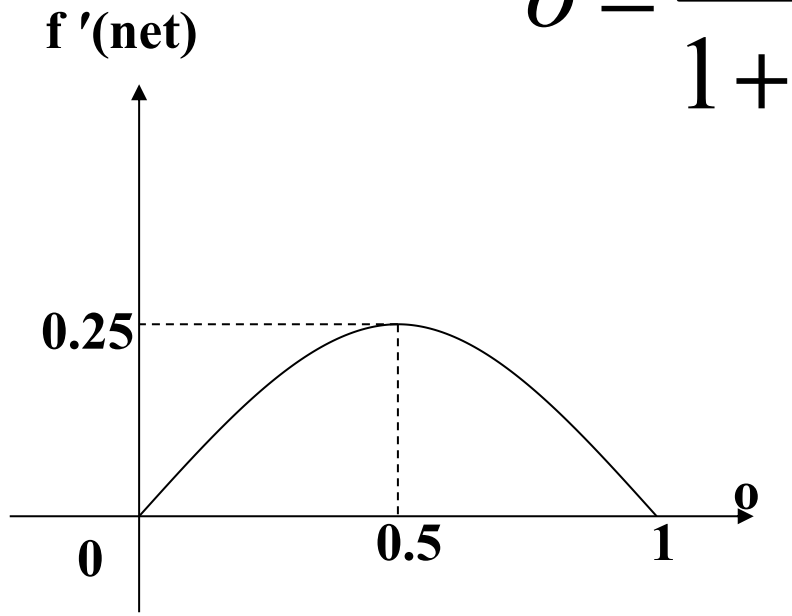




多层感知机的数学表达

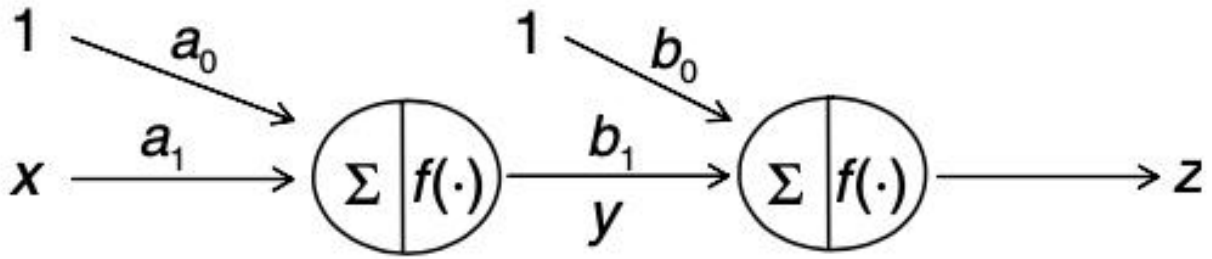


$$o = \frac{1}{1+e^{-net}}$$



- 应该将net的值尽量控制在收敛比较快的范围内
- 可以用其它的函数作为激活函数，只要该函数连续可导

多层感知机的运行



以单层单个隐藏神经元为例：

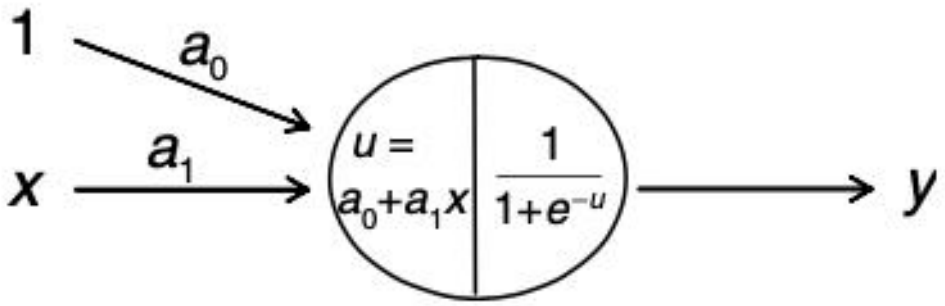
1. 加权输入

$$u = a_0 + a_1 x$$

权值a_0, a_1确定了输入x和u线性关系

权值a_0可以认为是除了x外的所有不显式包含在模型里的输入影响

多层感知机的运行



2.非线性处理

加权的输入传递给非线性函数 f

$$u = a_0 + a_1 x$$

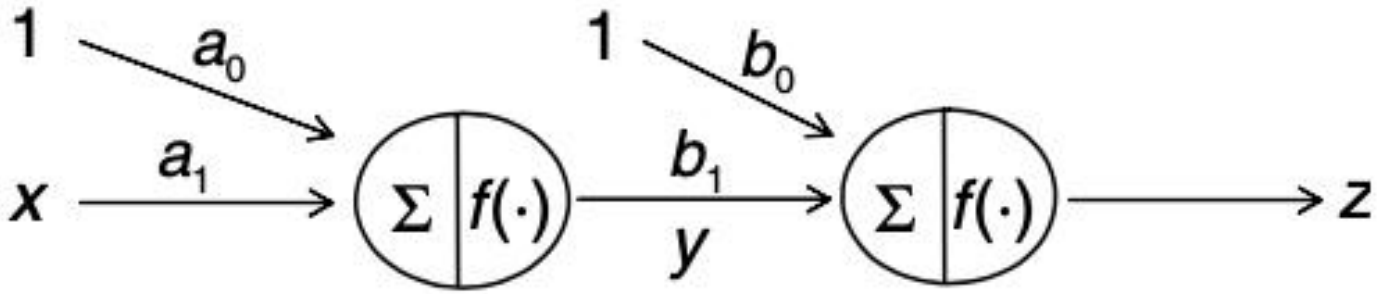
$$y = f(u)$$

●例： $f(u) = \frac{1}{1+e^{-u}}$ 则 $y = \frac{1}{1+e^{-(a_0+a_1 x)}}$

通过调整权重，输入和输出具有不同的非线性关系

隐藏神经元的非线性处理，使得输出相对初始输入是非线性的

多层感知机的运行



3.网络输出

$$u = a_0 + a_1 x$$

$$y = f(u)$$

$$v = b_0 + b_1 y$$

$$z = f(v)$$

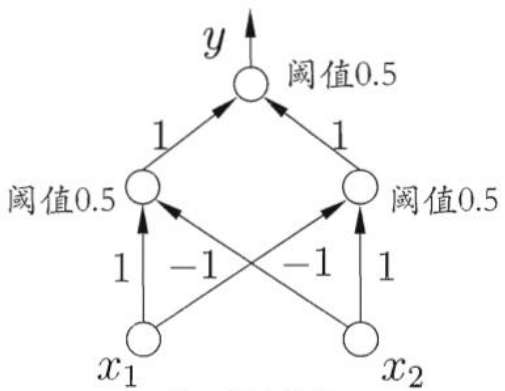
单一神经元的拟合能力是有限的，

增加隐藏神经元，每个神经元分别改变权值，共同拟合期望函数。

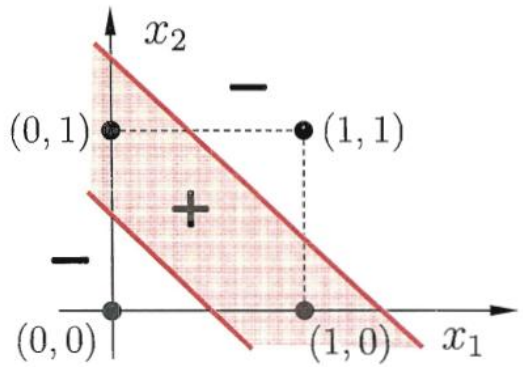
多层感知机：异或问题

- (0,0) $f_1(0\times 1+0\times -1-0.5)=0$ $f_2(0\times -1+0\times 1-0.5)=0$ $y=f_3(0\times 1+0\times 1-0.5)=0$
- (1,0) $f_1(1\times 1+0\times -1-0.5)=1$ $f_2(1\times -1+0\times 1-0.5)=0$ $y=f_3(1\times 1+0\times 1-0.5)=1$
- (0,1) $f_1(0\times 1+1\times -1-0.5)=0$ $f_2(0\times -1+1\times 1-0.5)=1$ $y=f_3(0\times 1+1\times 1-0.5)=1$
- (1,1) $f_1(1\times 1+1\times -1-0.5)=0$ $f_2(1\times -1+1\times 1-0.5)=0$ $y=f_3(0\times 1+0\times 1-0.5)=0$

其中 $f_n(u)=\begin{cases}0 & u<0\\ 1 & u\geq 0\end{cases}$



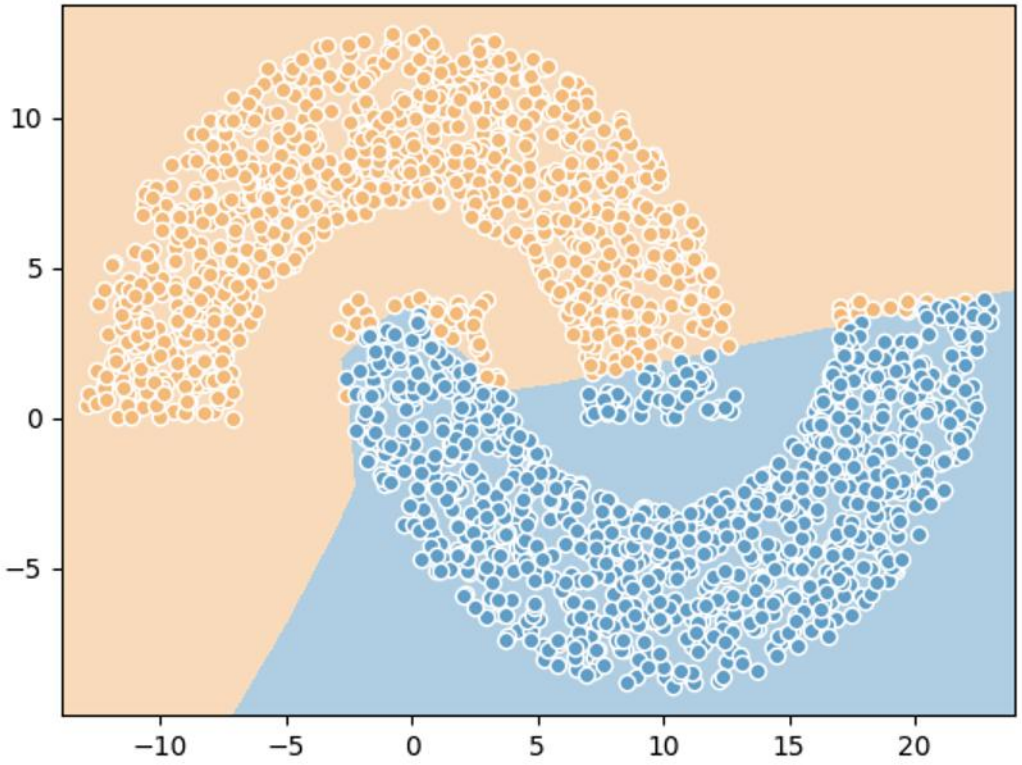
(a) 网络结构

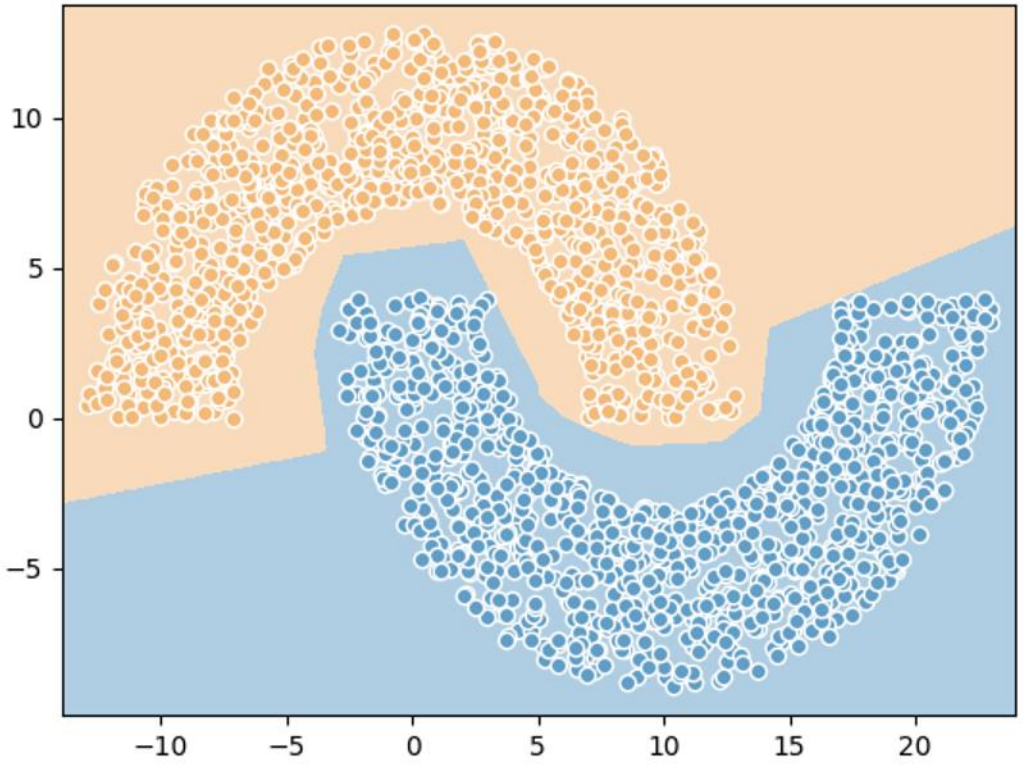


(b) 分类区域

多层感知机：双月模型

- 当 r=10，w=6 且 **d=-4** 时：**单层感知机**（左图）VS**多层感知机**（右图）





03 多层感知机的学习

3.1 基本原理

基本原理

为什么要学习：

多层感知机的神经元个数很多，不适合手工设计各个神经元的参数，需要神经元“自动”学习参数。

基本原理

在多层感知机中，最初的权值设置为任意值。网络每处理一个输入，网络输出都与期望输出相比较，差值称为“误差”。

误差表示：

- 平均平方误差(MSE)：$Err = \frac{1}{N}\sum_{i=1}^{N}(y_i - o_i)^2$
- 平均绝对误差(MAD)：$Err = \frac{1}{N}\sum_{i=1}^{N}|y_i - o_i|$

y_i和o_i表示第i个神经元的实际输出和期望输出，N是神经元总数。

基本原理

训练算法的主要步骤包括：

1） 从样本集合中取一个样本（X，Y）；

2） 计算出网络的实际输出O；

3） 计算误差$D = Y - O$；

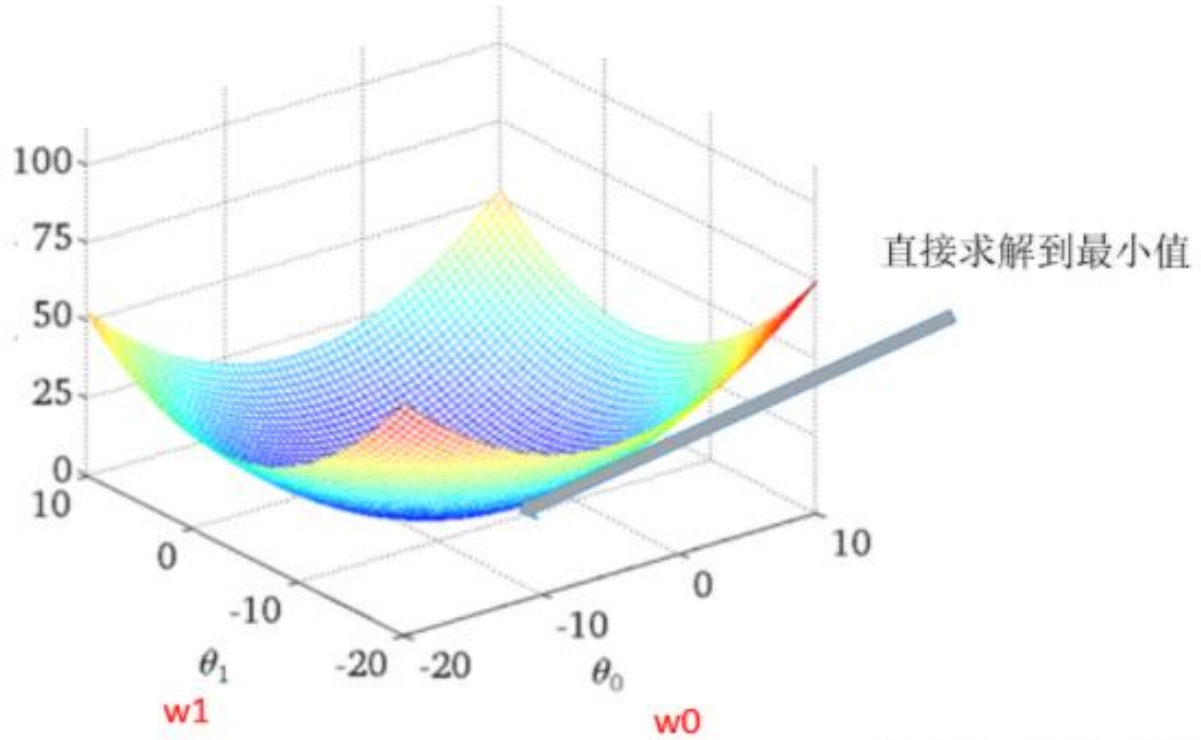
4） 根据D调整权值矩阵W和偏置矩阵B；

5） 对每个样本重复上述过程，直到对整个样本集来说，所有样本预测正确或误差不超过规定范围。

基本原理

在神经网络中，通过调整权重组合，得到产生最小的误差的网络。

通过迭代学习，得到网络中各权重值的集合。



3.2 梯度下降法

梯度下降法

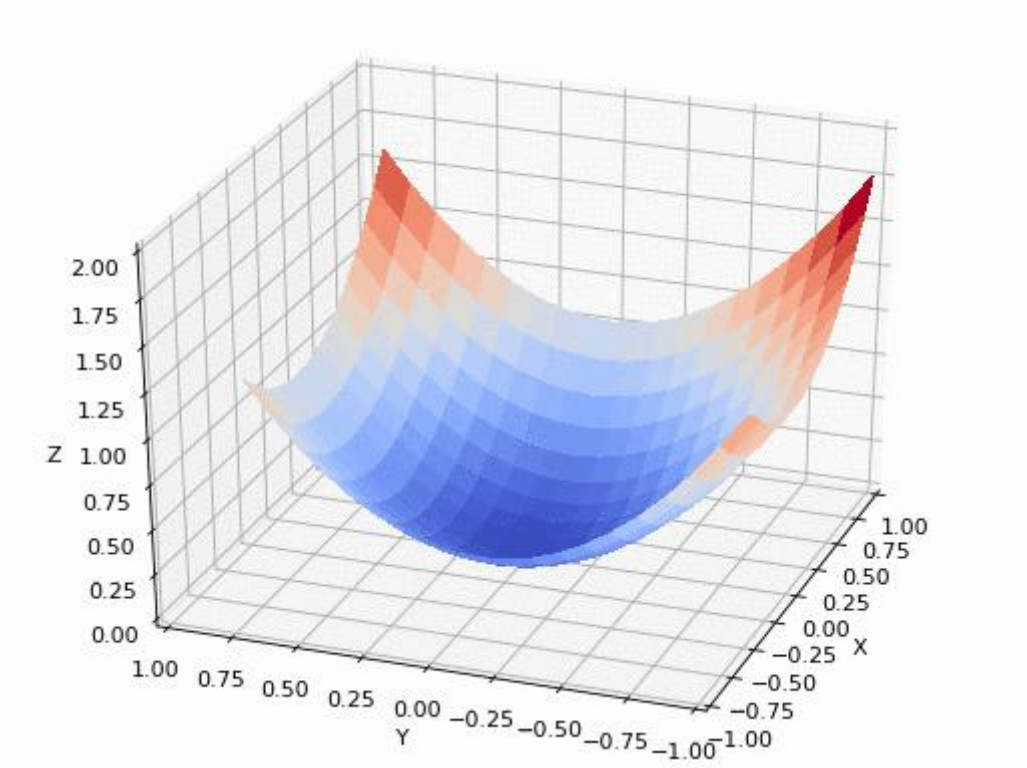
求全局最小的常用方法：梯度下降法

梯度下降的基本过程和下山的场景类似。

如何最快地下山？

--从最陡峭的地方向下

函数的梯度确定了这个“最陡峭的方向”



梯度下降法

梯度的方向就是函数之变化最快的方向。

看待微分的意义，可以有不同的角度，最常用的两种是：

- 函数图像中，某点的切线的斜率
- 函数的变化率

梯度下降法

几个微分的例子：

- 单变量的微分，函数只有一个变量时

$$\frac{d(x^2)}{dx} = 2x$$

$$\frac{d(-2y^5)}{dy} = -10y^4$$

$$\frac{d(5-\theta)^2}{d\theta} = -2(5-\theta)$$

- 多变量的微分，当函数有多个变量的时候，即分别对每个变量进行求微分输出：

$$\frac{\partial}{\partial x}(x^2y^2) = 2xy^2$$

$$\frac{\partial}{\partial y}(-2y^5 + z^2) = -10y^4$$

$$\frac{\partial}{\partial \theta_2}(5\theta_1 + 2\theta_2 - 12\theta_3) = 2$$

$$\frac{\partial}{\partial \theta_2}(0.55-(5\theta_1 + 2\theta_2 - 12\theta_3)) = -2$$

梯度下降法

- 在单变量的函数中，梯度其实就是函数的微分，代表着函数在某个给定点的切线的斜率
- 在多变量函数中，梯度是一个向量，如下式所示，向量有方向，梯度的方向就指出了函数在给定点的上升最快的方向

$$J(\Theta) = 0.55 - (5\theta_1 + 2\theta_2 - 12\theta_3)$$

$$\nabla J(\Theta) = \left\langle \frac{\partial J}{\partial \theta_1}, \frac{\partial J}{\partial \theta_2}, \frac{\partial J}{\partial \theta_3} \right\rangle = (-5, -2, 12)$$

- 梯度的方向实际就是函数在此点上升最快的方向，而我们需要朝着下降最快的方向走，自然就是负的梯度的方向，所以梯度下降法需要加上负号

梯度下降法

梯度下降法的基本公式：

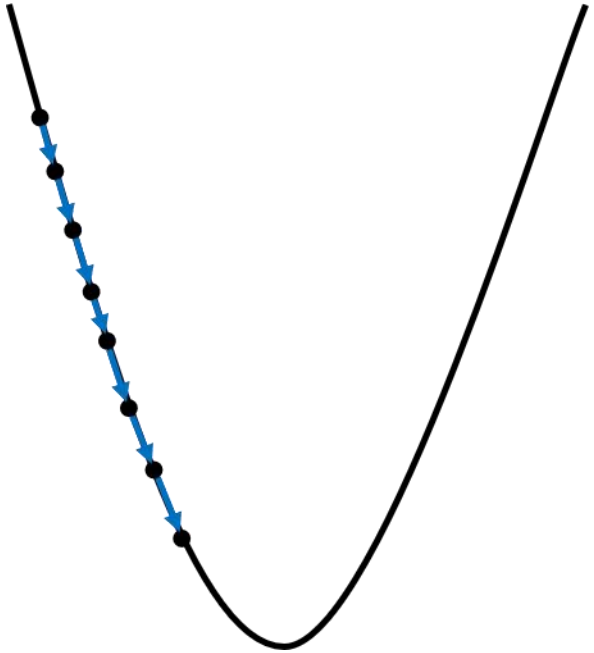
$$\Theta^1 = \Theta^0 - \alpha \nabla J(\Theta)$$

J 是关于 Θ 的一个函数，我们当前所处的位置为 Θ^0 点，要从这个点走到 J 的最小值点，也就是山底。首先我们先确定前进的方向，也就是梯度的反向，然后走一段距离的步长，也就是 α，走完这个段步长，就到达了 Θ^1 这个点。

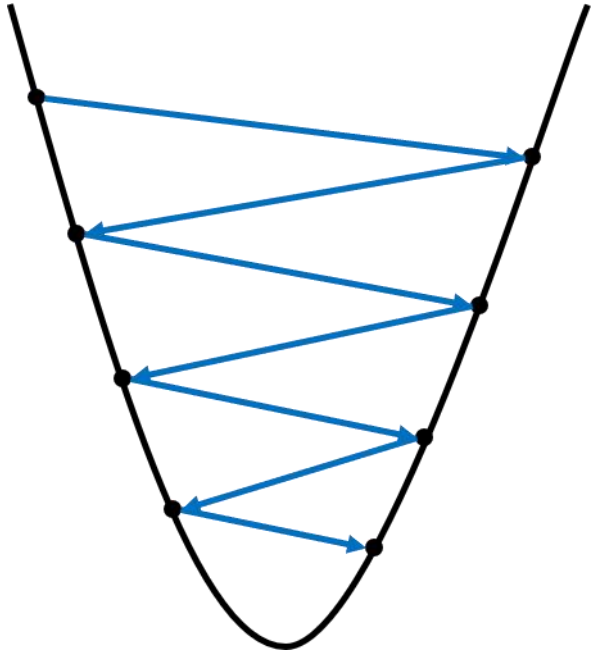
其中 α 在梯度下降法中被称作为学习率或者步长，意味着我们可以通过α来控制每一步走的距离

梯度下降法

- α太小可能导致迟迟走不到最低点或者无法跳出局部极小点；
- α太大可能导致错过最低点，无法稳定收敛。



学习率过小收敛速度慢



学习率过大错过最低点

单变量梯度下降法-示例

- 设单变量函数为：$J(\theta) = \theta^2$
- 假设起点：$\theta^0 = 1$，学习率：$\alpha = 0.4$
- 函数梯度为：$J'(\theta) = 2\theta$
- 根据梯度下降的公式：$\theta 1 = \theta 0 \quad \alpha \nabla J(\theta)$
- 迭代过程为：

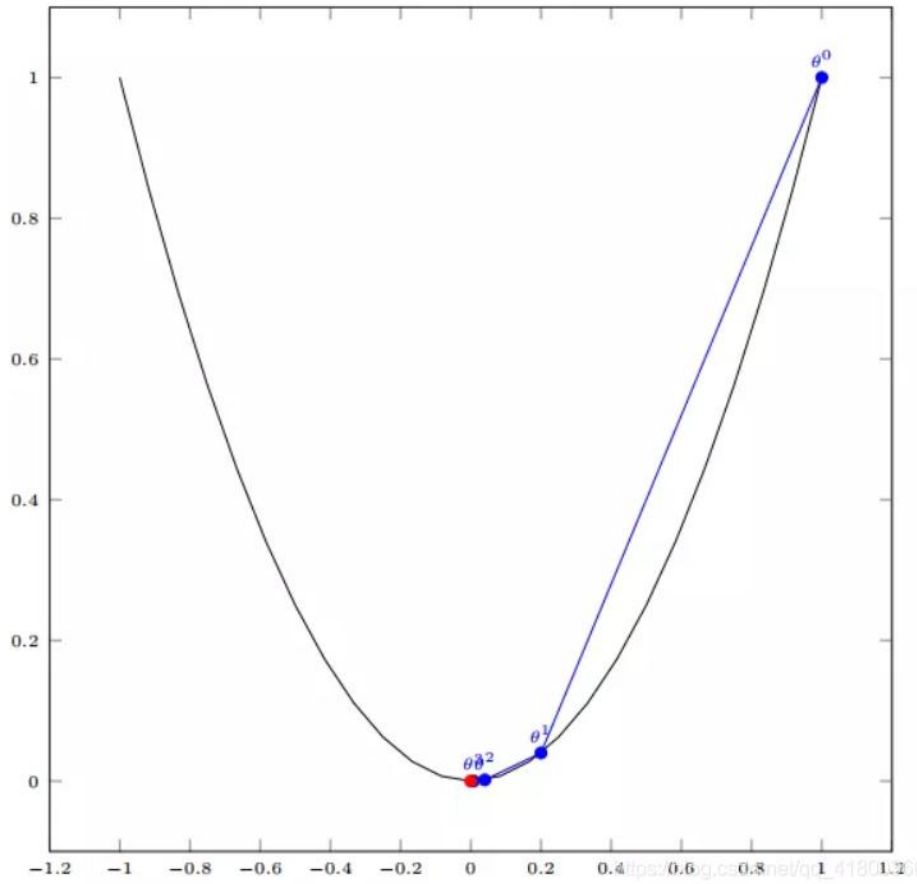
$$\theta^0 = 1$$

$$\theta^1 = \theta^0 - \alpha * J'(\theta^0) = 1 - 0.4 * 2 = 0.2$$

$$\theta^2 = \theta^1 - \alpha * J'(\theta^1) = 0.2 - 0.4 * 0.4 = 0.04$$

$$\theta^3 = 0.008$$

$$\theta^4 = 0.0016$$



多变量梯度下降法-示例

- 设多变量函数为：$J(\Theta) = \theta_1^2 + \theta_2^2$
- 假设起点：$\Theta^0 = (1,3)$，学习率：$\alpha = 0.1$
- 函数梯度为：$\nabla J(\Theta) = < 2\theta_1, 2\theta_2 >$
- 根据梯度下降的公式：$\Theta^1 = \Theta^0 - \alpha\nabla J(\Theta)$
- 迭代过程为：

$$\Theta^0 = (1,3)$$

$$\Theta^1 = \Theta^0 - \alpha\nabla J(\Theta) = (1,3) - 0.1 * (2,6) = (0.8,2.4)$$

$$\Theta^2 = (0.8,2.4) - 0.1 * (1.6,4.8) = (0.64,1.92)$$

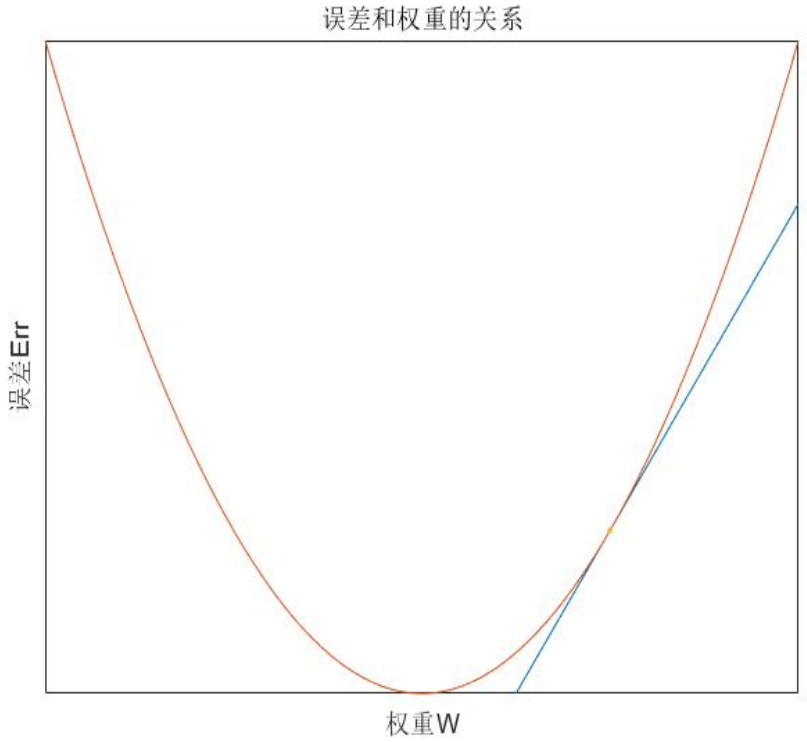
$$\Theta^3 = (0.5124,1.536)$$

$$\Theta^4 = (0.4096,1.228800000000001)$$

$$\vdots$$

梯度下降法

神经网络中常见的误差计量方式：



梯度下降法

我们可以推导出误差 Err 对于权矩阵 $\boldsymbol{W}$ 的梯度：

$$\frac{\partial \boldsymbol{Err}}{\partial \boldsymbol{W}} = \frac{\partial(\frac{1}{2}(\boldsymbol{Y}-\boldsymbol{O})^2)}{\partial \boldsymbol{W}} = (\boldsymbol{Y}-\boldsymbol{O})\frac{\partial(\boldsymbol{Y}-\boldsymbol{O})}{\partial \boldsymbol{W}} = (\boldsymbol{Y}_i-\boldsymbol{O})\frac{\partial(\boldsymbol{Y}-(\boldsymbol{WX}+\boldsymbol{B}))}{\partial \boldsymbol{W}}$$

$$= (\boldsymbol{Y}-\boldsymbol{O})(-\boldsymbol{X})$$

权矩阵 $\boldsymbol{W}$ 的更新应该沿着梯度相反的方向进行

$$\Delta \boldsymbol{W} = \alpha(\boldsymbol{Y}-\boldsymbol{O})\boldsymbol{X}$$

梯度下降法

同理，对于偏置矩阵 B 也可以做同样的分析，得到：

$$\frac{\partial Err}{\partial B} = -(Y - O)$$

$$\Delta B = \alpha(Y - O)$$

3.3 反向传播算法

基本原理

梯度下降法

反向传播算法（BP算法）

BP算法分析

BP算法改进

RBF网络

BP算法概述

- 多层网络的学习能力比单层感知机强得多，想要训练多层网络，使用简单的感知机学习规则是远远不够的。

- 因此，提出了**反向传播算法**（BP算法）来训练多层前馈神经网络。

BP算法概述

- BP算法的学习过程由正向传播过程和反向传播过程组成。

➢ 在**正向传播**过程中，输入信息通过输入层经隐藏层，逐层处理并传向输出层。

➢ 如果在输出层得不到期望的输出值，则取输出与期望的误差的平方和作为目标函数，转入**反向传播**，逐层求出目标函数对各神经元权值的偏导数，构成目标函数对权值向量的梯度，作为修改权值的依据，网络的学习在权值修改过程中完成。

➢ 误差达到所期望值时，网络学习结束。

BP算法概述

●权重更新

对于每个神经元上的权重，按照以下步骤进行更新：

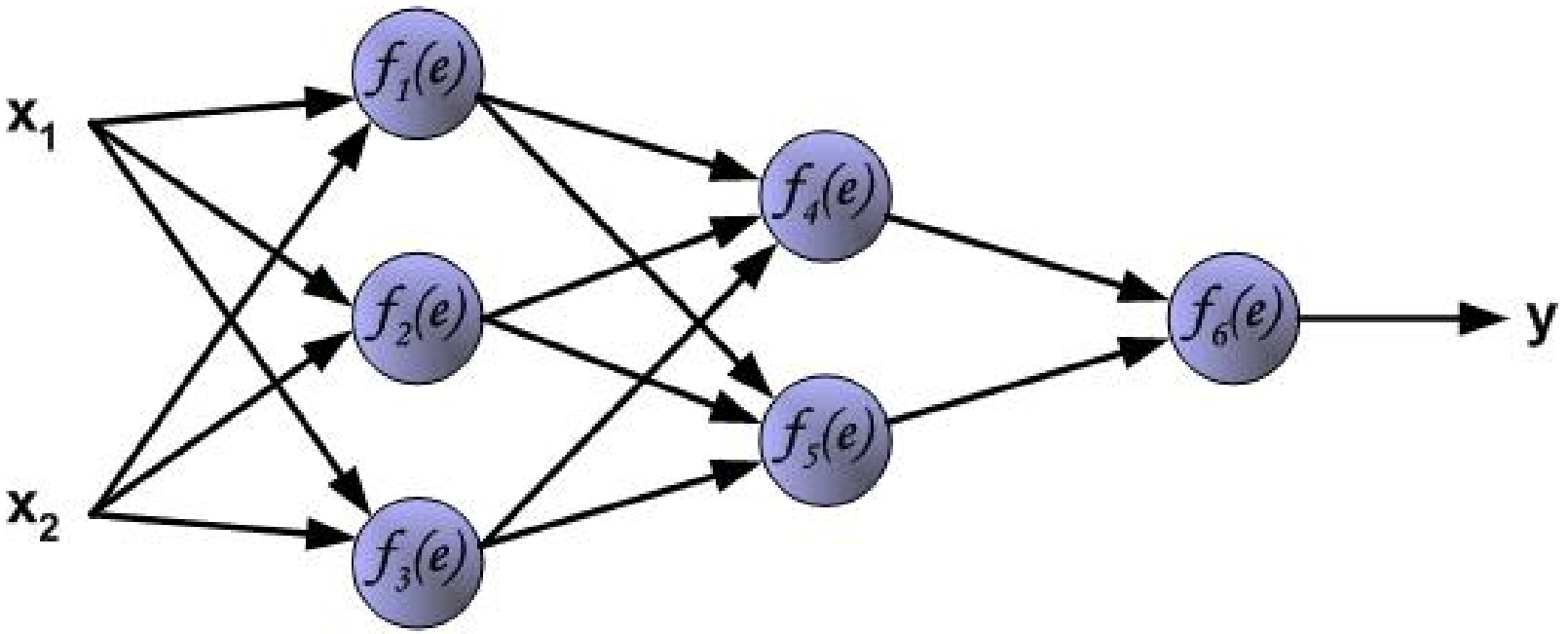
- 基于误差推导出权重的梯度 $\frac{\partial E}{\partial w}$;
- 将这个梯度乘上一个比例(学习率 α)并取反后加到权重上。

$$\Delta w = -\alpha \frac{\partial E}{\partial w}$$

$$w \leftarrow w + \Delta w$$

BP算法过程

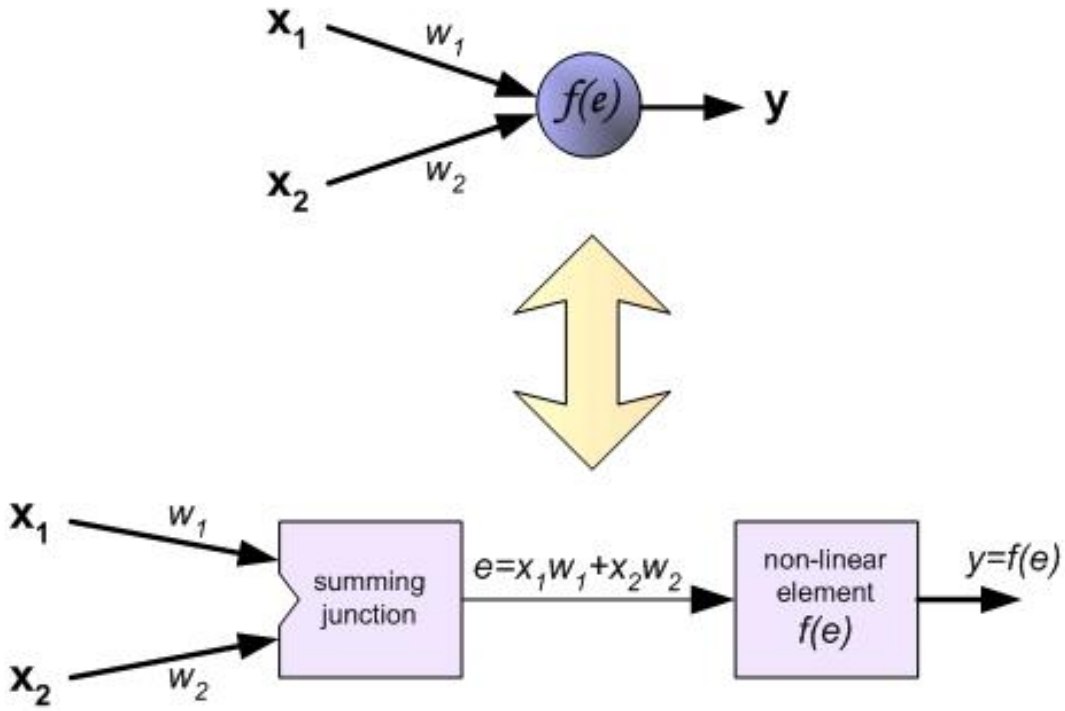
为了更好的描述反向传播算法，这里使用具有两个输入和一个输出的多层感知机作为示例：



http://galaxy.agh.edu.pl/~vlsi/AI/backp_t_en/backprop.html

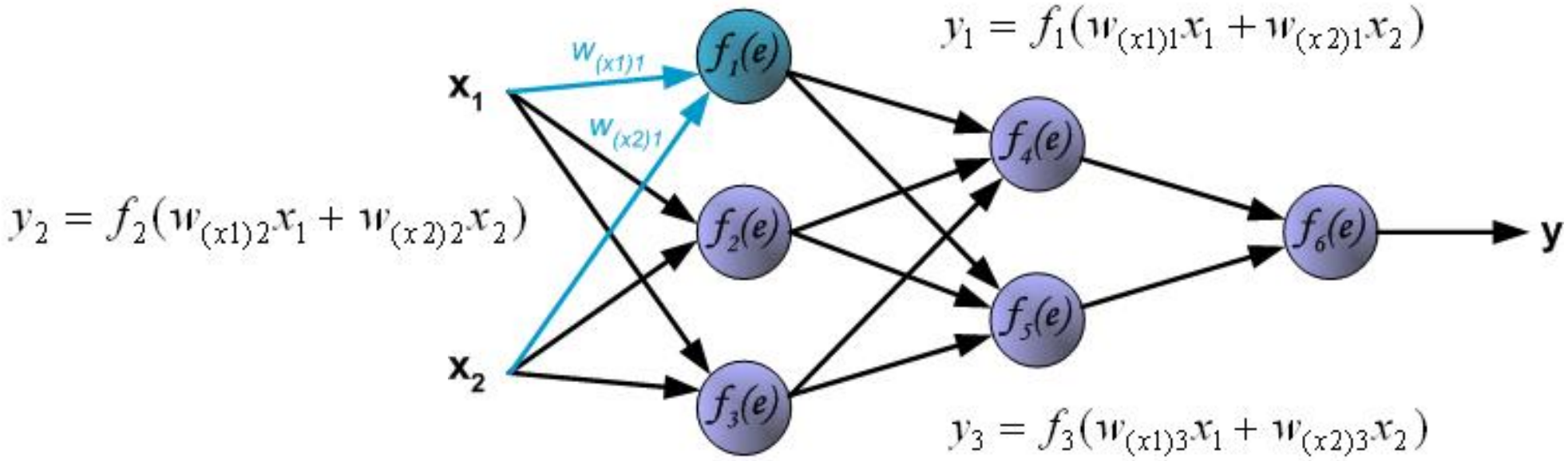
BP算法过程

- 每个神经元的处理过程由两部分组成的。第一个方块包含权重系数和输入信号。第二个方块实现非线性函数，即激活函数。
- e 是由上一层输入加权求和后得到的，$y = f(e)$ 是通过激活函数后得到的神经元输出信号。



BP算法过程-前向传播阶段

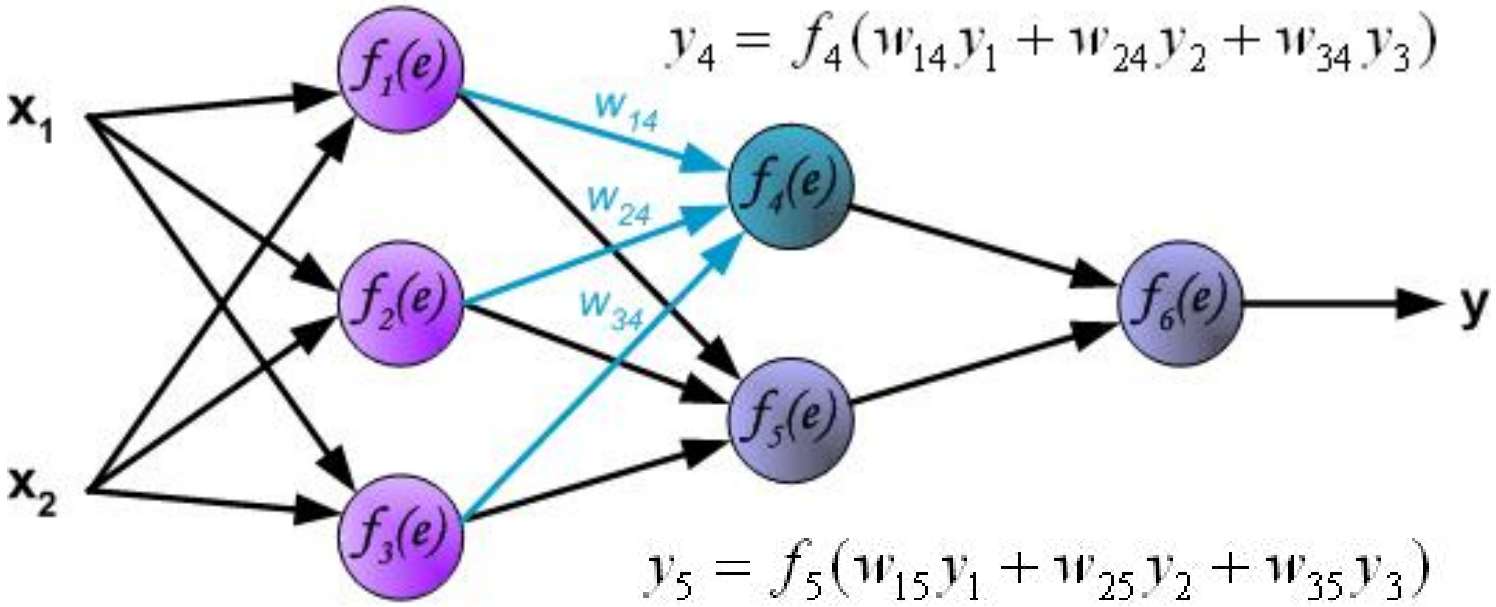
- 前向传播阶段从训练集中的两个输入信号开始，在这个阶段我们可以确定每个网络层中的每个神经元的输出信号值。



- 其中$w(x_m)_n$代表网络输入x_m和神经元n之间的连接权重，y_n代表神经元n的输出信号。

BP算法过程-前向传播阶段

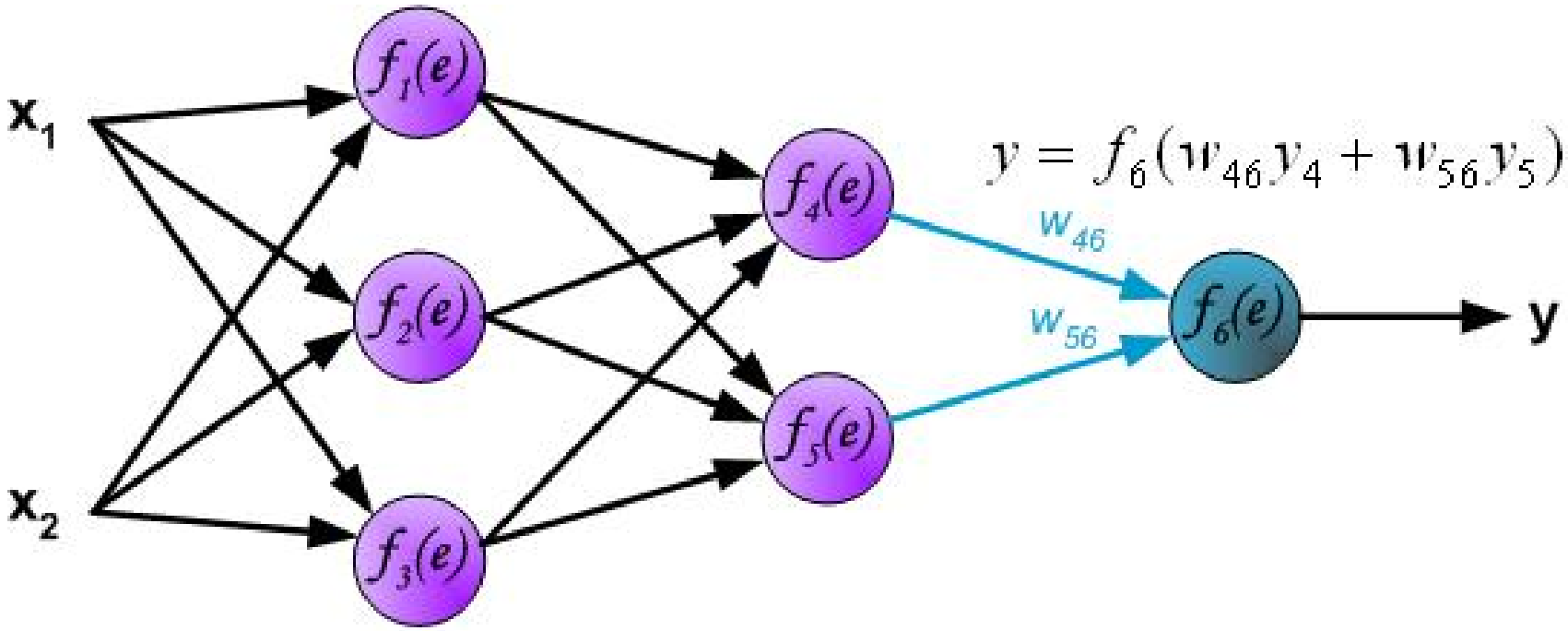
- 上一层神经元的输出信号继续通过隐藏层传播。



- 其中w_{mn}代表输入神经元m和输出神经元n之间的连接权重， y_n代表神经元n的输出信号。

BP算法过程-前向传播阶段

- 最后传播到输出层，得到输出 y。



- 至此，前向传播阶段完成。

BP算法过程-反向传播阶段

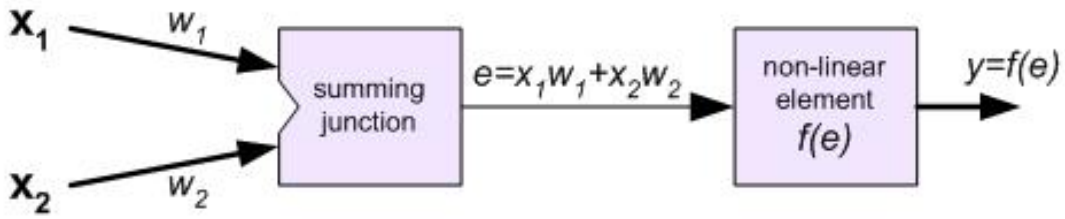
- 前向传播得到的计算输出 y 和训练集的真实标签 z 会存在一定的误差：

$$E = \frac{1}{2}(z - y)^2$$

- 为了方便表示，定义误差信号：

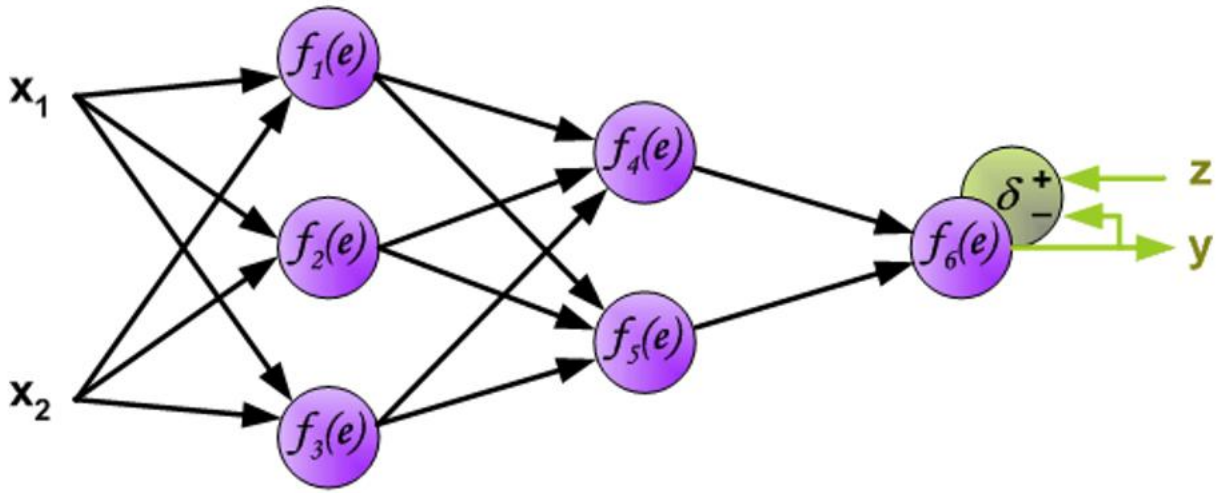
$$\delta_n = -\frac{\partial E}{\partial e_n} = -\frac{\partial E}{\partial y_n}\frac{\partial y_n}{\partial e_n} = -\frac{\partial E}{\partial y_n}\frac{df(e_n)}{de_n} = -\frac{\partial E}{\partial y_n}f'(e_n)$$

其中$e_n = \sum_i w_{in} y_i$是神经元n的加权和，$y_n = f(e_n)$是神经元n的输出信号；



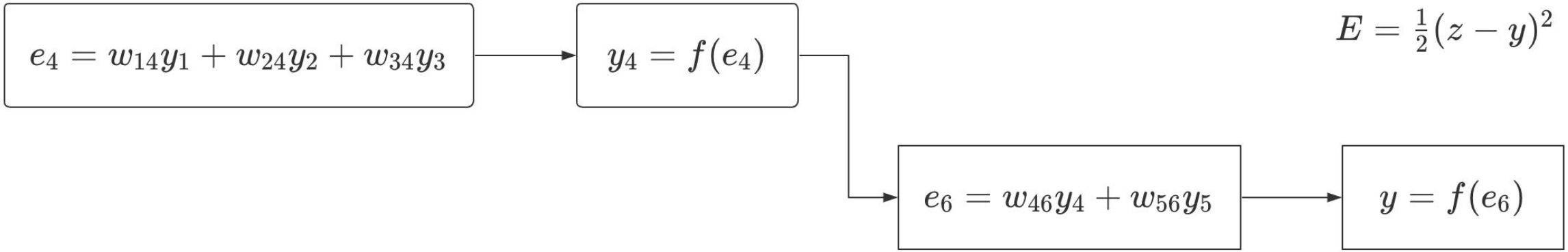
BP算法过程-反向传播阶段

- 则输出层误差信号：



BP算法过程-反向传播阶段

- 反向传播，求上个隐藏层的误差信号δ_4：



$$E = \frac{1}{2}(z-y)^2$$

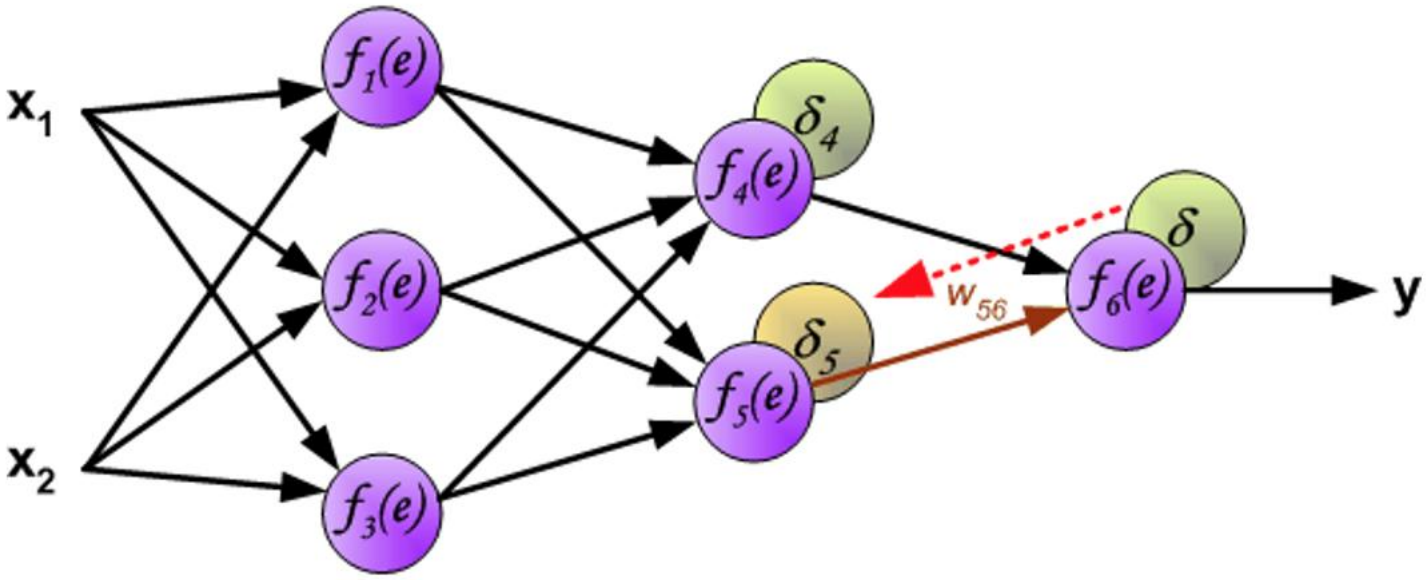
$$\frac{\partial E}{\partial y_4} = \frac{\partial E}{\partial y}\frac{\partial y}{\partial e_6}\frac{\partial e_6}{\partial y_4} = -(z-y)f'(e_6)\frac{\partial(w_{46}y_4 + w_{56}y_5)}{\partial y_4} = -(z-y)f'(e_6)w_{46}$$

$$\delta_4 = -\frac{\partial E}{\partial y_4}f'(e_4) = (z-y)f'(e_6)w_{46}f'(e_4) = \delta w_{46}f'(e_4)$$

BP算法过程-反向传播阶段

- 同理，误差信号δ_5：

$$\delta_5 = \delta w_{56} f'(e_5)$$

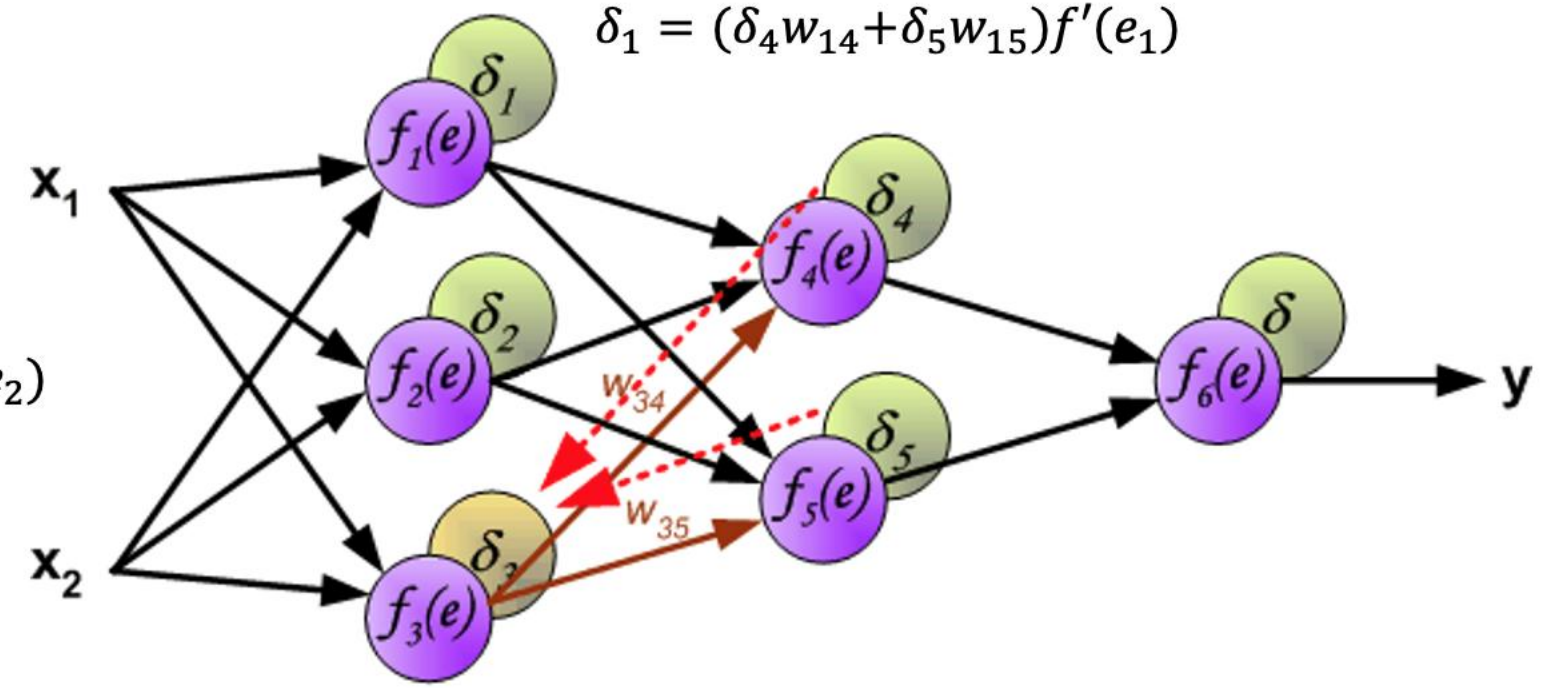


BP算法过程-反向传播阶段

- 继续反向传播得到每个神经元的误差信号δ_1，δ_2，δ_3

$$\delta_1 = (\delta_4 w_{14} + \delta_5 w_{15}) f'(e_1)$$

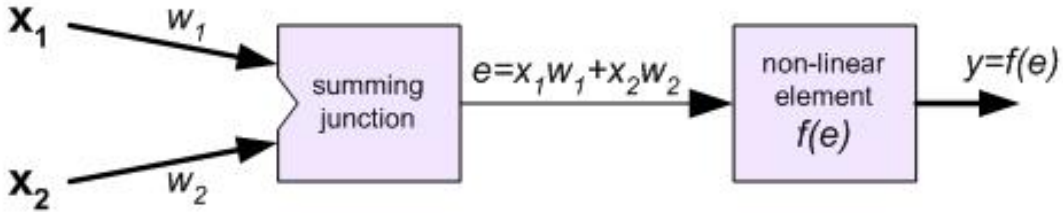
$$\delta_2 = (\delta_4 w_{24} + \delta_5 w_{25}) f'(e_2)$$



$$\delta_3 = (\delta_4 w_{34} + \delta_5 w_{35}) f'(e_3)$$

BP算法过程-权值更新

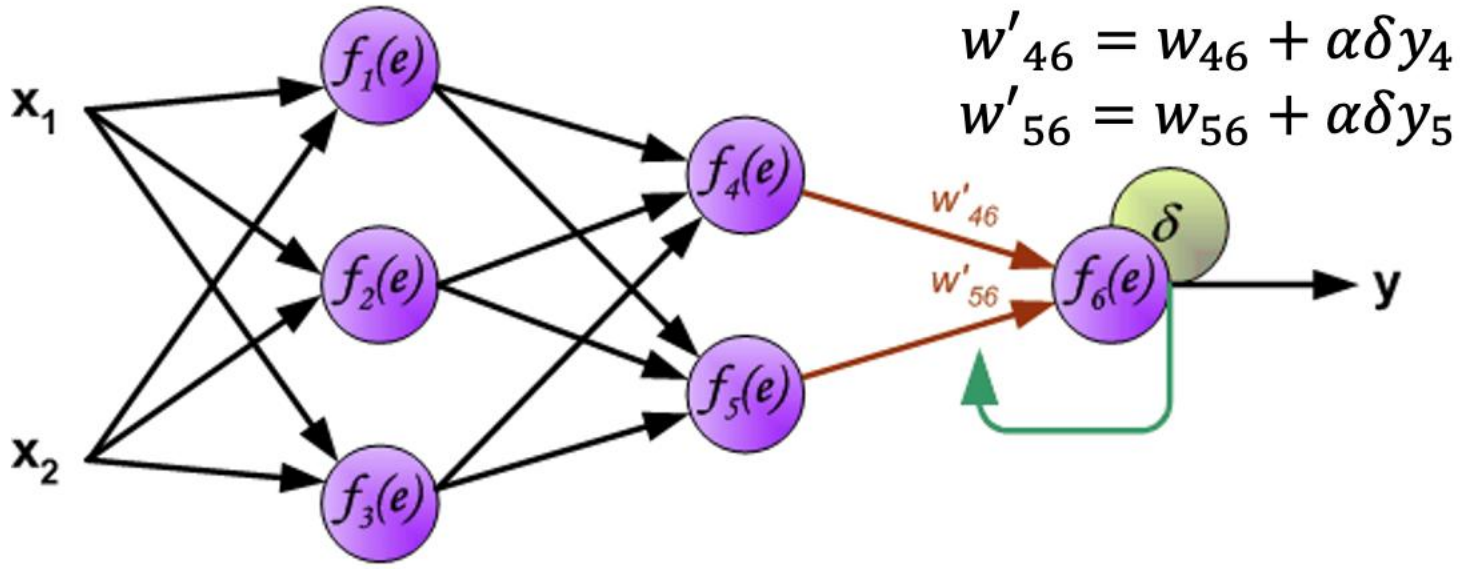
- $e_n = \sum_i w_{in} y_i$是神经元n的加权和；
- $y_n = f(e_n)$是神经元n的输出信号；
- $\delta_n = -\frac{\partial E}{\partial e_n}$是神经元$n$的误差信号；



- 用梯度下降法更新权值，可得：$w'_{mn} = w_{mn} + \Delta w_{mn}$，其中$\Delta w_{mn} = -\alpha \frac{\partial E}{\partial w_{mn}}$
- 通过链式法则，可得：$\frac{\partial E}{\partial w_{mn}} = \frac{\partial E}{\partial e_n} \frac{\partial e_n}{\partial w_{mn}} = -\delta_n \frac{\partial \sum_i w_{in} y_i}{\partial w_{mn}} = -\delta_n y_m$
- 综上，可得：$w'_{mn} = w_{mn} + \alpha \delta_n y_m$

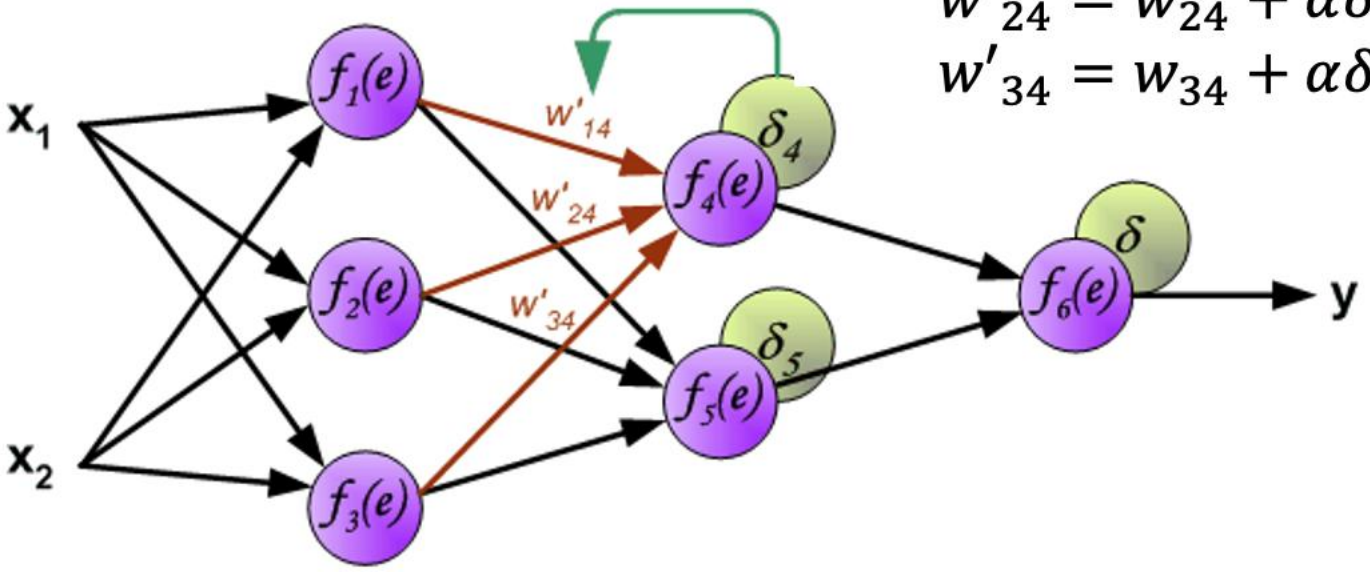
BP算法过程-权值更新

- 根据公式，调整各层权值：



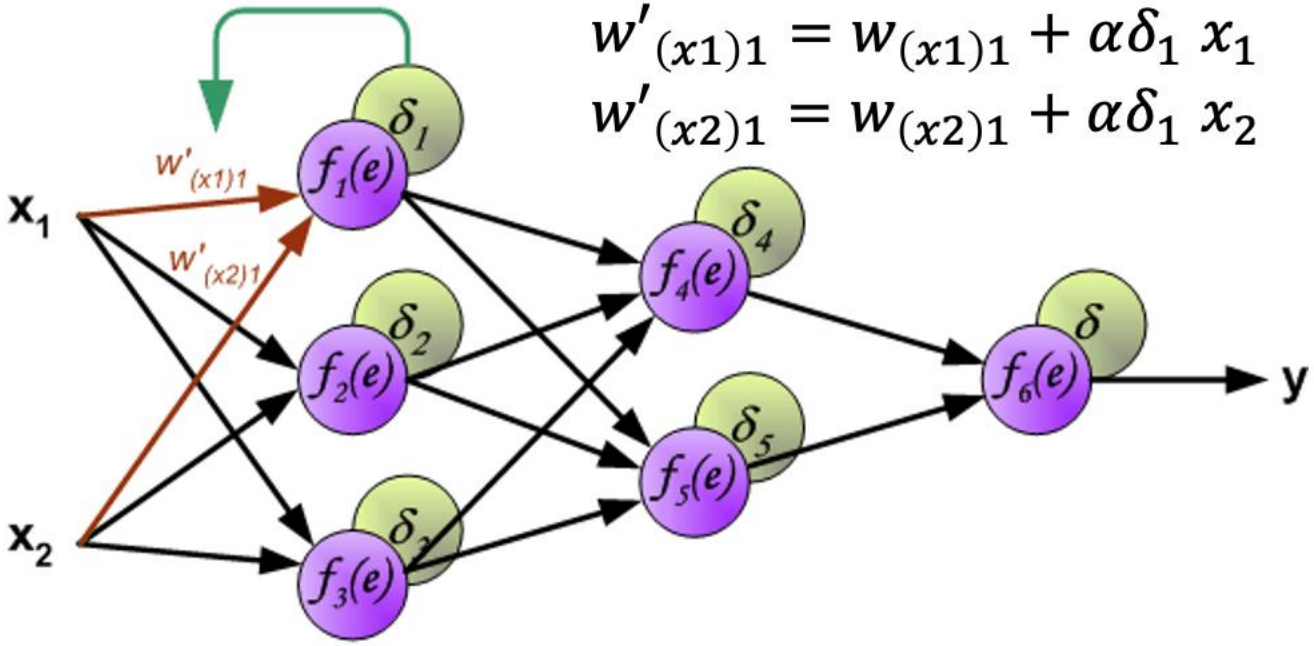
BP算法过程-权值更新

$$w'_{14} = w_{14} + \alpha\delta_4 y_1$$
$$w'_{24} = w_{24} + \alpha\delta_4 y_2$$
$$w'_{34} = w_{34} + \alpha\delta_4 y_3$$



BP算法过程-权值更新

$$w'_{(x1)1} = w_{(x1)1} + \alpha\delta_1 x_1$$
$$w'_{(x2)1} = w_{(x2)1} + \alpha\delta_1 x_2$$



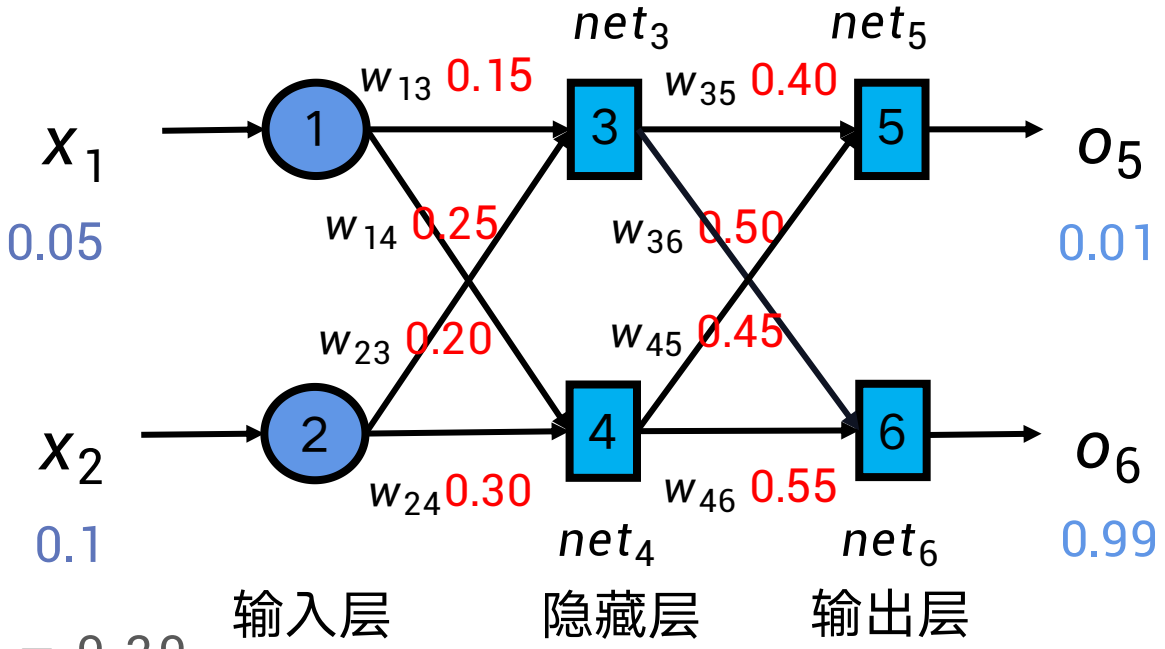
- 至此，完成反向传播和权重更新。之后继续传入新的样本数据，重复前面过程，通过调整权重，不断减小误差。

BP算法过程

1. 初始化权值；
2. 输入训练样本对，计算各层输出；
3. 计算网络输出误差；
4. 计算各层误差信号；
5. 调整各层权值；
6. 对所有样本重复2-5步，直到网络总误差达到精度要求。

BP算法计算-示例

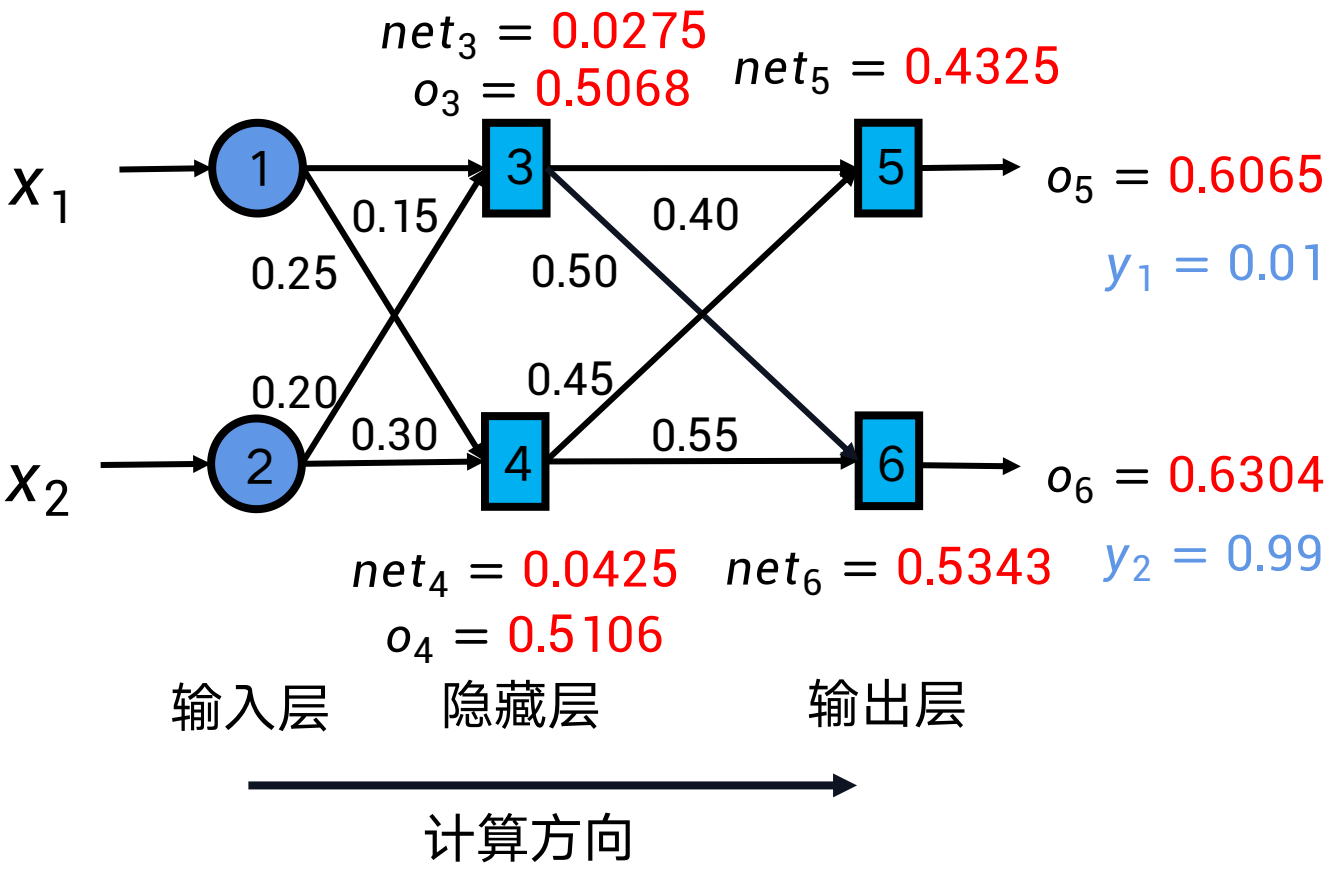
第一步：输入输出与网络初始化



- 训练样本：$\boldsymbol{x} = [0.05,\ 0.1],\ \boldsymbol{y} = [0.01,\ 0.99]$
- 随机初始化权矩阵：
  - $\boldsymbol{W}^{(1)}: w_{13} = 0.15, w_{14} = 0.25, w_{23} = 0.20, w_{24} = 0.30$
  - $\boldsymbol{W}^{(2)}: w_{35} = 0.40, w_{36} = 0.50, w_{45} = 0.45, w_{46} = 0.55$
- 神经元的激活函数：$f(net) = \frac{1}{1+e^{-net}}$
- 学习率：$\alpha = 0.5$

BP算法计算

第二步：前向传播阶段



- 前向计算

$$net_3 = x_1 \cdot w_{13} + x_2 \cdot w_{23}$$
$$= 0.05 \times 0.15 + 0.2 \times 0.1$$
$$= 0.0275$$

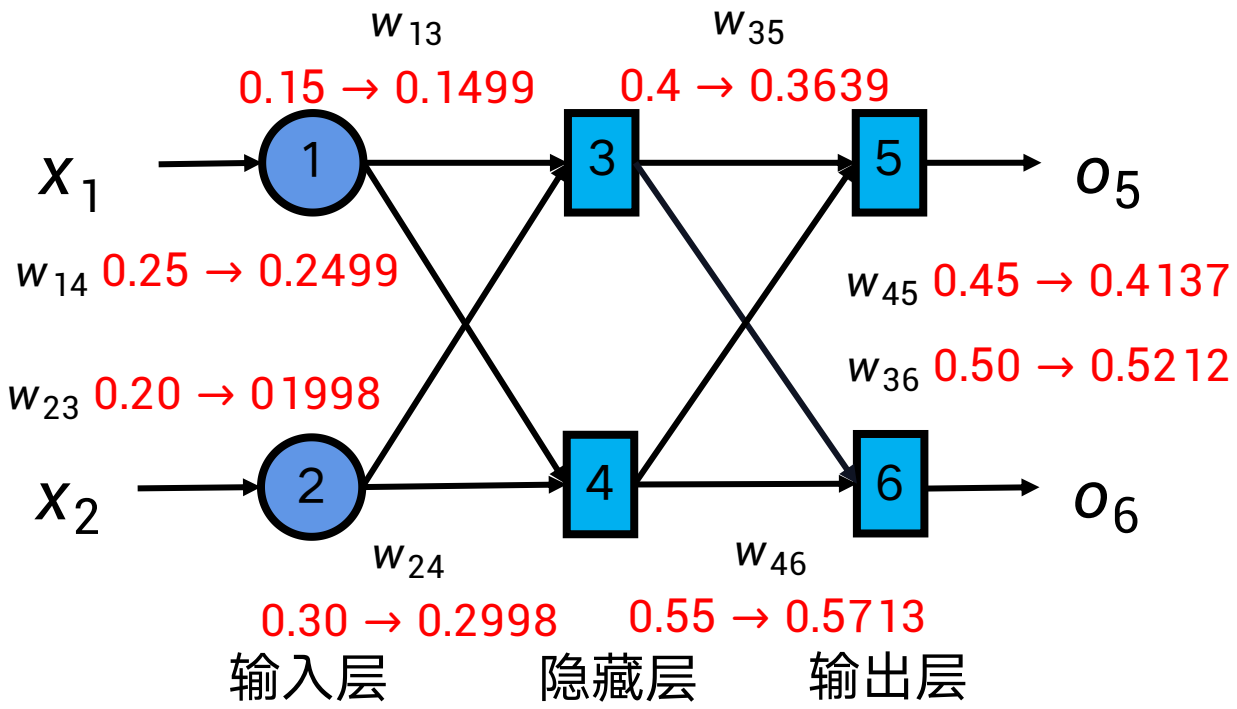
$$o_3 = \frac{1}{1+e^{-net}} = \frac{1}{1+e^{-0.0275}}$$
$$= 0.5068$$

$$net_5 = o_3 \cdot w_{35} + o_4 \cdot w_{45}$$
$$= 0.5068 \times 0.40 + 0.5106 \times 0.45$$
$$= 0.4325$$

……

BP算法计算

第三步：反向传播阶段



- 计算误差信号：

$$\frac{\partial o_j}{\partial net_j}=\frac{\partial(\frac{1}{1+e^{-net_j}})}{\partial net_j}=o_j(1-o_j)$$

$$\delta_5=(y_1-o_5)o_5(1-o_5)$$
$$=(0.01-0.6065)\times 0.6065\times(1-0.6065)=-0.1424$$

$$\delta_6=(y_2-o_6)o_6(1-o_6)$$
$$=(0.99-0.6304)\times 0.6304\times(1-0.6304)=0.0838$$

……

- 调整各层权值：

$$w_{35}^{+}=w_{35}+\alpha\delta_5 o_3$$
$$=0.4-0.5\times 0.1424\times 0.5068=0.3639$$

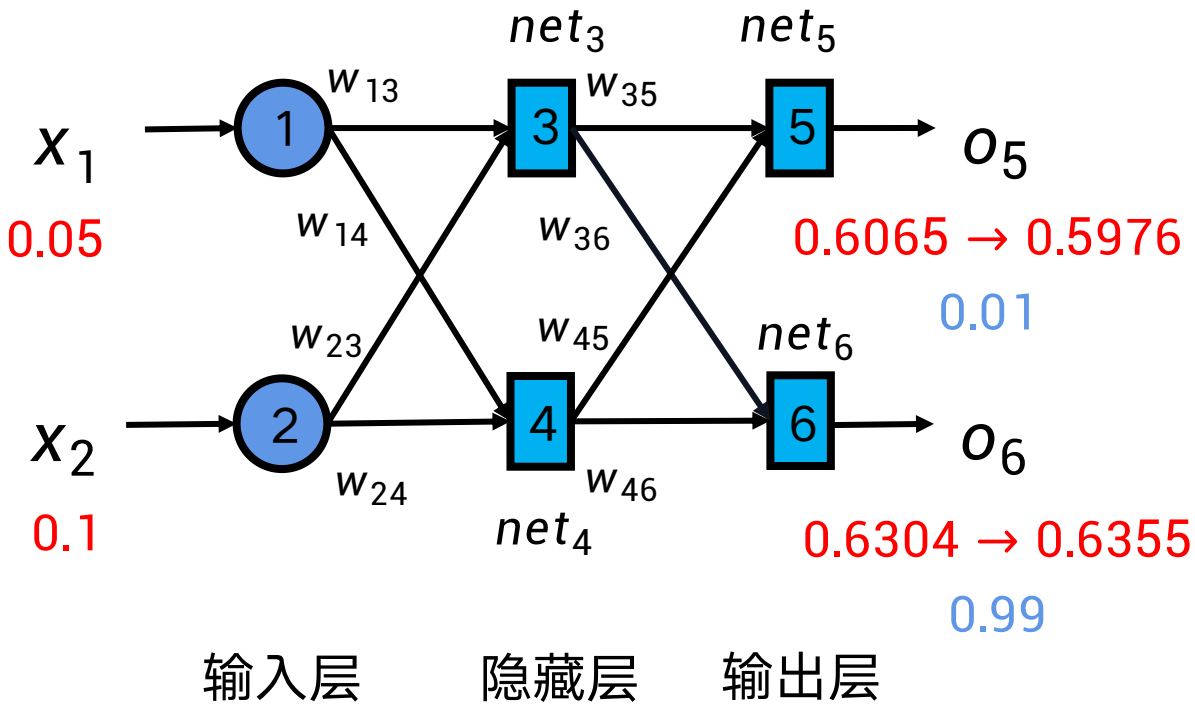
$$w_{13}^{+}=w_{13}+\alpha o_3(1-o_3)\left(\delta_5 w_{35}+\delta_6 w_{36}\right)x_1$$
$$=0.15+0.5\times 0.5068\times(1-0.5068)\times$$
$$(-0.1424\times 0.4+0.0838\times 0.5)\times 0.05$$
$$=0.1499$$

……

BP算法计算

第四步：训练结果

- 第二次前向结果相比第一次前向结果更靠近目标输出，误差更小 E: 0.2355 → 0.2286
- 当经过10000轮训练之后，误差将减少到 $5.583e-0.5$，输出为 (0.0174,0.9825)



3.4 BP算法分析

基本原理

梯度下降法

反向传播算法（BP算法）

BP算法分析

BP算法改进

RBF网络

BP算法分析

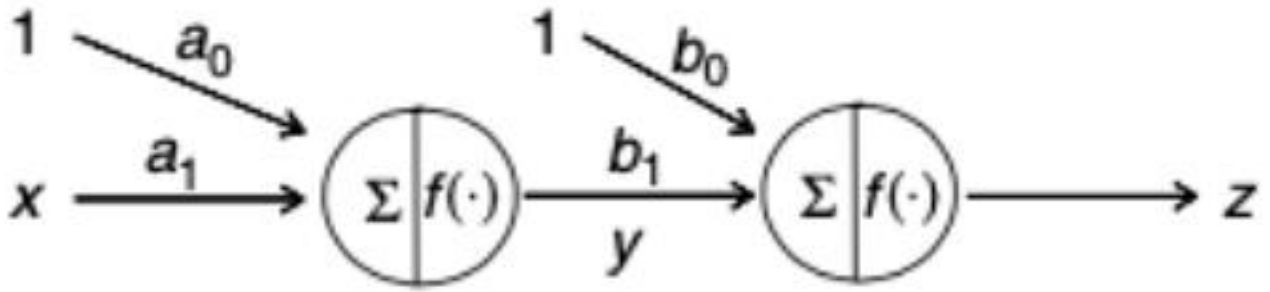
$$u = a_0 + a_1 x$$

$$y = f(u) = \frac{1}{1+e^{-u}} = \frac{1}{1+e^{-(a_0+a_1 x)}}$$

$$v = b_0 + b_1 y$$

$$z = f(v) = \frac{1}{1+e^{-v}} = \frac{1}{1+e^{-(b_0+b_1)\{\frac{1}{1+e^{-(a_0+a_1 x)}}\}}}$$

$$E = \frac{1}{2}(z-t)^2 = \frac{1}{2}\{\frac{1}{1+e^{-(b_0+b_1)\{\frac{1}{1+e^{-(a_0+a_1 x)}}\}}} - t\}^2$$



初始化参数

a_0	a_1	b_0	b_1	x	t
0.3	0.2	−0.1	0.4	0.7853	0.707
				1.571	1.00

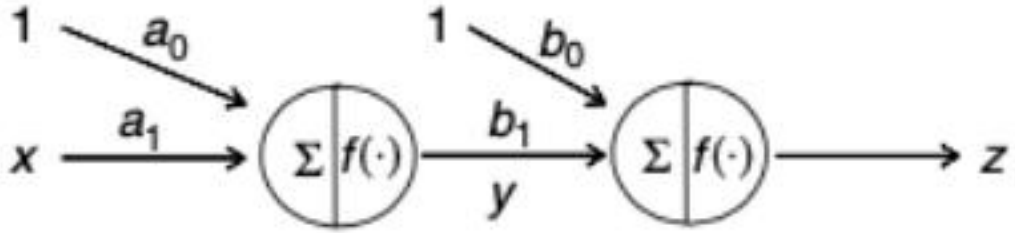
输出神经元权值

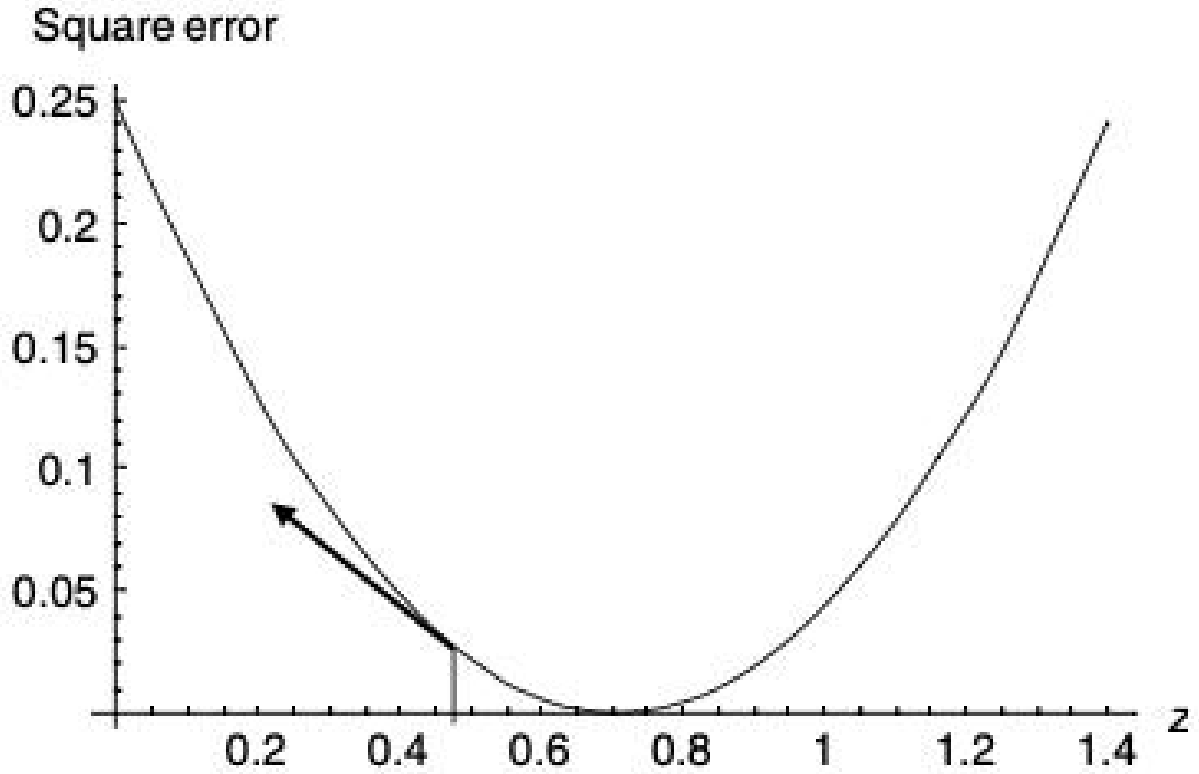
根据链式规则：

$$\frac{\partial E}{\partial b} = \frac{\partial E}{\partial z} \cdot \frac{\partial z}{\partial v} \cdot \frac{\partial v}{\partial b}$$

1. 误差对网络输出的灵敏度:

$$\frac{\partial E}{\partial z} = z - t$$





输出神经元权值

2. 网络输出对输出神经元的加权和 v 的变化的灵敏度

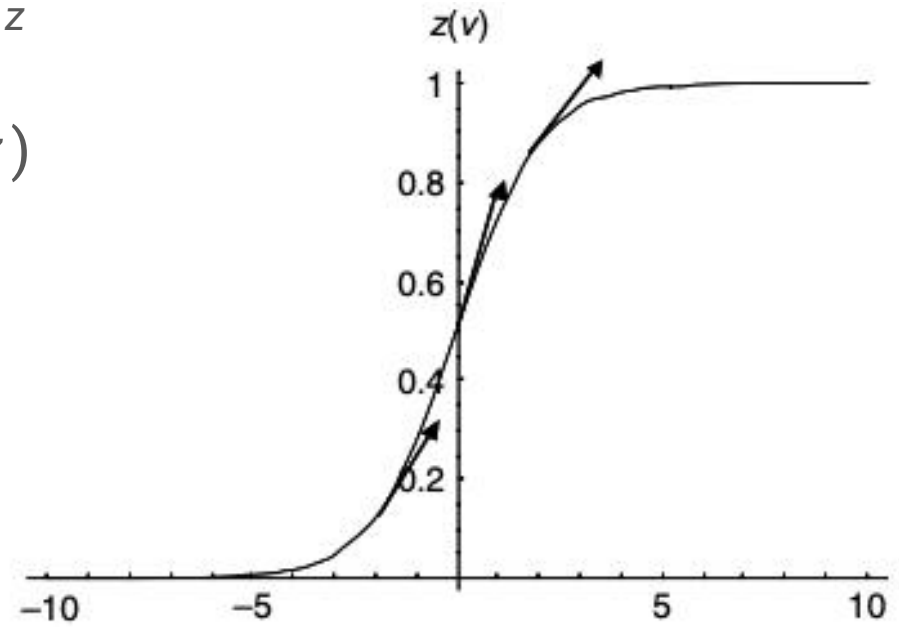
$$\frac{\partial z}{\partial v} = \left(\frac{1}{1+e^{-v}}\right)' = \frac{e^{-v}}{(1+e^{-v})^{-2}}$$

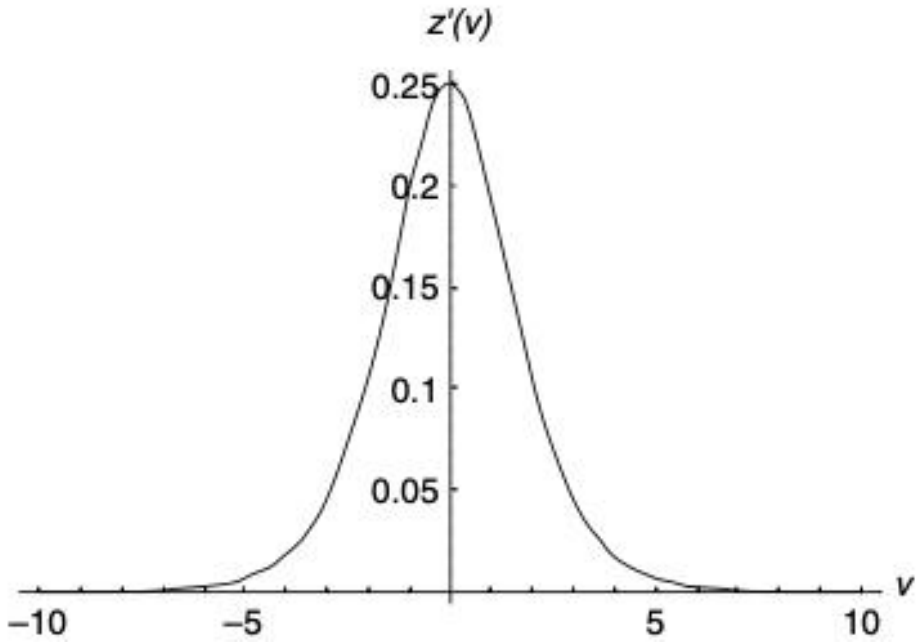
已知：$1+e^{-v} = \frac{1}{z}$

则：$\frac{\partial z}{\partial v} = z(1-z)$

根据链式规则：

$$\frac{\partial E}{\partial b} = \frac{\partial E}{\partial z} \cdot \frac{\partial z}{\partial v} \cdot \frac{\partial v}{\partial b}$$





输出神经元权值

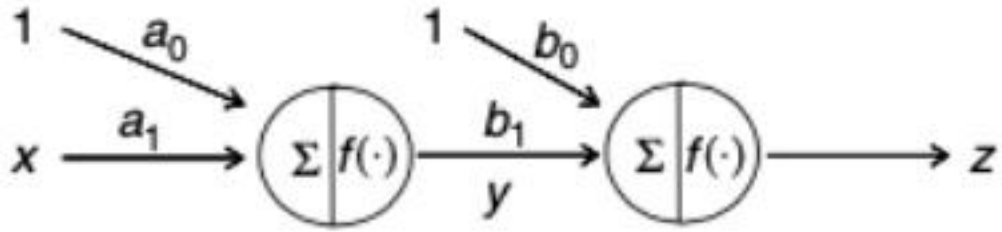
3. 加权和 v 对输出神经元权值的变化的灵敏度

$$\frac{\partial v}{\partial b_1} = y$$

$$\frac{\partial v}{\partial b_0} = 1$$

根据链式规则：

$$\frac{\partial E}{\partial b} = \frac{\partial E}{\partial z} \cdot \frac{\partial z}{\partial v} \cdot \frac{\partial v}{\partial b}$$



输出神经元权值

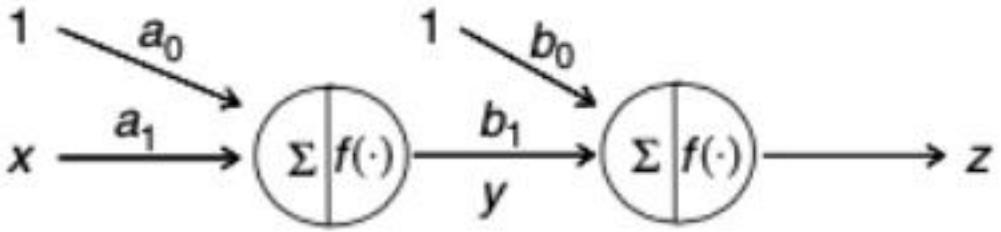
输出神经元权值误差梯度

$$\frac{\partial E}{\partial b_1} = (z - t)z(1 - z)y = py$$

$$\frac{\partial E}{\partial b_0} = (z - t)z(1 - z) = p$$

根据链式规则：

$$\frac{\partial E}{\partial b} = \frac{\partial E}{\partial z} \cdot \frac{\partial z}{\partial v} \cdot \frac{\partial v}{\partial b}$$

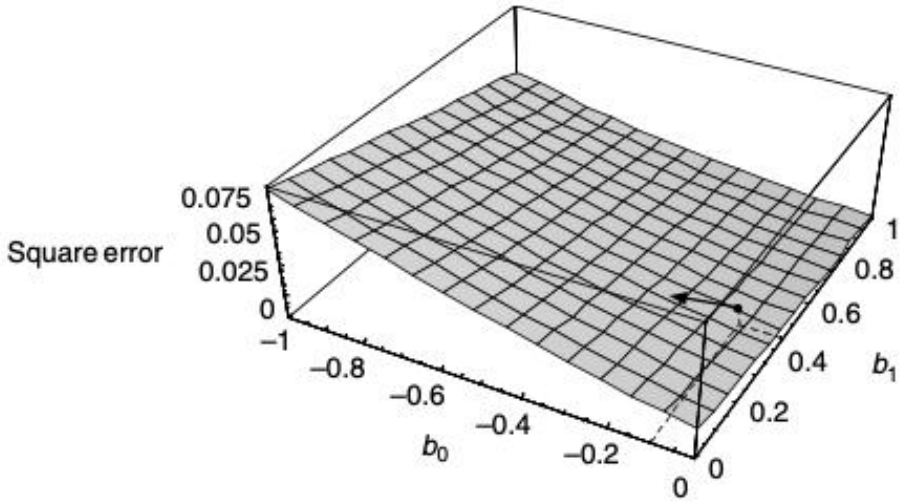


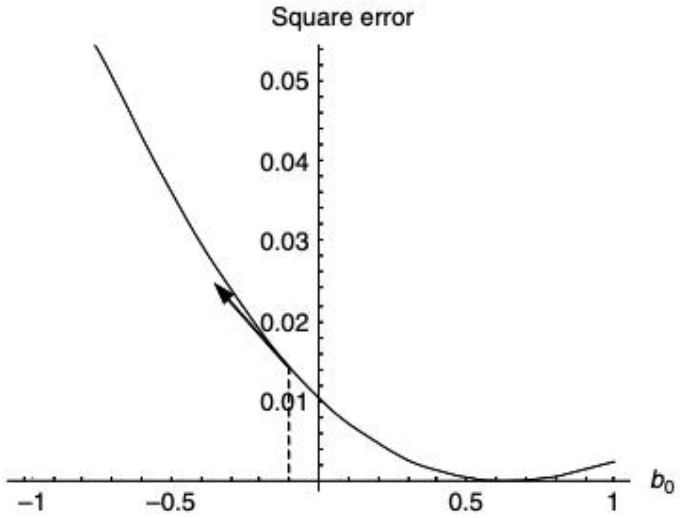
输出神经元权值

输出神经元权值误差梯度

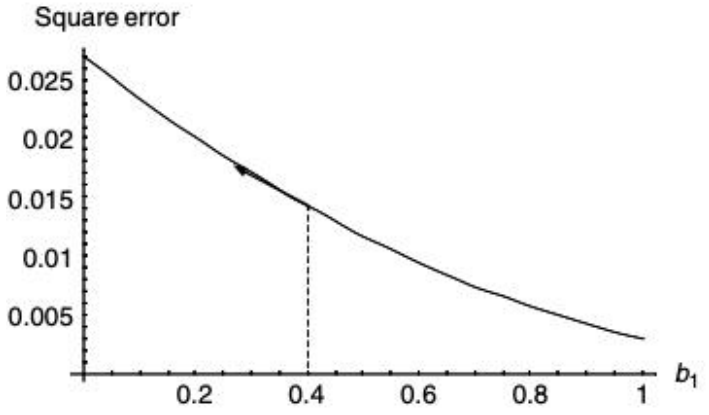
$$\frac{\partial E}{\partial b_1} = (z - t)z(1 - z)y = py$$

$$\frac{\partial E}{\partial b_0} = (z - t)z(1 - z) = p$$





$b_1 = 0$ 处 b_0 的误差

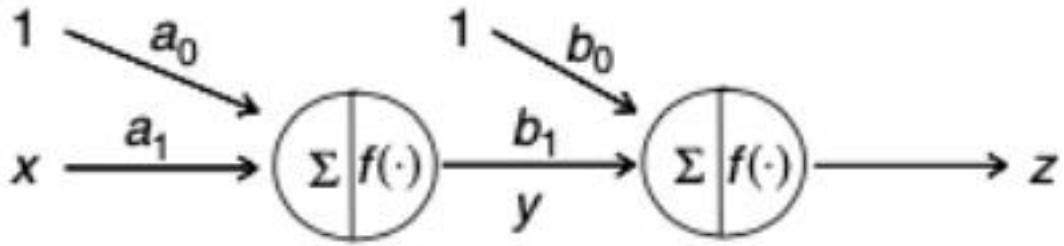


$b_0 = -0.1$ 处 b_1 的误差

隐藏神经元权值

根据链式规则：

$$\frac{\partial E}{\partial a} = \underbrace{\left(\frac{\partial E}{\partial z} \cdot \frac{\partial z}{\partial v} \cdot \frac{\partial v}{\partial y}\right)}_{1} \cdot \frac{\partial y}{\partial u} \cdot \frac{\partial u}{\partial a}$$



1. 已知：$v = b_0 + b_1 y$

$$\frac{\partial E}{\partial z} \cdot \frac{\partial z}{\partial v} = p$$

得：$\frac{\partial v}{\partial y} = b_1$

$$\frac{\partial E}{\partial z} \cdot \frac{\partial z}{\partial v} \cdot \frac{\partial v}{\partial y} = p b_1$$

隐藏神经元权值

2. $\frac{\partial y}{\partial u} = y(1-y)$

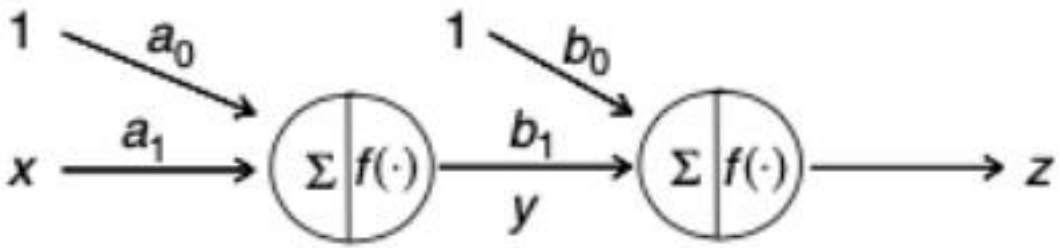
3. $\frac{\partial u}{\partial a_1} = x$

$\frac{\partial u}{\partial a_0} = 1$

根据链式规则：

$$\frac{\partial E}{\partial a} = \left(\frac{\partial E}{\partial z} \cdot \frac{\partial z}{\partial v} \cdot \frac{\partial v}{\partial y}\right) \cdot \boxed{\frac{\partial y}{\partial u}} \cdot \boxed{\frac{\partial u}{\partial a}}$$

2 3



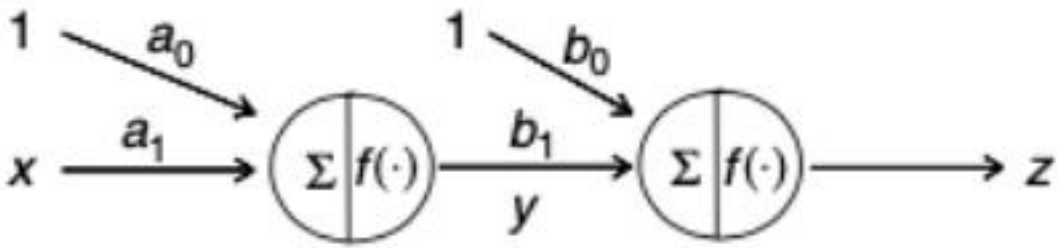
隐藏神经元权值

$$\frac{\partial E}{\partial a_1} = pb_1 y(1-y)x = qx$$

$$\frac{\partial E}{\partial a_0} = pb_1 y(1-y) = q$$

根据链式规则：

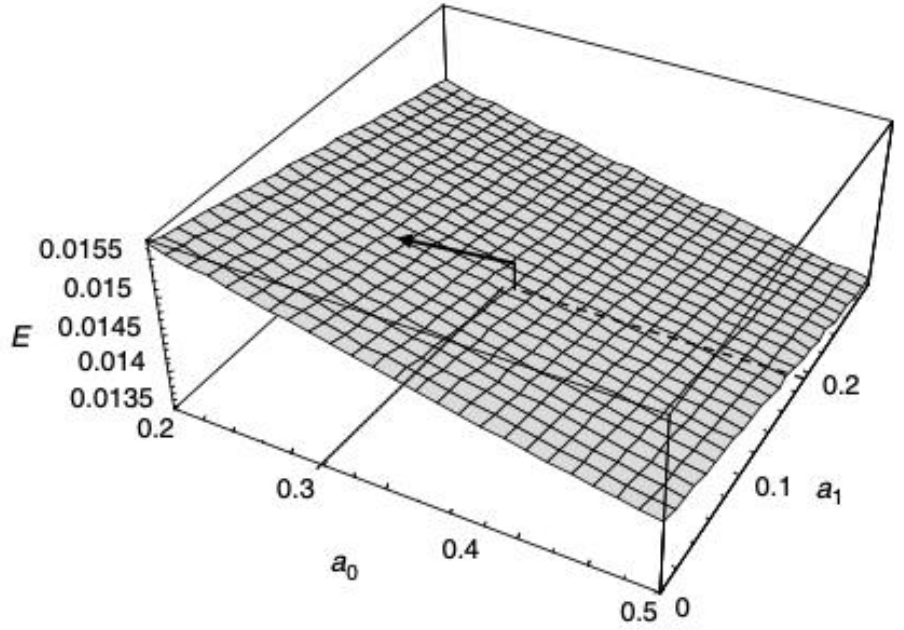
$$\frac{\partial E}{\partial a} = \left(\frac{\partial E}{\partial z} \cdot \frac{\partial z}{\partial v} \cdot \frac{\partial v}{\partial y}\right) \cdot \frac{\partial y}{\partial u} \cdot \frac{\partial u}{\partial a}$$

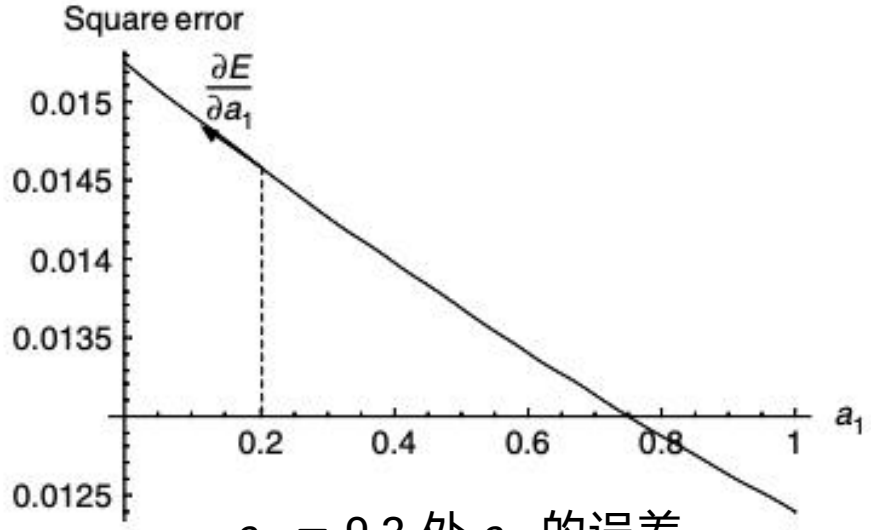


隐藏神经元权值

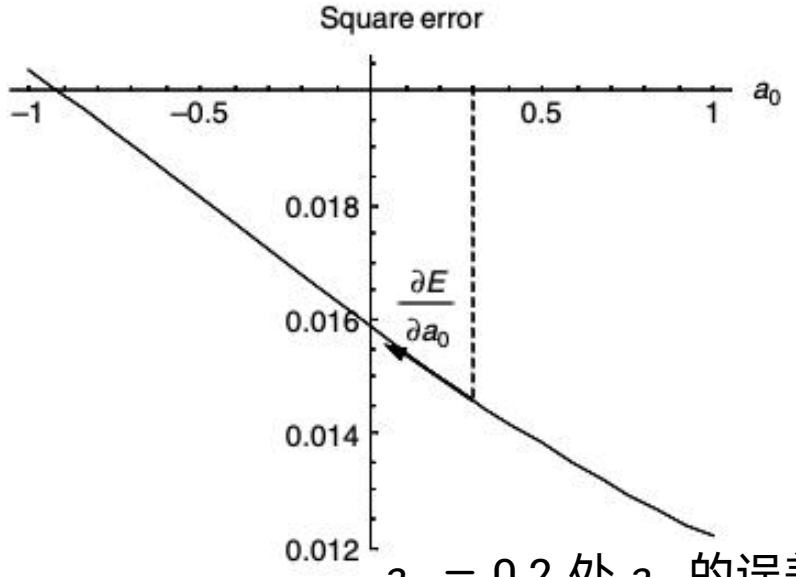
$$\frac{\partial E}{\partial a_1} = pb_1 y(1-y)x = qx$$

$$\frac{\partial E}{\partial a_0} = pb_1 y(1-y) = q$$





$a_0 = 0.3$ 处 a_1 的误差



$a_1 = 0.2$ 处 a_0 的误差

3.5 BP算法改进

BP算法缺点

- 每次学习后对当前实例的误差最小，可能导致网络震荡或不稳定
- 不同样例的更新的效果可能“抵消”
- BP网络接受样本的顺序对训练结果有较大影响，它更“偏爱”较后出现的样本
- 给样本安排一个适当的顺序，是非常困难的

BP算法改进-累积BP算法

- 在整个训练集上的全局误差以一个平均水平逐渐下降。

- 存储全部实例梯度并计算平均梯度，误差在这个梯度方向最小。

累积BP算法

用$(\boldsymbol{X}_1, \boldsymbol{Y}_1), (\boldsymbol{X}_2, \boldsymbol{Y}_2), \ldots, (\boldsymbol{X}_s, \boldsymbol{Y}_s)$的“总效果”修改$W^{(1)}, W^{(2)}, \ldots, W^{(L)}$

$$\Delta w_{ij}^{(k)} = \sum_p \Delta w_{ij}^{(k)}$$

针对累积误差最小化，读取整个训练集D后，才对参数进行更新

降低了参数更新的频率

累积BP算法

- 消除样本顺序影响的BP算法

Algorithm 2 消除样本顺序影响的BP算法

1: **for** k = 1 to L **do**
2: 初始化W^k
3: **end for**
4: 初始化精度控制参数ϵ
5: $E = \epsilon + 1$
6: **while** $E > \epsilon$ **do**
7: $E = 0$
8: 对所有的i,j,k: $\triangle w_{ij}^k = 0$

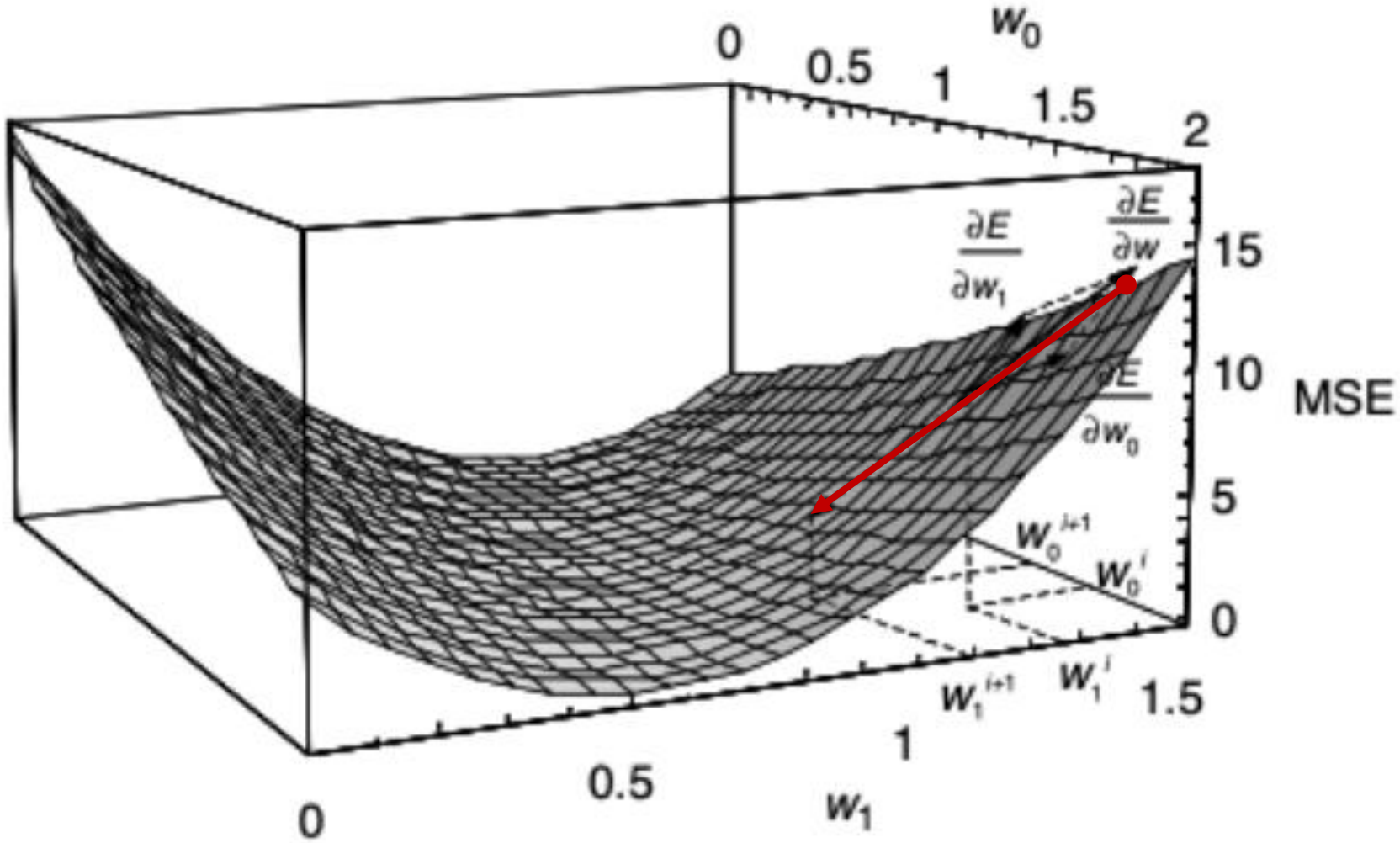
累积BP算法

- 对每个样本（X_p, Y_p）进行的操作

9: **while** 对S中每一个样本(X_p,Y_p) **do**
10: 计算出X_p对应的实际输出O_p;
11: 计算出E_p;
12: $E = E + E_p$;
13: 对所有的i,j根据相应式子计算$\triangle_p w_{ij}^{L}$;
14: 对所有的i,j：$\triangle w_{ij}^{L} = \triangle w_{ij}^{L} + \triangle_p w_{ij}^{L}$
15: $k = L - 1$;
16: **while** $k \neq 0$ **do**
17: 对所有i,j根据相应式子计算$\triangle_p w_{ij}^{k}$;
18: 对所有i,j：$\triangle w_{ij}^{k} = \triangle w_{ij}^{k} + \triangle_p w_{ij}^{k}$
19: $k = k - 1$.
20: **end while**
21: 对所有i,j,k：$w_{ij}^{k} = w_{ij}^{k} + \triangle w_{ij}^{k}$;
22: **end while**
23: $E = E/2.0$
24: **end while**

累积BP算法

两个权值情况下的误差变化



累积BP算法

累积BP算法分析

- 较好地解决了因样本的顺序引起的精度问题和训练的抖动问题
- 收敛速度：比较慢

解决方案

- 偏移量：给每一个神经元增加一个偏移量来加快收敛速度
- 动量：联接权的本次修改要考虑上次修改的影响，以减少抖动问题

BP算法改进-动量法

- 物理中，一个物体动量是指该物体在它运动方向上保持运动的趋势，是物体质量和速度的乘积。

- 在梯度下降过程中，动量法用之前积累动量代替真正的梯度。

每次迭代的梯度可以看作是加速度。

- 动量法是一种平均方法，有助于提升梯度下降法在寻找最优解时的稳定性。

动量法

$$\Delta w_m = \mu \Delta w_{m-1} - (1-\mu)\varepsilon d_m^w$$

Δw_m 是权值变化， μ是0和1之间的动量参数， d_m^w是对权值w的当前全导。

$\mu \Delta w_{m-1}$ 表示过去权值对当前权值影响

$(1-\mu)\varepsilon d_m^w$ 表示m训练时间的权值变化

动量法

动量法可以稳定学习过程

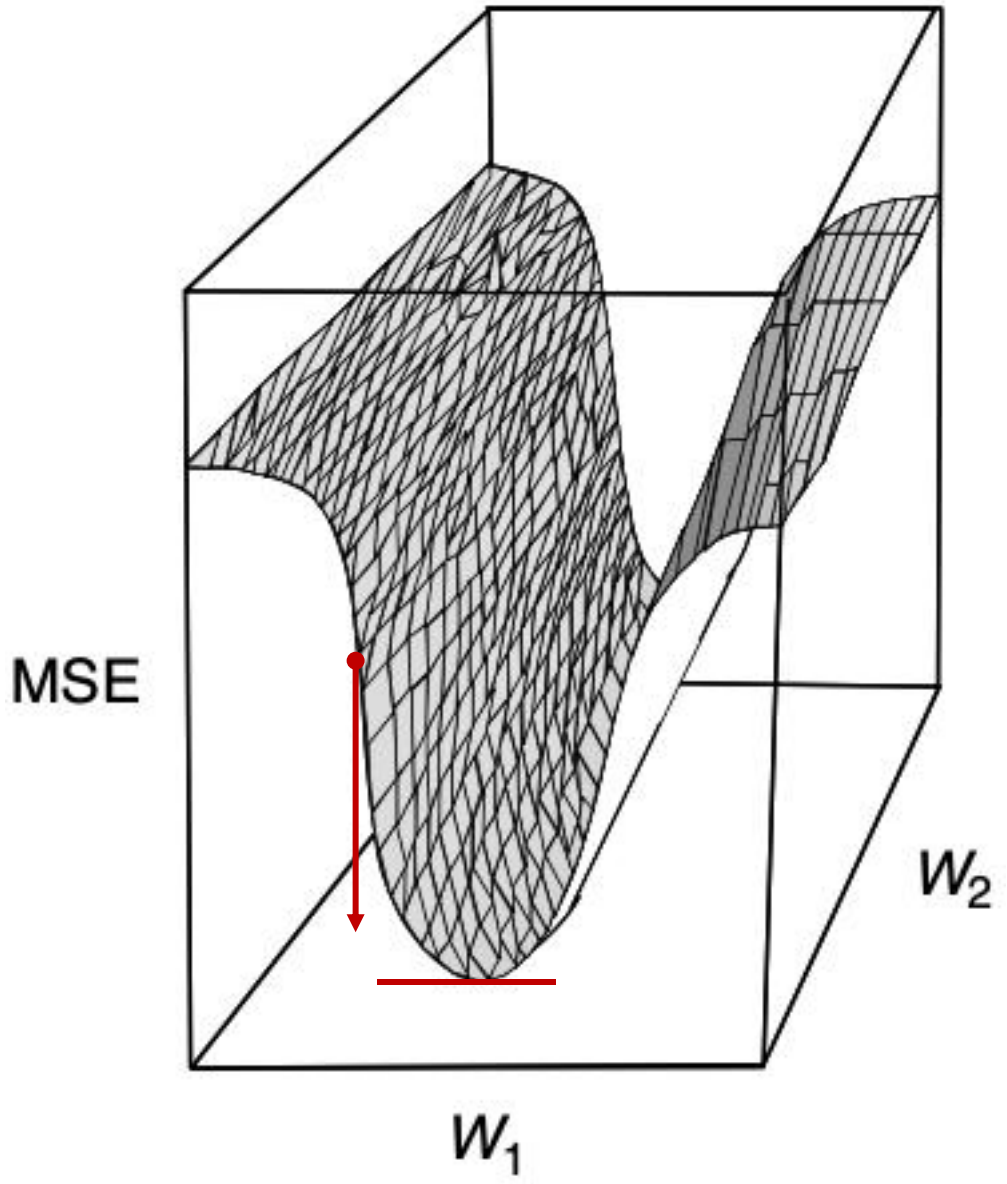
- 如果以前累积的变化与当前方向一致：加速当前权值改变
- 如果以前累积的变化与当前方向相反：阻止当前权值改变

动量法

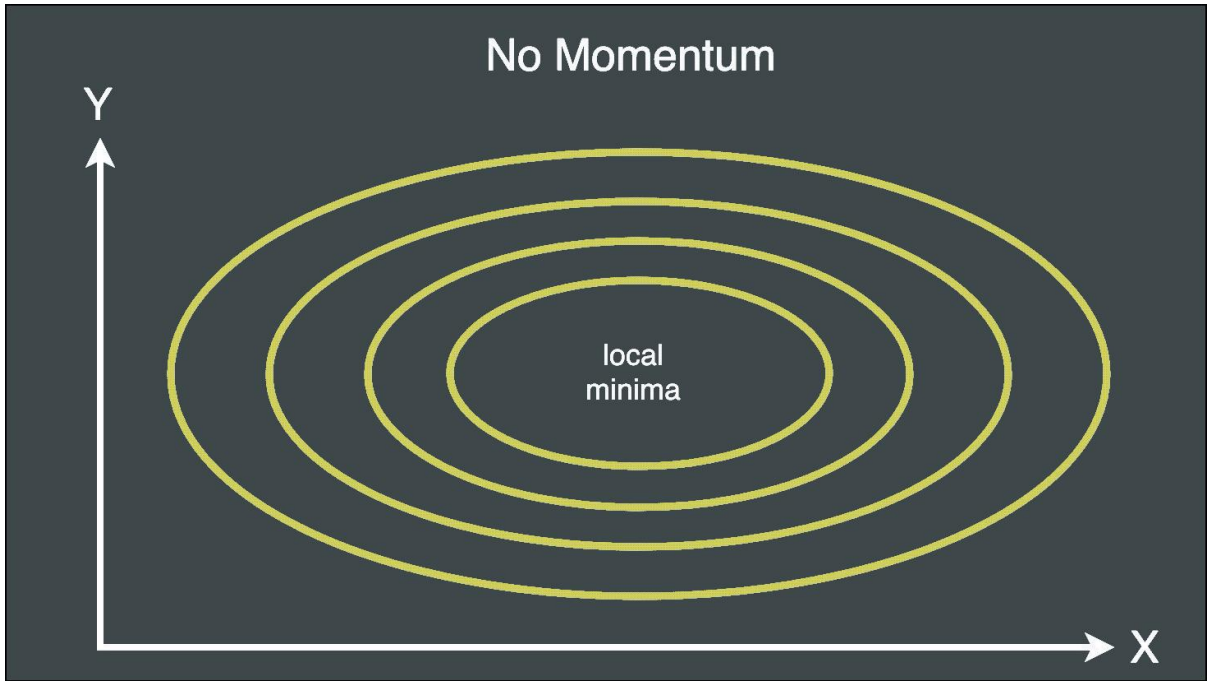
动量法可以稳定学习过程

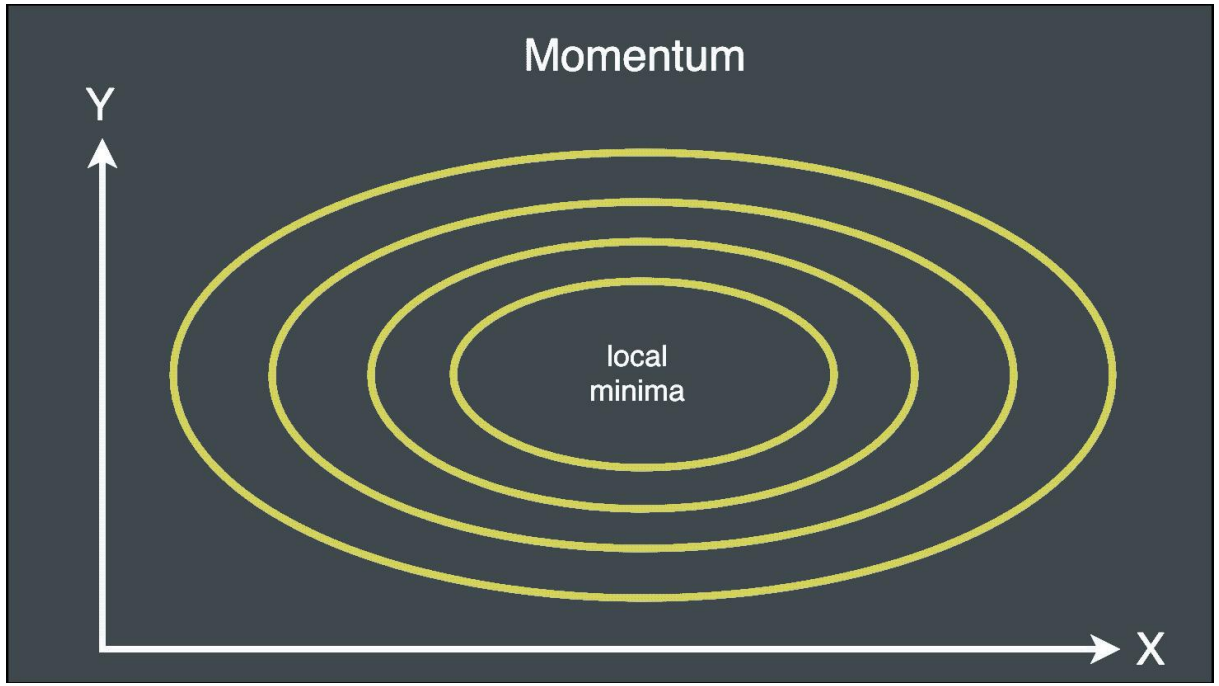
- 当位于谷底一半时，最快下降方向指向谷底，与最优解方向垂直。搜索路径在谷底附近震荡。

- 动量法持续沿陡坡寻找，帮助克服震荡。



动量法

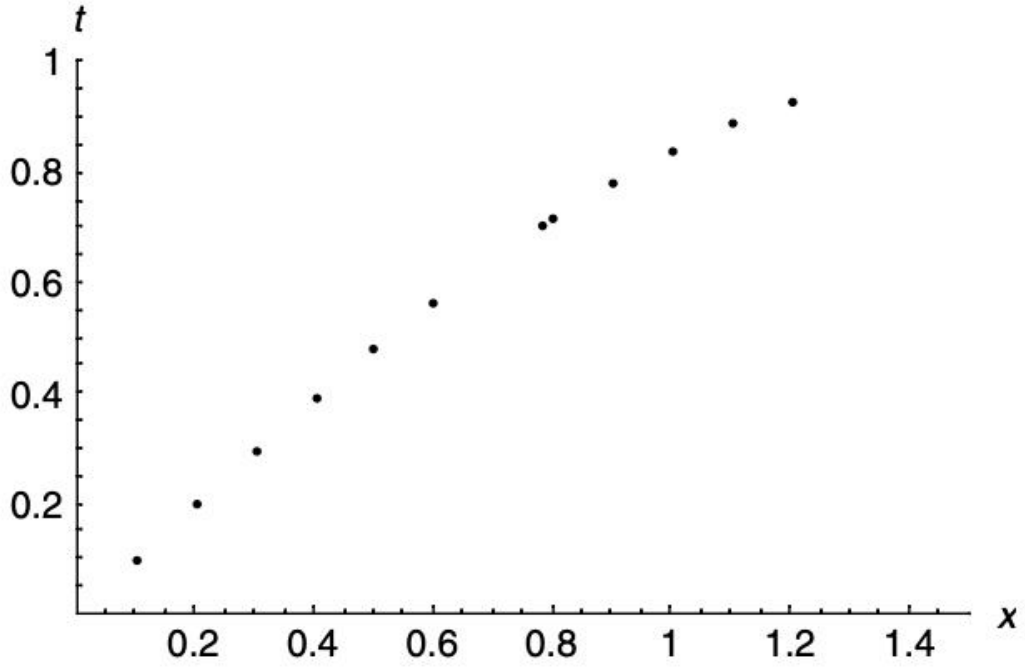




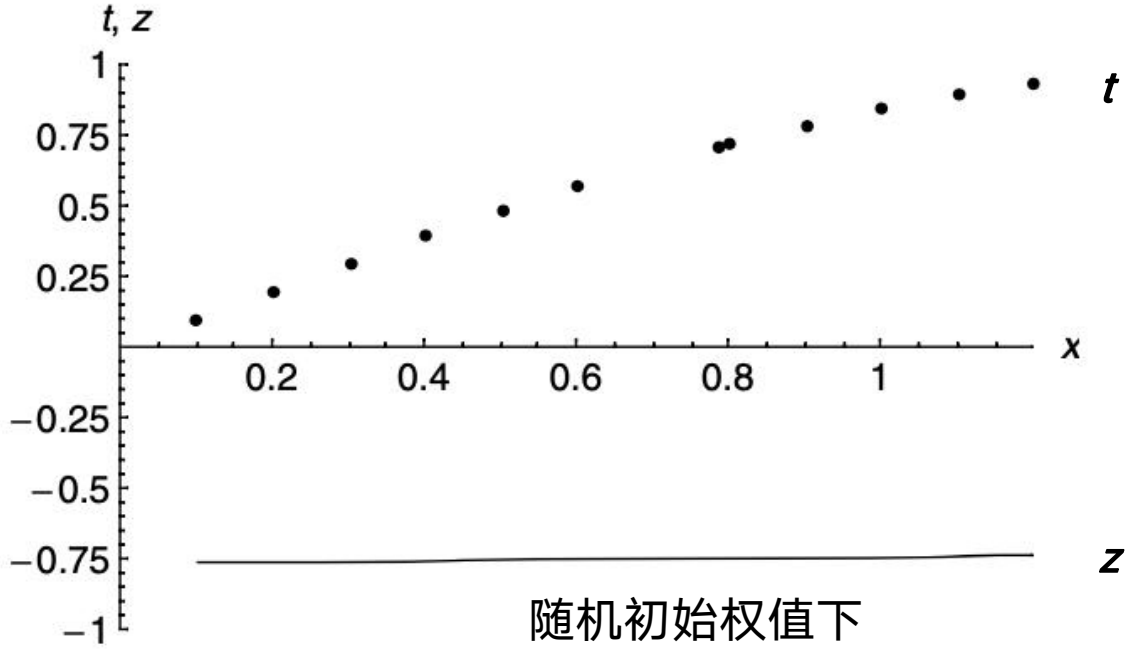
学习率和动量影响

例：一个输入，一个输出，一个隐藏神经元的网络

x：输入； *t*：实际值；*z*：预测值

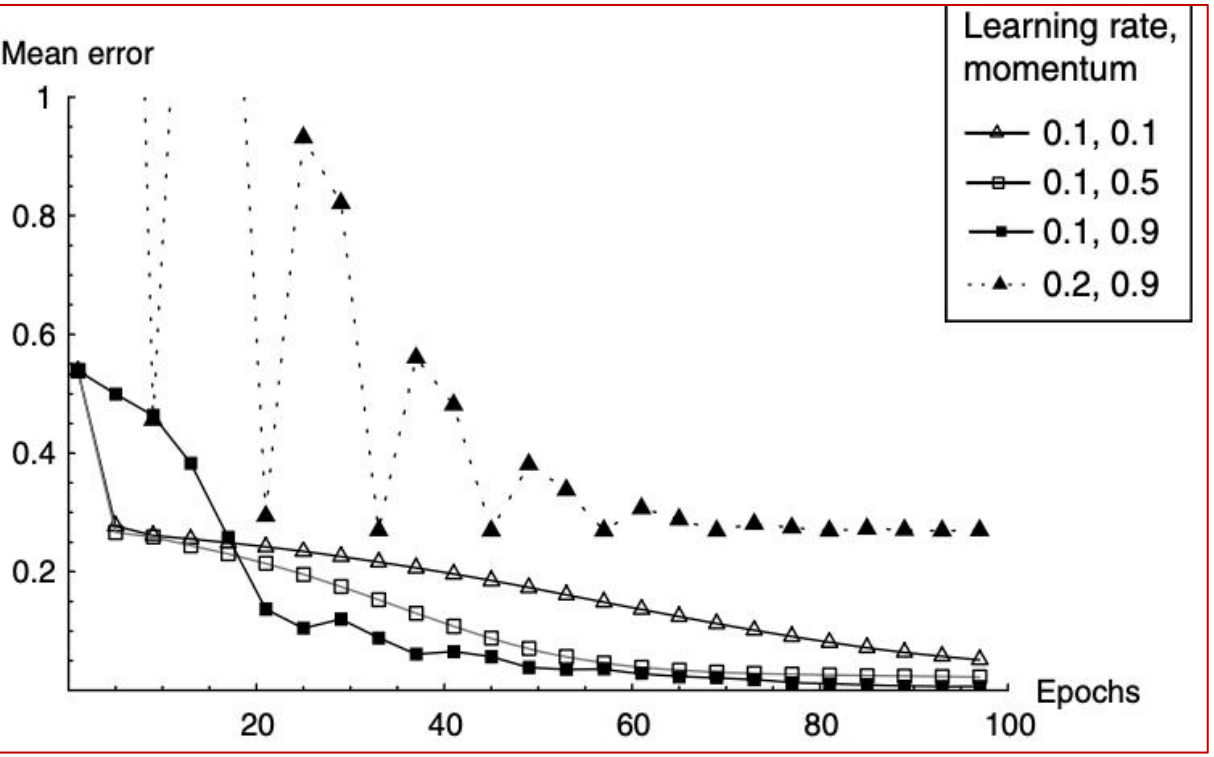


训练数据

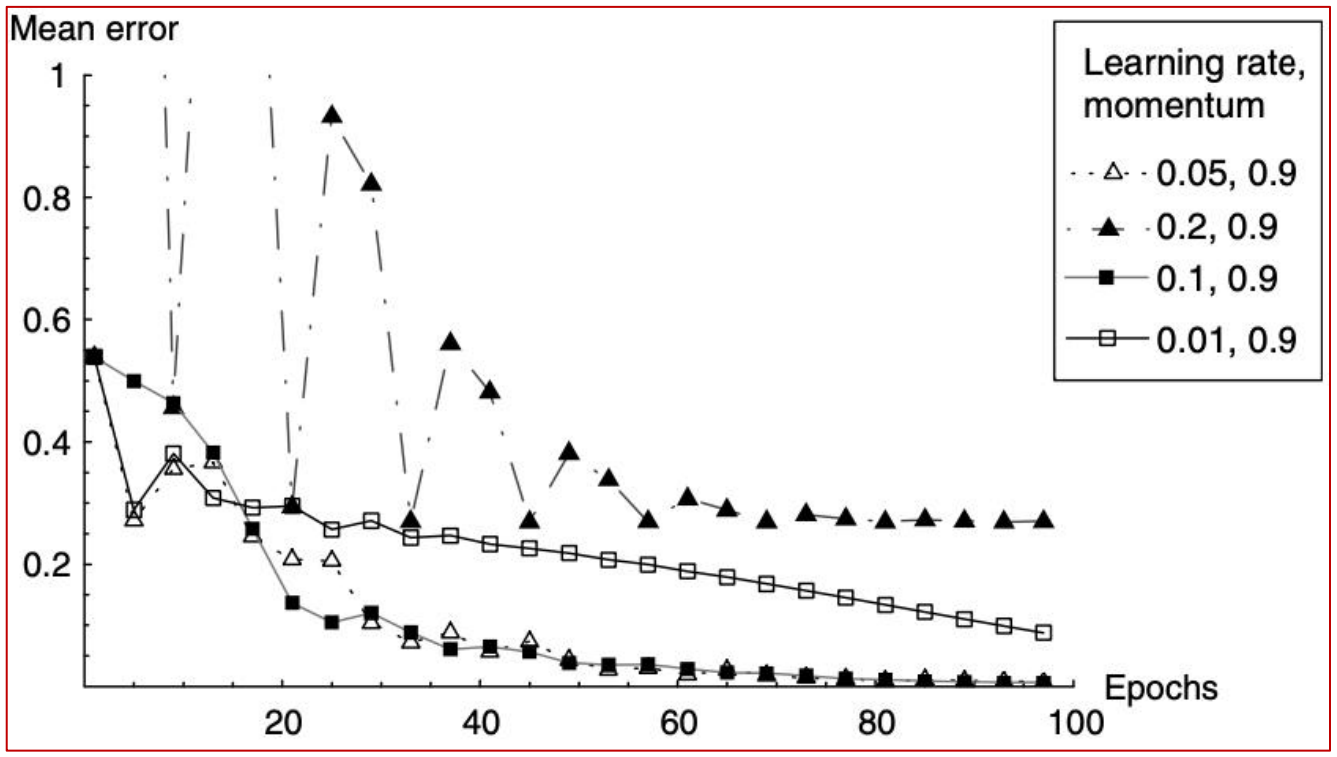


随机初始权值下
预测数据

学习率和动量影响



动量对学习影响



学习率对学习影响

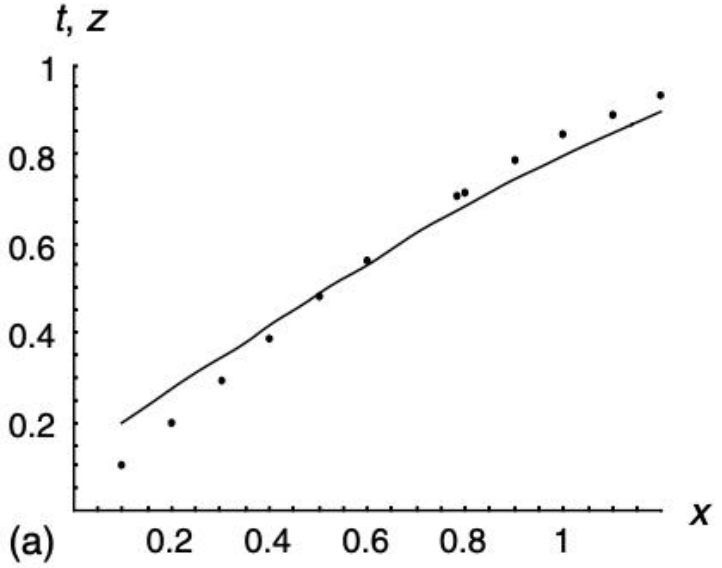
学习率和动量影响

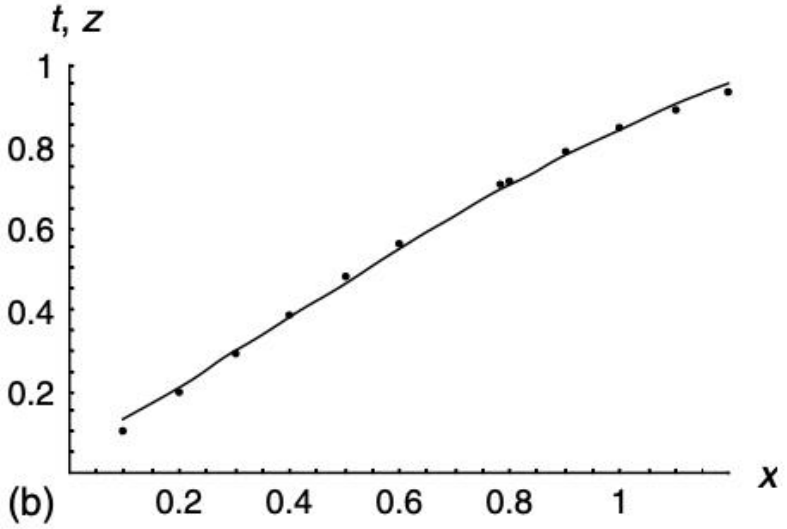
学习率为0.1时网络预测数据

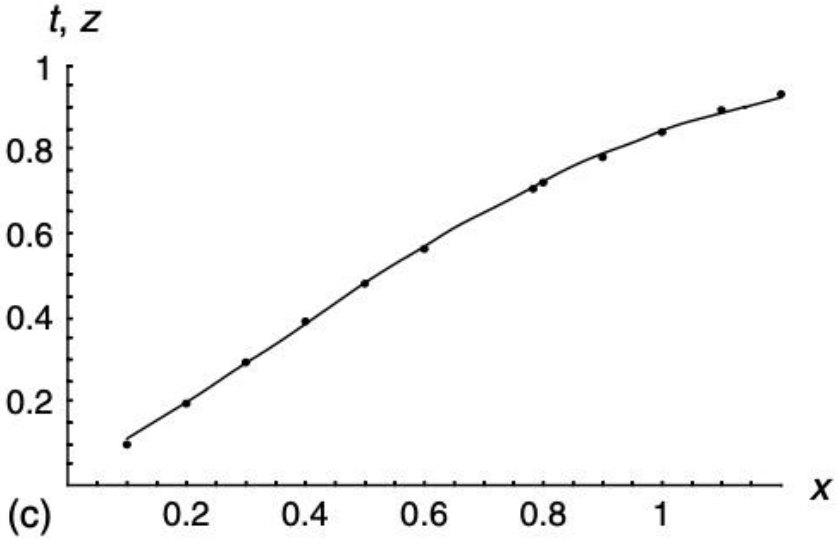
（a）动量为0.1

（b）动量为0.5

（c）动量为0.9







3.6 RBF网络

基本原理

梯度下降法

反向传播算法（BP算法）

BP算法分析

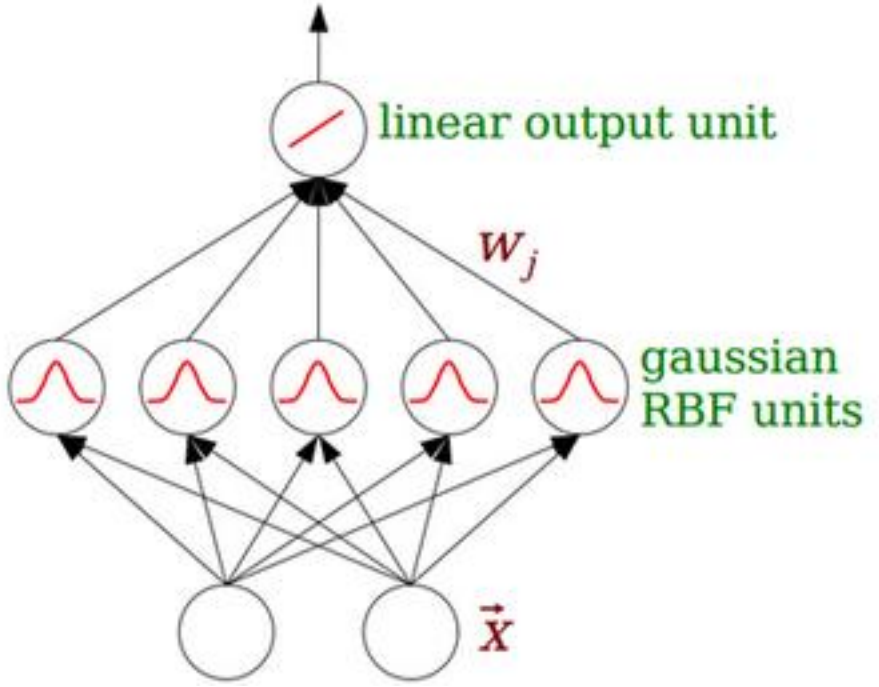
BP算法改进

RBF网络

RBF网络

径向基函数是某种沿径向对称的标量函数。通常定义为空间中任一点x到某一中心c之间欧氏距离的单调函数，可记作k(||x-c||)，其作用往往是局部的，即当x远离c时函数取值很小。

径向基函数RBF神经网络是一种三层神经网络，其包括输入层、隐层、输出层。从输入空间到隐层空间的变换是非线性的，而从隐层空间到输出层空间变换是线性的。



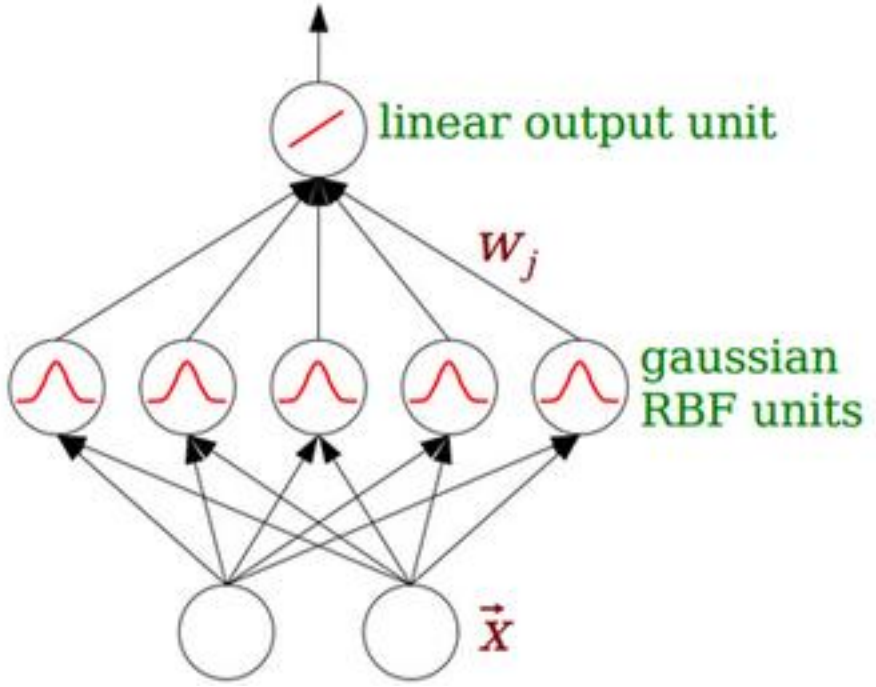
RBF网络

激活函数：

$$R(x_p - c_i) = exp(-\frac{1}{2\sigma^2}\|x_p - c_i\|^2)$$

网络输出：

$$y_j = \sum_{i=1}^{h} w_{ij} exp(-\frac{1}{2\sigma^2}\|x_p - c_i\|^2)\ j = 1, 2, \cdots, n$$



RBF网络的学习

求解的参数有3个：基函数的中心、方差以及隐含层到输出层的权值。

1）自组织选取中心学习方法：

- 无监督学习过程，求解隐含层基函数的中心与方差
- 有监督学习过程，求解隐含层到输出层之间的权值
- 首先，选取h个中心做k-means聚类，对于高斯核函数的径向基，方差由如下公式求解，c_{max}为所选取中心点之间的最大距离。

$$\sigma_i = \frac{c_{max}}{\sqrt{2h}} \quad i = 1, 2, \cdots, h$$

隐含层至输出层之间的神经元的连接权值可以用最小二乘法直接计算得到，即对损失函数求解关于w的偏导数，使其等于0，可以化简得到计算公式为：

$$w = exp(\frac{h}{c_{max}^2}\|x_p - c_i\|^2) \quad p = 1, 2, \cdots, P; i = 1, 2, \cdots, h$$

RBF网络的学习

2）直接计算法

- 隐含层神经元的中心是随机地在输入样本中选取，且中心固定。一旦中心固定下来，隐含层神经元的输出便是已知的，这样的神经网络的连接权就可以通过求解线性方程组来确定。适用于样本数据的分布具有明显代表性。

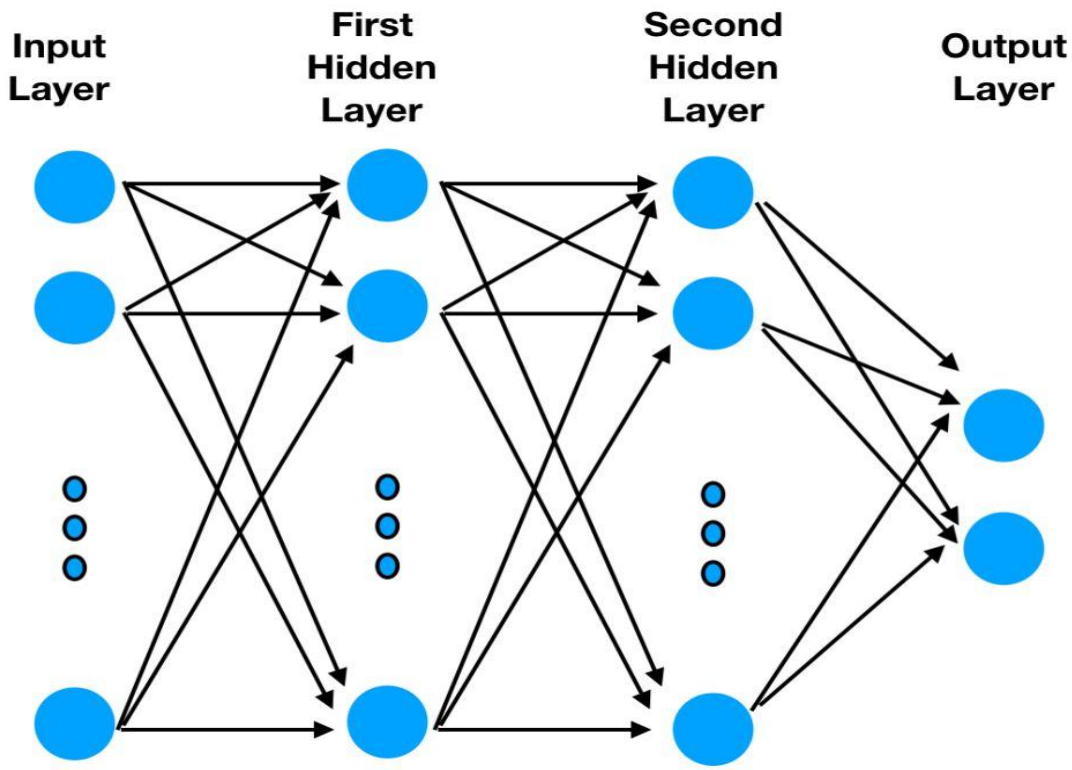
3）有监督学习算法

- 通过训练样本集来获得满足监督要求的网络中心和其他权重参数，经历一个误差修正学习的过程，与BP网络的学习原理一样，同样采用梯度下降法。

04 多层感知机深入分析

多层感知机

多层感知机是至少含有一个**隐藏层**的全连接神经网络，且每个隐藏层的输出通过**激活函数**进行变换。



4.1 激活函数

激活函数

通用近似定理

网络结构超参数选择

激活函数

- 以单隐藏层为例，**没有激活函数的情况下**，多层感知机按以下方式计算输出：

$$H = XW_h + b_h$$

$$O = HW_o + b_o$$

- 将以上两个式子联立起来，可以得到

$$O = (XW_h + b_h)W_o + b_o = XW_hW_o + b_hW_o + b_o$$

- 这里虽然神经网络引入了隐藏层，却依然等价于一个单层神经网络：其中输出层权重参数为 W_hW_o ，偏差参数为 $b_hW_o + b_o$ 。即便再添加更多的隐藏层，以上设计依然只能与仅含输出层的单层神经网络等价。

激活函数

- **加入激活函数后**，多层感知机按以下方式计算输出，其中 ø 表示激活函数：

$$H = \text{ø}(XW_h + b_h)$$

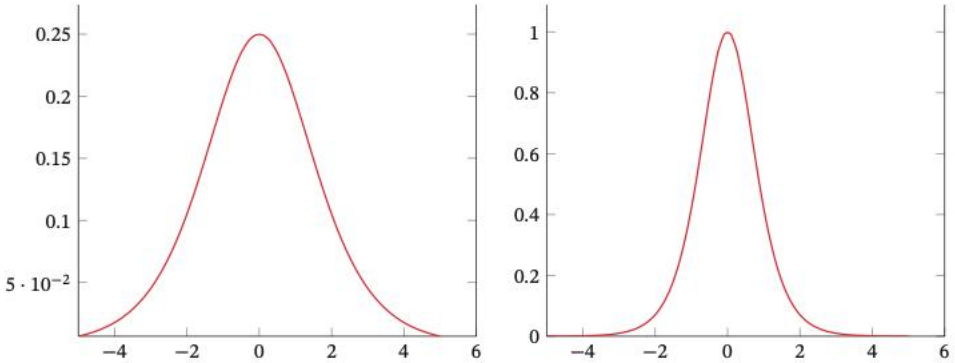
$$O = HW_o + b_o$$

- 激活函数引入非线性变换，对变量实现非线性映射，然后再作为下一层的输入。常见的激活函数如下所示：

激活函数	函数	导数
Logistic 函数	$f(x) = \frac{1}{1+\exp(-x)}$	$f'(x) = f(x)\big(1 - f(x)\big)$
Tanh 函数	$f(x) = \frac{\exp(x)-\exp(-x)}{\exp(x)+\exp(-x)}$	$f'(x) = 1 - f(x)^2$
ReLU 函数	$f(x) = \max(0, x)$	$f'(x) = I(x > 0)$

激活函数

- **Logistic(Sigmoid)函数**和**Tanh函数**的导数的值域都小于或等于 1。



(a) Logistic 函数的导数　　(b) Tanh 函数的导数

- 误差从输出层反向传播时，在每一层都要乘以该层的激活函数的导数。
- 这样，误差经过每一层传递都会不断衰减。当网络层数很深时，梯度就会不停衰减，甚至消失，使得整个网络很难训练，即梯度消失问题(Vanishing Gradient Problem)
- 一种减轻梯度消失问题的方法就是使用导数较大的激活函数，比如ReLU函数。

激活函数

- **ReLU函数**

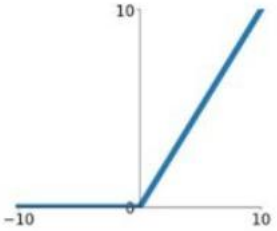
➢缓解梯度消失问题

➢计算速度非常快，只需要判断输入是否大于0

➢收敛速度远快于Sigmoid和Tanh

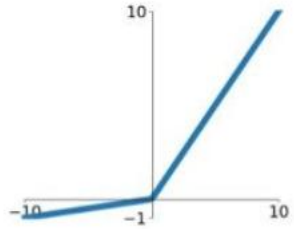
ReLU

$$\max(0, x)$$



Leaky ReLU

$$\max(0.1x, x)$$



- Dead ReLU Problem：采 用 ReLU 作为激活函数的神经元在训练时比较容易“死亡”。

➢在训练时，如果参数在一次不恰当的更新后，隐藏层中的某个神经元在所有的训练数据上都不能被激活，那么这个神经元自身参数的梯度永远都会是 0，在以后的训练过程中也永远不能被激活。

- 为了避免上述情况，有几种 ReLU 的变种（如**Leaky ReLU**等）也会被广泛使用。

激活函数影响分析

●实验对比：以手写数据集为例

其他参数相同，分别使用Sigmoid、Tanh、ReLU作为激活函数，对比实验效果为：

激活函数	Sigmoid	Tanh	ReLU
错误数量	41	39	45
准确率	95.67%	95.88%	95.24%

- 这个实验中Tanh的效果较好，但也只能说明在当前实验环境和参数设置下Tanh的效果相对较好，不能一概而论；
- 由于梯度消失问题，有时要谨慎使用Sigmoid和Tanh函数；
- 目前ReLU函数的使用相对最为频繁，实验中一般可以从ReLU函数开始，如果ReLU函数没有提供较好结果，再尝试其他激活函数；

4.2 通用近似定理

激活函数

通用近似定理

网络结构超参数选择

通用近似定理

令$\phi(\cdot)$是一个非常数、有界、单调递增的的连续函数，I_D是一个D维的单位超立方体$[0,1]^D$，$C(\mathrm{I}_D)$是定义在I_D上的连续函数的集合，对于任意给定的一个函数$f \in C(\mathrm{I}_D)$，存在一个整数M，和一组实数$v_m, b_m \in \mathbb{R}$以及实数向量$\omega_m \in \mathbb{R}^D, m = 1, \ldots, M$，定义函数：

$$F(x) = \sum_{m=1}^{M} v_m \phi(\omega_m^{\mathrm{T}} x + b_m)$$

作为函数f的近似实现，即

$$|F(x) - f(x)| < \epsilon, \ \forall x \in \mathrm{I}_D$$

ϵ是大于0的小正数。

通用近似定理

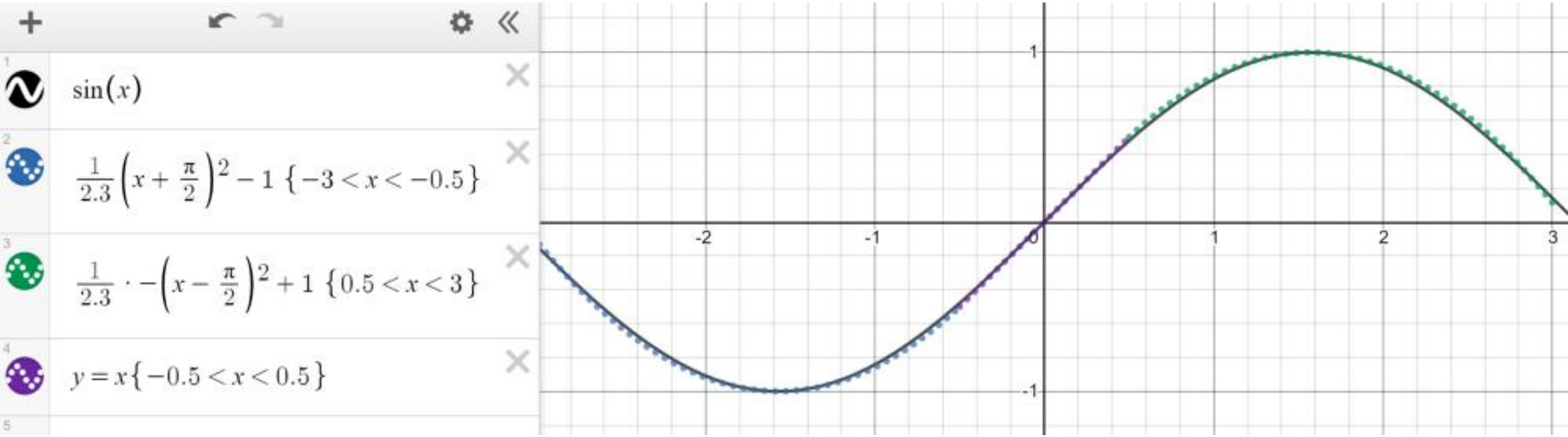
- 通用近似定理 (Universal approximation theorem)：如果一个神经网络具有线性输出层和至少一层隐藏层，只要给予网络足够数量的神经元，便可以实现以足够高精度来逼近任意一个在 $\mathbb{R}n$ 的紧子集 (Compact subset) 上的连续函数。

- George Cybenko在 1989 年最早提出这一定理，并证明在激活函数为 Sigmoid 函数的情况下的准确性。这一定理被看作是 Sigmoid 函数所具有特殊性质。
- Kurt Hornik 在后续研究中发现，造就**通用拟合**这一特性的根本原因并非 Sigmoid 函数，而是**多层前馈神经网络结构**。

通用近似定理

紧凑（有限、封闭）集合上的任何连续函数都可以用分段函数逼近。

以-3到3之间的正弦波为例，它可以用三个函数来近似——两个二次函数和一个线性函数。

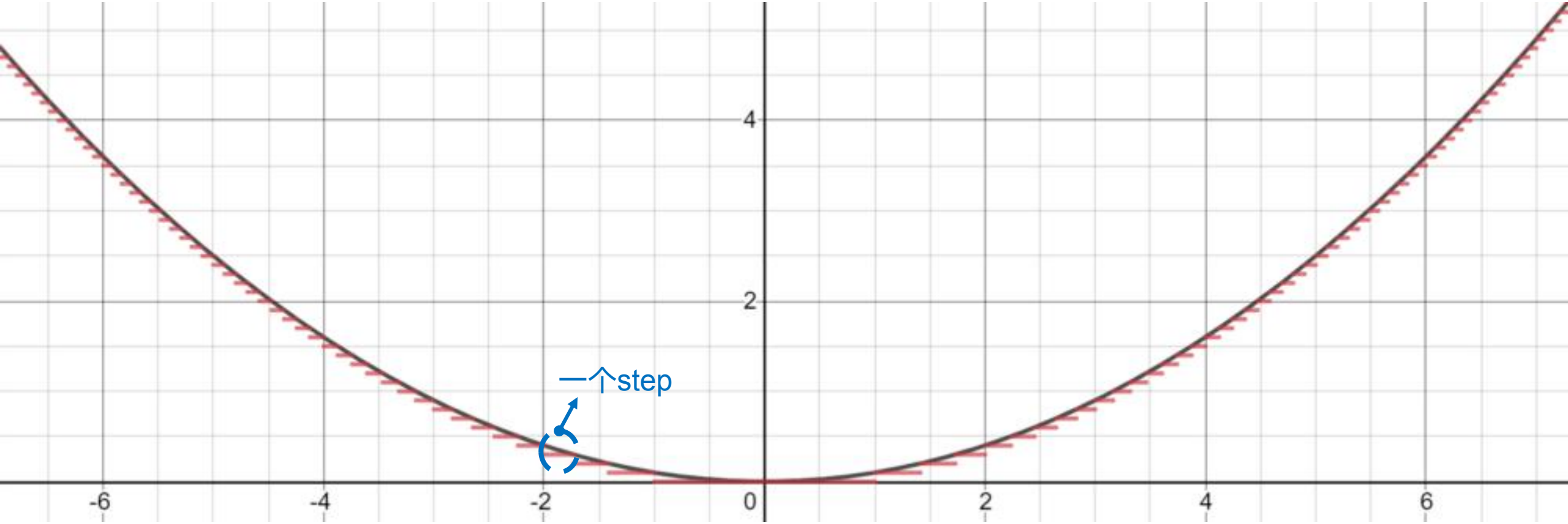


https://medium.com/analytics-vidhya/you-dont-understand-neural-networks-until-you-understand-the-universal-approximation-theorem-85b3e7677126

通用近似定理

分段函数每段可以是恒定的，即每个分段函数由一些恒定区域的‘step’组成。

只要有足够多的step，就可以在给定的范围内合理估计函数。



通用近似定理

基于这种近似，我们可以将神经元当做step来构建网络。

利用权值和偏差作为「门」来确定输入进入时哪个神经元应该被激活，一个有足够多数量神经元的神经网络可以简单地将一个函数划分为多个恒定区域来估计。

4.3 网络结构超参数选择

激活函数

通用近似定理

网络结构超参数选择

网络结构超参数选择

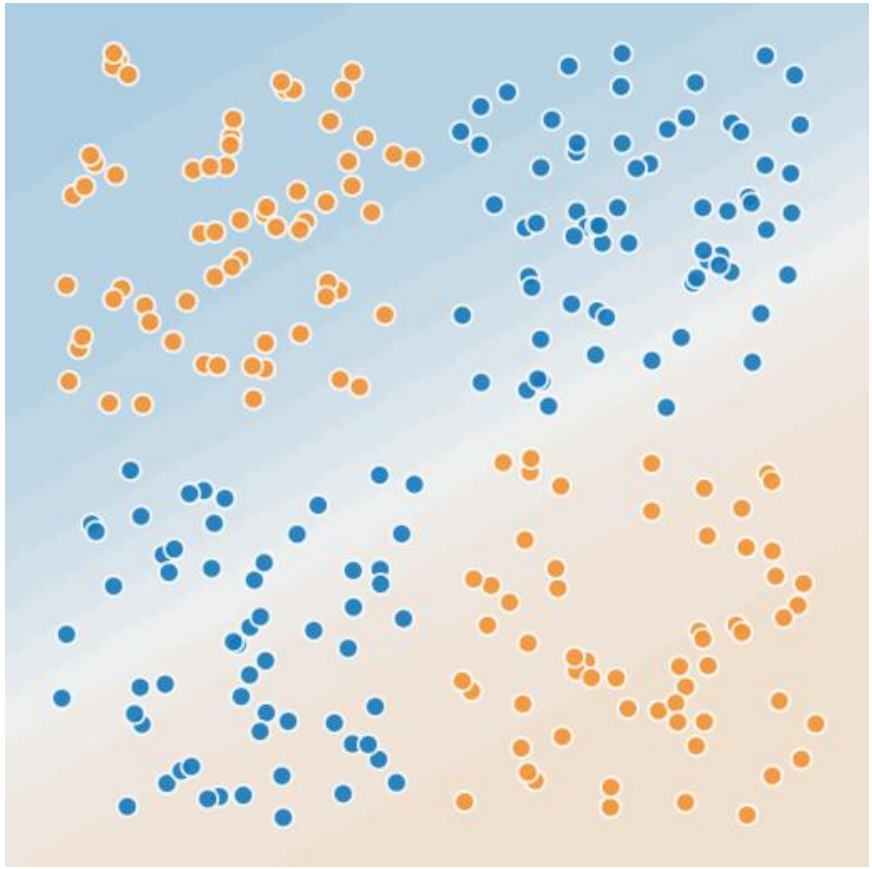
- **网络结构超参数**：输入\输出向量的维数、隐藏层的层数以及隐藏层神经元的个数
- 一般来说，输入层的神经元数量等于待处理数据中输入变量的数量，输出层的神经元的数量等于与每个输入关联的输出的数量。

- 困难之处在于**确定合适的隐藏层的层数以及隐藏层神经元的个数**。

- 一般情况下，更深更宽的结构可以模拟更复杂的分布，但增加隐藏层的层数和隐藏层神经元个数也不一定总能够提高网络精度和表达能力。

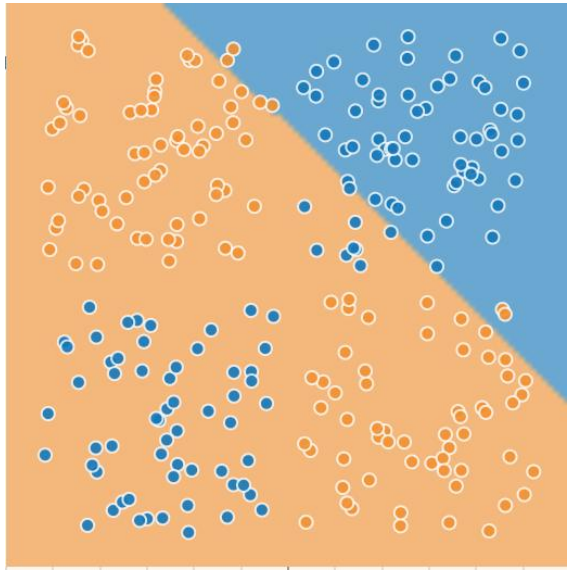
不同个数隐藏神经元拟合能力

假设单个隐藏层中分别含有1、2、3、4个神经元，其他参数相同，对下面数据集的蓝色和橙色点进行分类：



不同个数隐藏神经元拟合能力

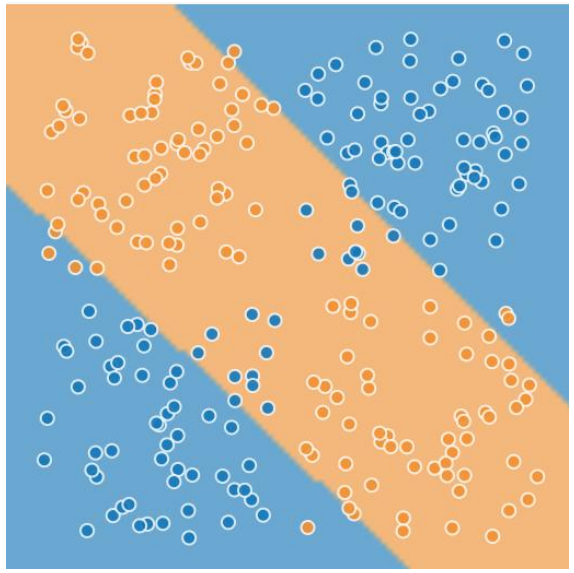
- 隐藏层中神经元个数：1、2、3、4（从左至右）



OUTPUT

Test loss 0.377

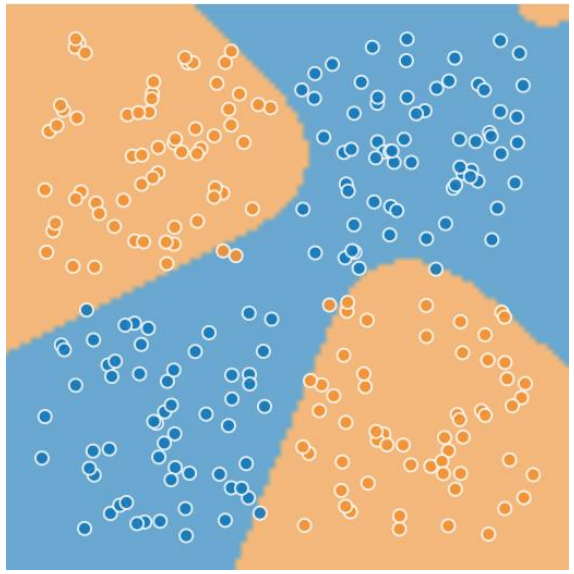
Training loss 0.374



OUTPUT

Test loss 0.166

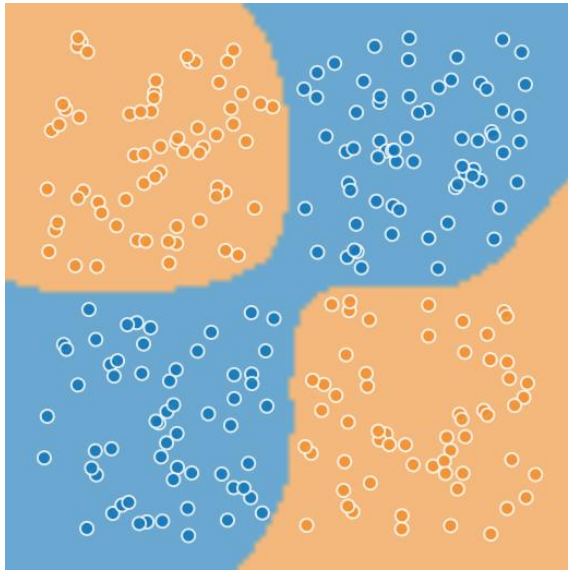
Training loss 0.169



OUTPUT

Test loss 0.050

Training loss 0.037



OUTPUT

Test loss 0.021

Training loss 0.010

隐藏层神经元个数影响分析

●实验对比：以手写数据集为例

其他参数相同，运行隐藏层神经元个数为10、50、100、200、2000的单隐藏层多层感知机，对比实验效果为：

神经元个数	10	50	100	200	2000
错误数量	181	44	41	40	45
准确率	80.87%	95.35%	95.67%	95.77%	95.24%

- 一般来说，随着隐藏层神经元个数的增加，MLP的正确率越来越高；
- 隐藏层神经元个数增加到一定数量后，训练难度增大但对准确率的提升变得很小，造成计算负担与结果提升不对等的现象；
- 如果隐藏层神经元个数过多，出现过拟合，反而会使测试集准确率下降；

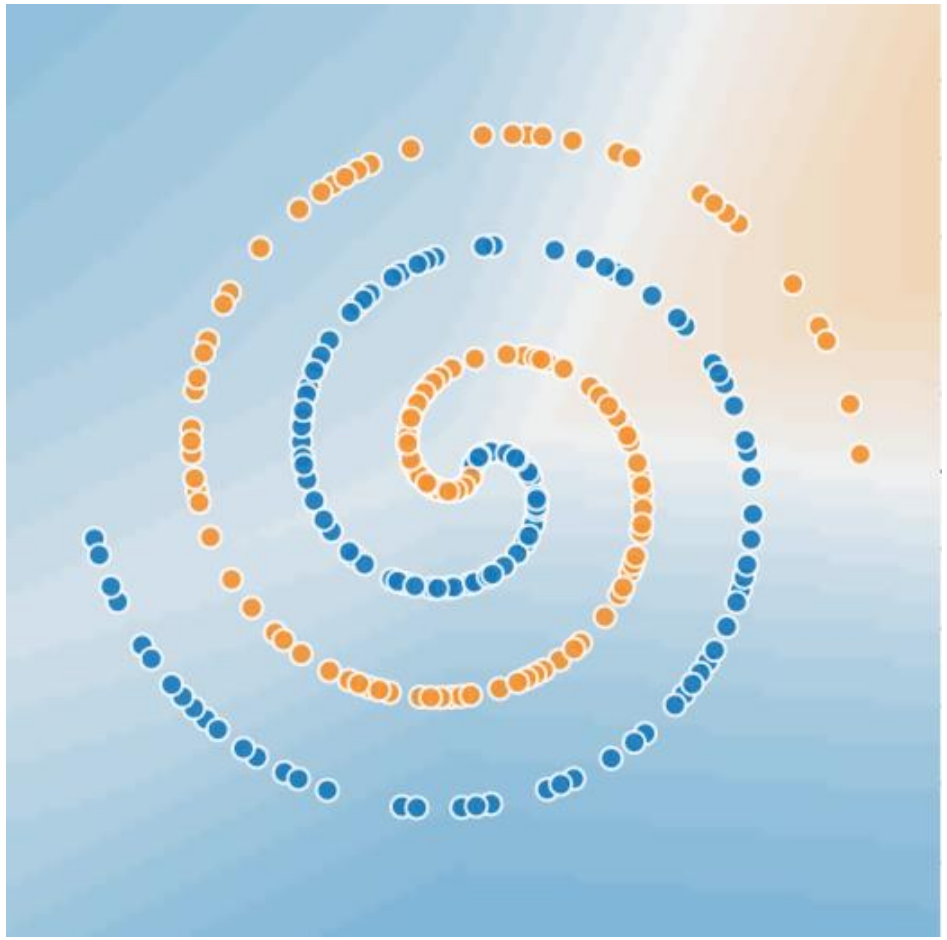
不同个数隐藏神经元

隐藏神经元的作用是从样本中提取并存储其内在规律，每个神经元有若干个权值，而每个权值都是增强网络映射能力的一个参数。

- 在隐藏层中使用太少的神经元，网络从样本中获取的信息能力较差，不足以概况和体现训练集中的样本规律，导致**欠拟合**(underfitting)。
- 使用过多的神经元也存在问题：
  - 首先，隐藏层中的神经元过多，可能学到样本中非规律性的内容（如噪声等），导致**过拟合**(overfitting)，降低泛化能力。
  - 另外，隐藏层中过多的神经元会增加训练时间，增大计算负担。
- 所以，选择一个合适的隐藏层神经元数量是至关重要的。

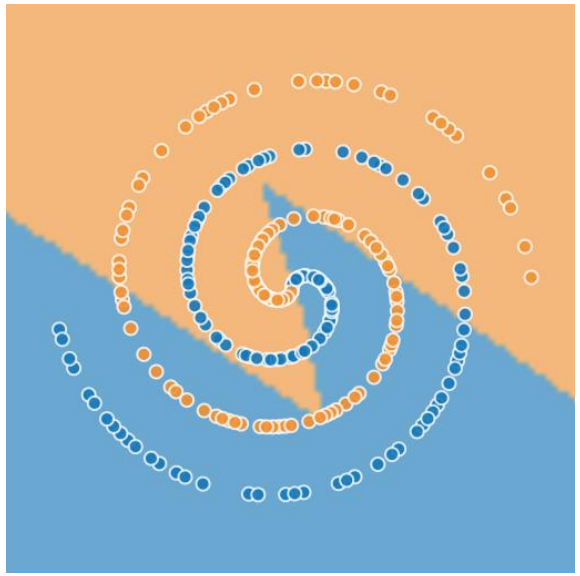
不同层数隐藏神经元拟合能力

选用1、2、3、4层隐藏层，其他参数相同，对下面数据集的蓝色和橙色点进行分类：



不同层数隐藏神经元拟合能力

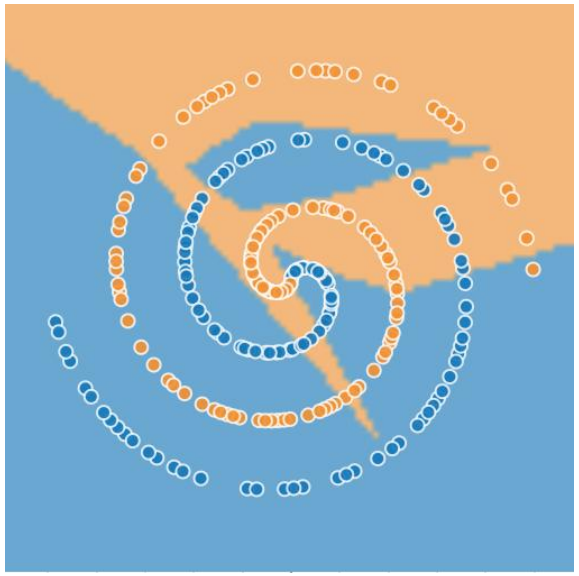
- 隐藏层数：1、2、3、4（从左至右）



OUTPUT

Test loss 0.459

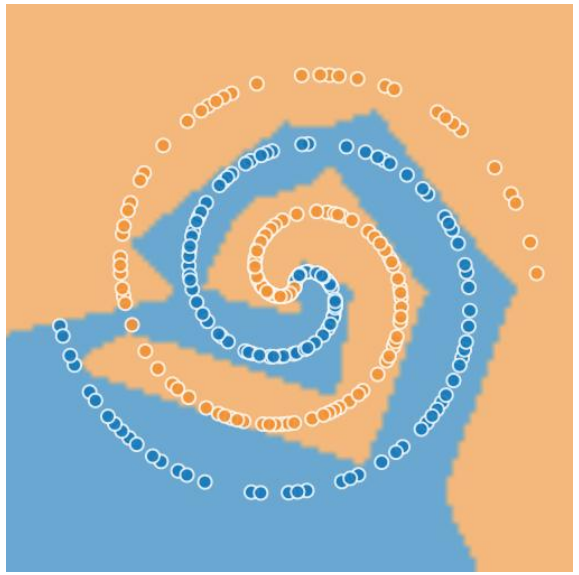
Training loss 0.454



OUTPUT

Test loss 0.340

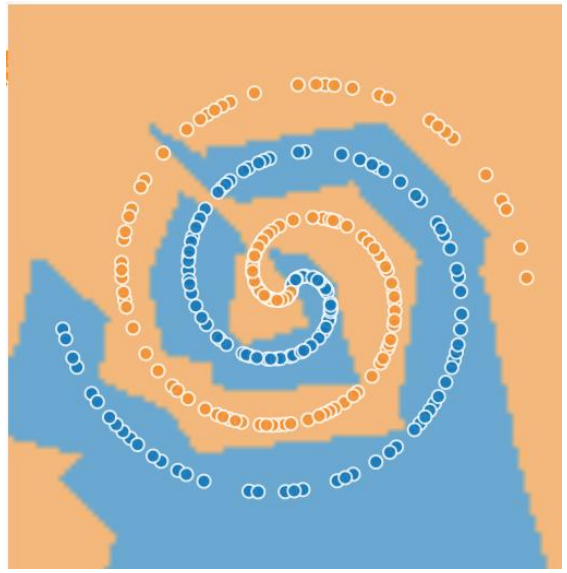
Training loss 0.345



OUTPUT

Test loss 0.052

Training loss 0.008



OUTPUT

Test loss 0.031

Training loss 0.000

隐藏层层数影响分析

●实验对比：以手写数据集为例

其他参数相同，运行隐藏层数为1、2、3、4的多层感知机，对比实验效果为：

神经元层数	1	2	3	4
错误数量	44	36	35	42
准确率	95.35%	96.19%	96.30%	95.56%

- 和隐藏层神经元个数一样，随着层数的增加，MLP的准确率通常越来越高；
- 同样，隐藏层层数增加到一定数量后，训练难度增大但对准确率的提升变小，甚至出现准确率下降的情况；

不同层数隐藏神经元

- 层数越深，理论上拟合函数的能力增强，效果一般会更好。
- 但更深的层数可能会带来过拟合的问题，同时层数越深，参数会爆炸式增长，出现梯度消失或梯度爆炸现象，增加训练难度，使模型难以收敛。
- 对于一般简单的数据集，一两层隐藏层通常就足够了。但对于涉及时间序列或计算机视觉的复杂数据集，则需要额外增加层数。
- 所以，选择一个合适的隐藏神经元层数同样至关重要。

总结

- 如何设置隐藏层神经元的数量仍是未决的问题，目前没有严格的理论指导。
- 实际应用中，隐藏层神经元的最佳数量需要自己通过不断试验调整

即“**试错法**”(trial-by-error)

- 如果欠拟合，就慢慢添加更多的层和神经元个数；
- 如果过拟合，就逐渐减小层数和神经元个数。

思考题：更深的网络和更广的网络哪一个更好？

05 多层感知机的应用

实现多层感知机的工具包

- **scikit-learn**

➢ from sklearn.neural_network import MLPClassifier # 分类

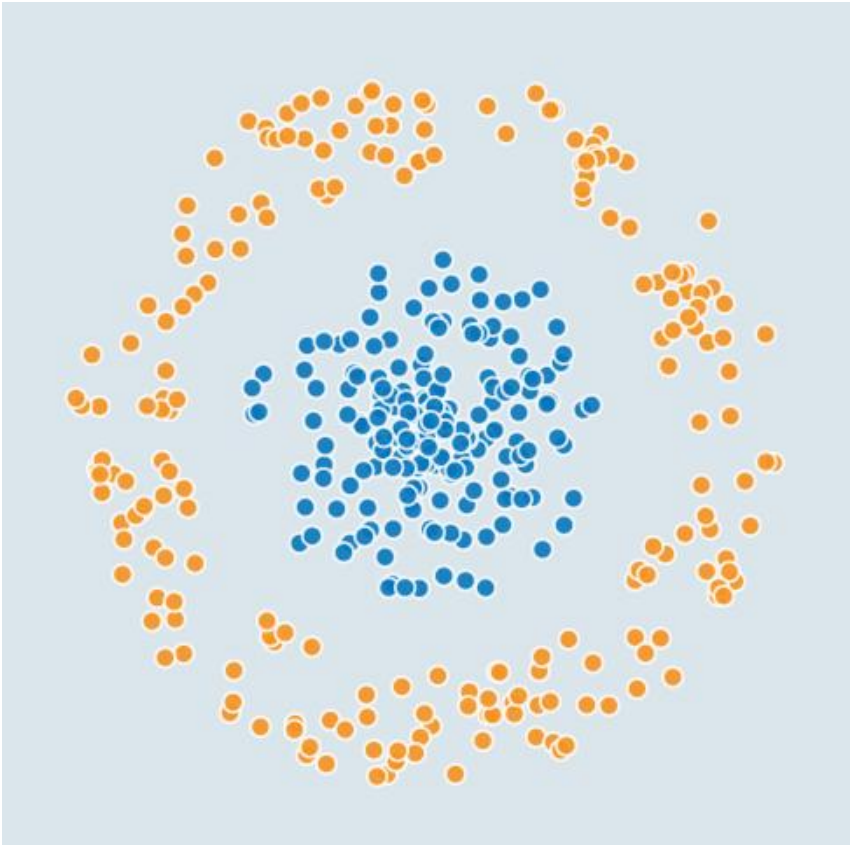
class `sklearn.neural_network.` **`MLPClassifier`**(*hidden_layer_sizes=100, activation='relu', *, solver='adam', alpha=0.0001, batch_size='auto', learning_rate='constant', learning_rate_init=0.001, power_t=0.5, max_iter=200, shuffle=True, random_state=None, tol=0.0001, verbose=False, warm_start=False, momentum=0.9, nesterovs_momentum=True, early_stopping=False, validation_fraction=0.1, beta_1=0.9, beta_2=0.999, epsilon=1e-08, n_iter_no_change=10, max_fun=15000*) [source]

➢ from sklearn.neural_network import MLPRegressor # 回归

参数与MLPClassifier类似

分类

如右图所示的二维平面有两个类别，分别为橙色类别和蓝色类别。



搭建一个二分类的多层感知机对这两种类别进行分类。

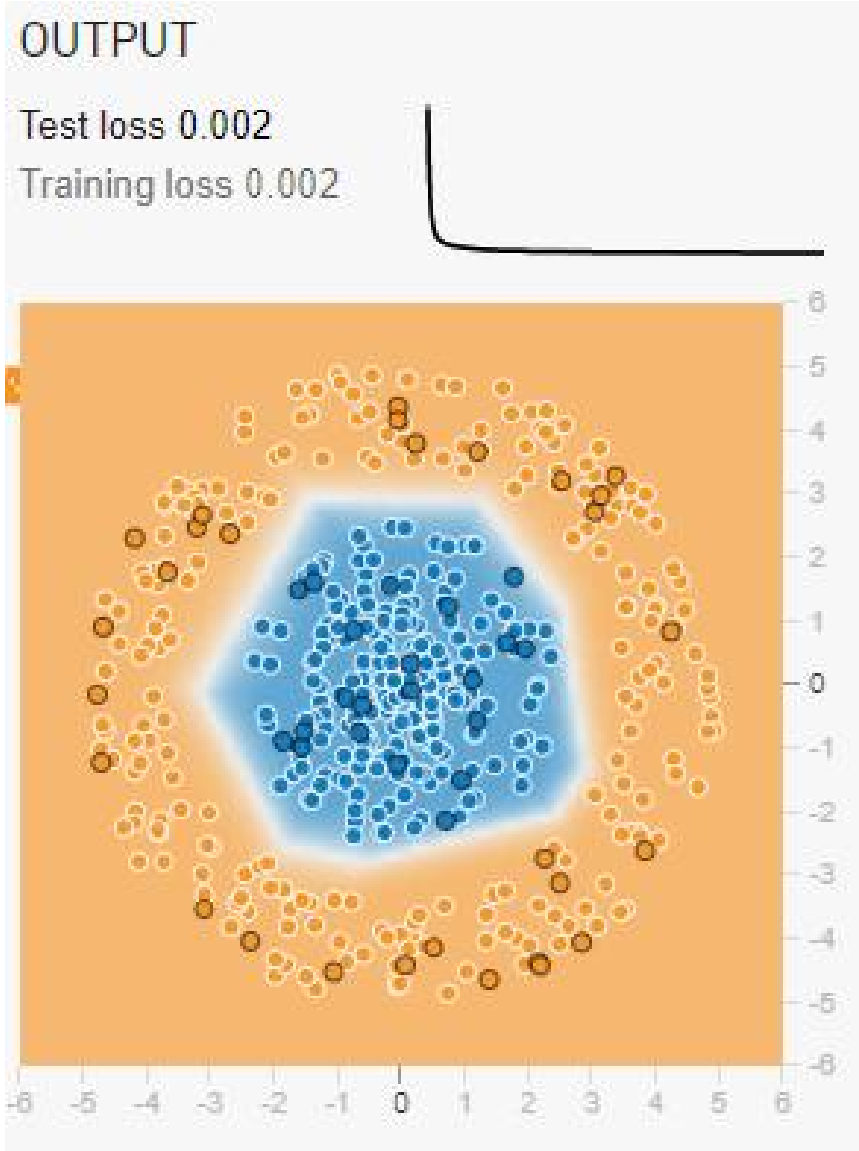
分类

搭建一个结构为 输入-隐藏层-输出 的多层感知机进行分类。

其中输出层包含两个神经元，隐藏层只有一层且只包含四个神经元，输出包含一个神经元。损失函数使用均方误差，激活函数使用ReLU。

训练结果如右图所示。可以看到，训练loss和测试loss都达到了很低的水平，同时该神经网络也学到了很好的分类边界，证明了多层感知器有很强大的非线性表示能力。

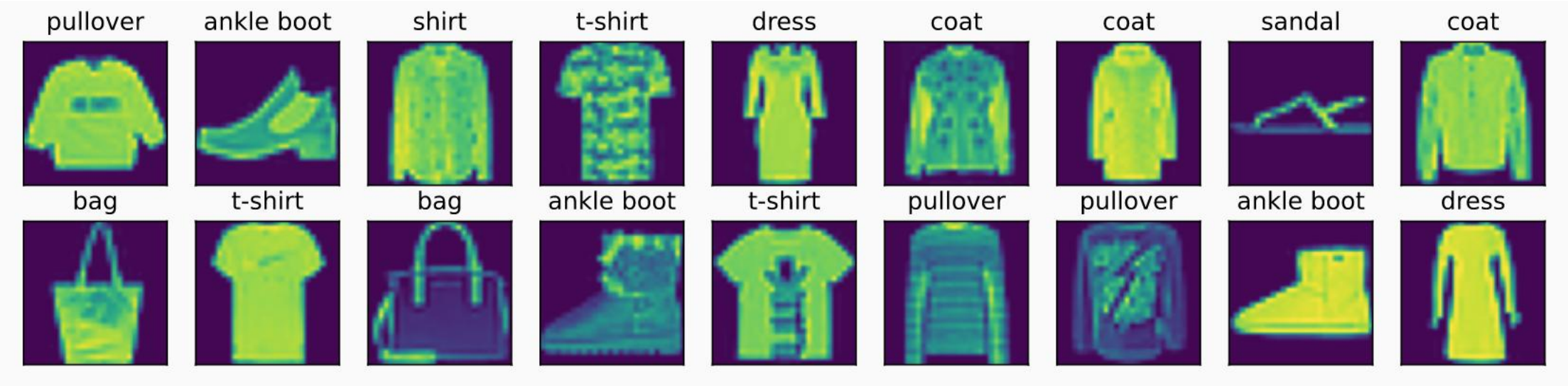
注：本实例是使用了在线机器学习平台
https://playground.tensorflow.org/



图像分类

- **示例：** Fashion-MNIST数据集分类
- 数据集介绍：

Fashion-MNIST中包含的10个类别分别为t-shirt（T恤）、trouser（裤子）、pullover（套衫）、dress（连衣裙）、coat（外套）、sandal（凉鞋）、shirt（衬衫）、sneaker（运动鞋）、bag（包）和ankle boot（短靴）



搭建多层感知机进行图像分类

我们可以通过高级API（MXNet、PyTorch、Tensorflow）简洁地实现多层感知机。

- 这里搭建一个具有单隐藏层的多层感知机进行分类：
  - 输入是Fashion-MNIST数据集中由 28×28=784 个灰度像素值组成的图像
  - 第一层是隐藏层，它包含 256 个隐藏单元，并使用了 ReLU 激活函数
  - 第二层是输出层，将图像分为 10 个类别

```
# MXNet
from mxnet.gluon import nn
net = nn.Sequential()
net.add(nn.Dense(256, activation='relu'),
        nn.Dense(10))
```

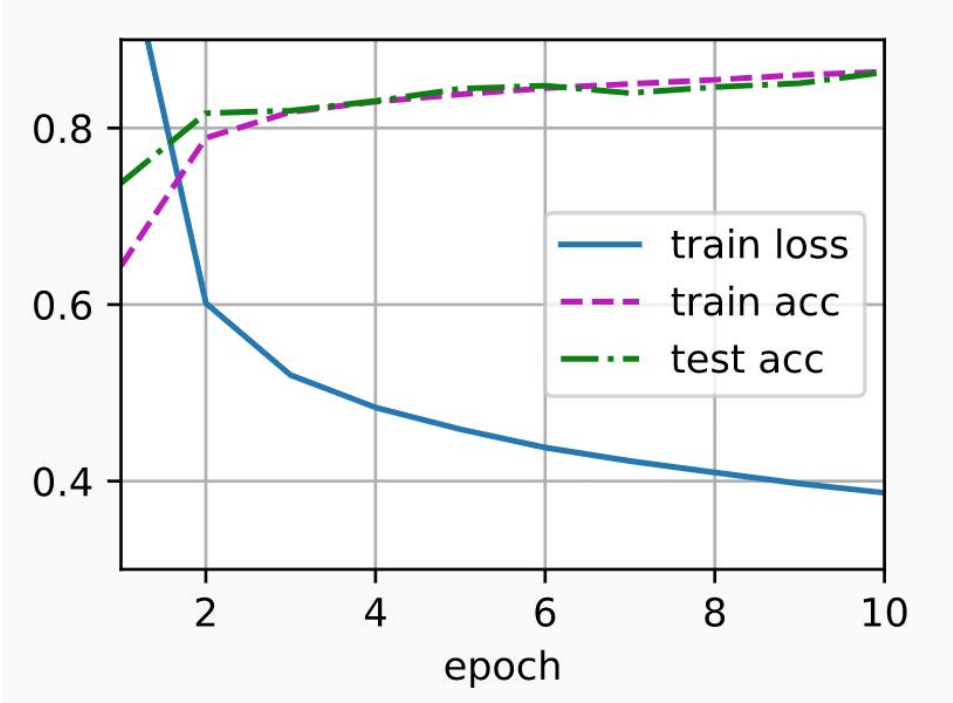
搭建多层感知机进行图像分类

```pytorch
# PyTorch
from torch import nn
net = nn.Sequential(nn.Flatten(),
                    nn.Linear(784, 256),
                    nn.ReLU(),
                    nn.Linear(256, 10))
```

```tensorflow
# Tensorflow
import tensorflow as tf
net = tf.keras.models.Sequential([
    tf.keras.layers.Flatten(),
    tf.keras.layers.Dense(256, activation='relu'),
    tf.keras.layers.Dense(10)])
```

分类效果

- 将迭代周期数设置为10，并将学习率设置为0.1，可以得到如图所示结果：
- 测试集准确率为85.72%



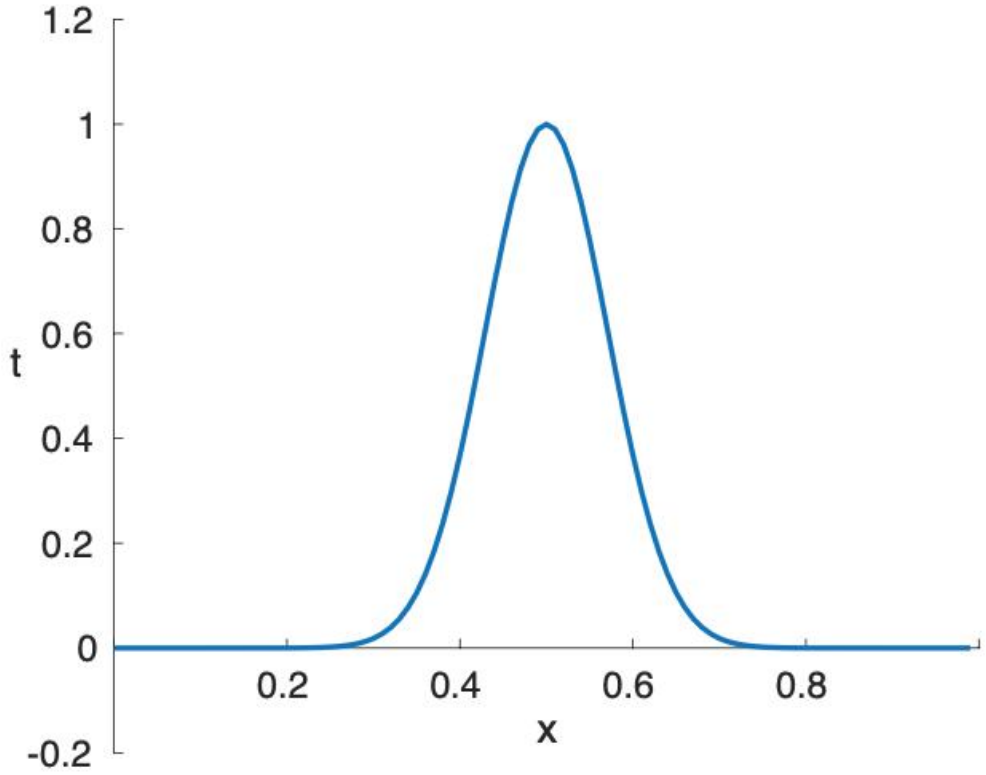
- **思考题**：根据本章知识，哪些改进可以帮助该MLP提升分类准确率？

拟合函数

根据**通用近似定理**可知：只要有足够的参数，神经网络能够近似任意函数

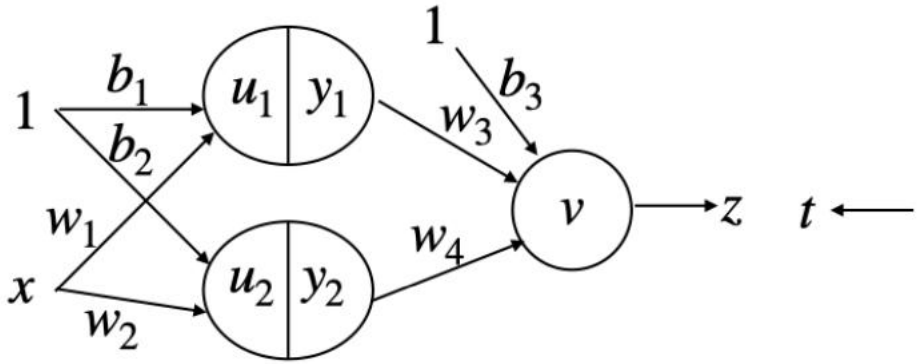
示例：函数的解析式为 $t = e^{-100(x-0.5)^2}$，

- $x < 0.2$或$x > 0.8$时，函数值t近似为常数 0；
- $0.2 \le x \le 0.5$时，函数值上升至 1；
- $0.5 \le x \le 0.8$时，函数值下降至 0。



拟合函数

为了逼近该函数，我们采用含有两个隐含神经元和一个输出神经元的MLP模型，其中隐含神经元具有 Sigmoid 激活函数。



其中 x 为输入，b_1 和 b_2 分别为两个隐含神经元的偏置，w_1 和 w_2 分别为输入到两个隐含神经元的权值，而对于输出神经元，b_3 表示偏置，w_3 和 w_4 分别为两个隐含神经元到输出神经元相关的权值。

该神经网络中一共有 7 个未知权值，对神经网络的训练就是对这些未知权值进行调整。

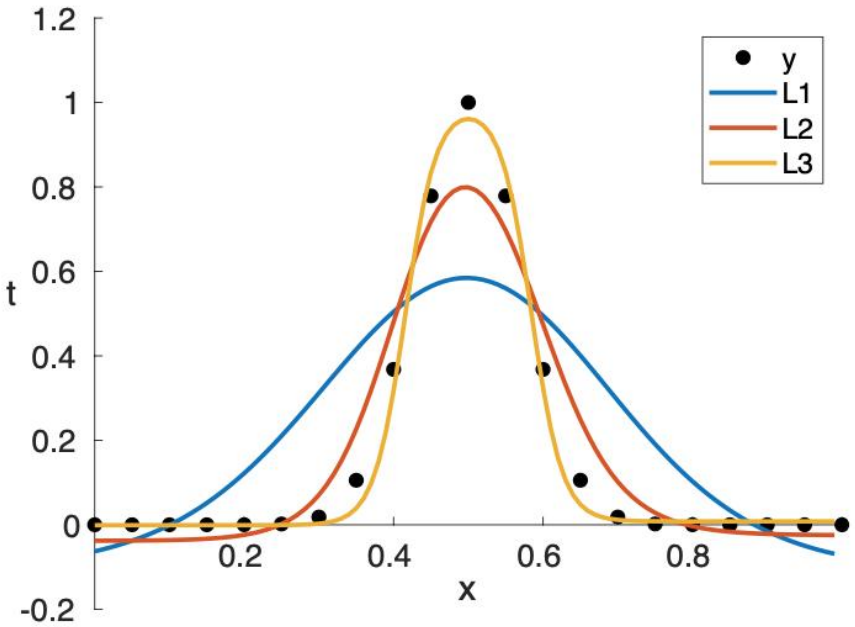
拟合函数

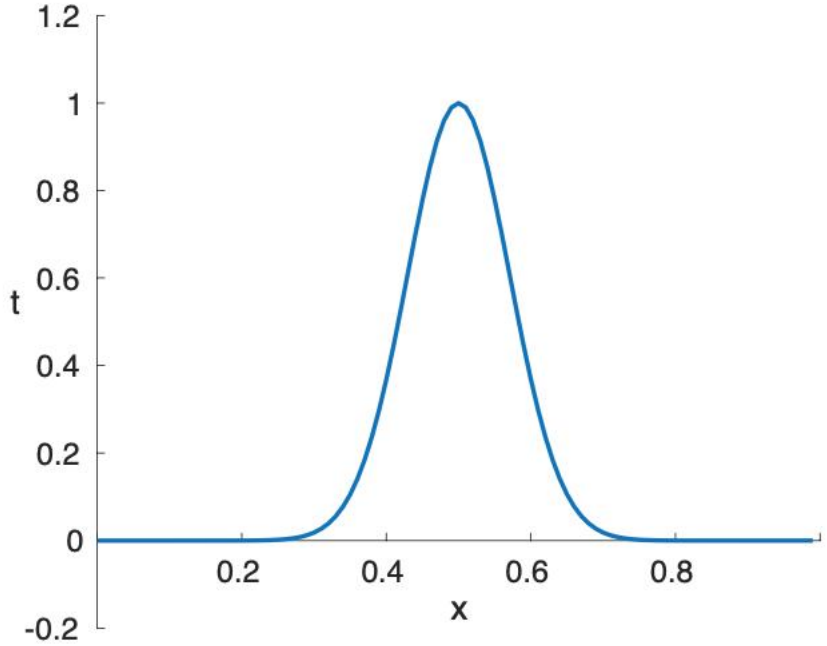
- 初始权值下模型的输出值和目标函数的值相差很大，通过对网络进行训练，网络中各权值会产生较大的变化。
- 选取初始权值、训练过程中的两次中间权值和训练完成后的最终权值，这四组网络权值如表所示：

	b_1	b_2	w_1	w_2	b_3	w_3	w_4
0	0.0	0.0	1.188	−1.295	0.002	−0.013	0.483
1	6.093	3.058	−9.292	−8.985	−0.114	1.131	−1.128
2	11.14	8.594	−18.834	−21.314	−0.025	1.122	−1.135
3	28.258	20.088	−48.348	−48.338	0.008	0.986	−0.995

拟合函数

- 神经网络产生的输出如左图所示：



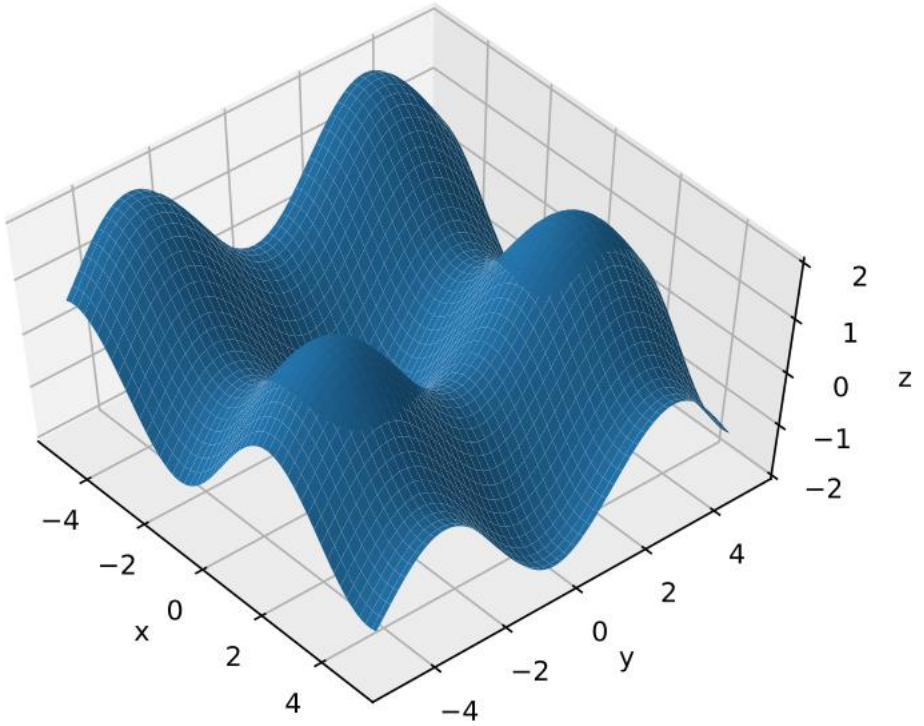


- 随着网络的训练，拟合结果越来越接近目标函数，网络的最终输出基本上与目标函数的值一致。

拟合曲面

多层神经网络具有强大的非线性表示能力，可以用多层感知机来拟合一个曲面。

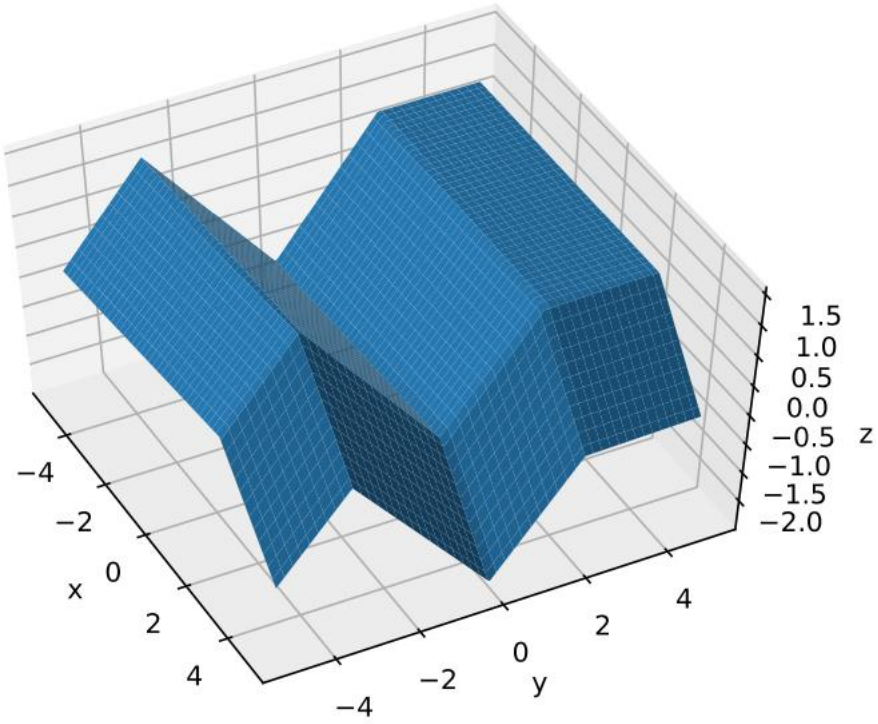
- 曲面的方程：$y = \sin(x_1) - \cos(x_2),\ x_1 \in [-5, 5],\ x_2 \in [-5, 5]$
- 曲面如图所示：



拟合曲面

使用单隐藏层的MLP来拟合，其中隐藏层采用 ReLU 激活函数，输出层使用线性激活函数。

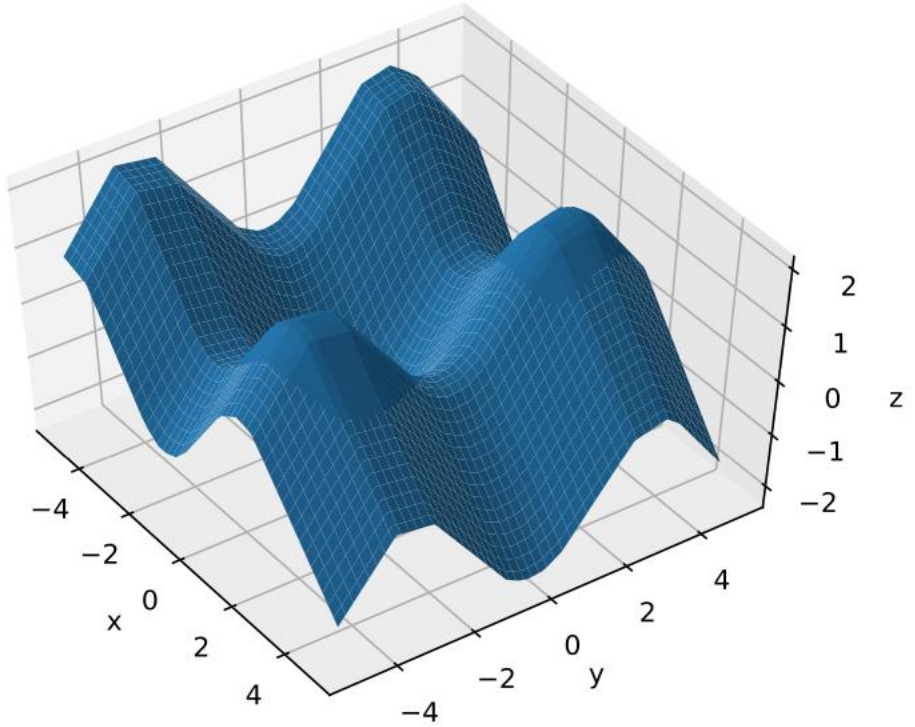
首先，将隐藏层神经元的数量设为 4，训练结果如图所示，训练完成后的模型并不能很好的拟合原始曲面。



拟合曲面

在这里 4 个神经元无法表示这个二维曲面，所以我们采用增加神经元数量的方法提高网络性能。

将隐藏层神经元的数量提升到 32 个，训练结果如图所示，此时可以很好的拟合原始曲面。



05 思考題

思考题

1. 什么是单层感知机的局限性，造成其局限性的原因是什么，该如何解决？

2. 相对于单层感知机而言，多层感知机有什么优点，又带来什么问题？

3. 比较绝对误差与平方误差的优劣，了解其他误差函数

4．描述梯度下降的优缺点和局部最小值的定义，使用梯度下降方法优化 Himmelblau 函数：$f(x,y)=(x^2+y-11)^2+(x+y^2-7)^2$．可使用 PyTorch 框架。（Himmelblau 函数可视化代码如下）：

```
import numpy as np
import matplotlib.pyplot as plt
from mpl_toolkits.mplot3d import Axes3D
def him(x):
    return (x[0]**2+x[1]-11)**2 + (x[0]+x[1]**2-7)**2

x = np.arange(-6,6,0.1)
y = np.arange(-6,6,0.1)
X,Y = np.meshgrid(x,y)
Z = him([X,Y])

fig = plt.figure()
ax = fig.gca(projection='3d')
ax.plot_surface(X,Y,Z,cmap='rainbow')
ax.set_xlabel('x[0]')
ax.set_ylabel('x[1]')
ax.set_zlabel('f')
fig.show()
```

5．在第 4 题的基础上，使用不同的优化器并添加动量参数 momentum，观察收敛过程以及最后的收敛的结果

6. 代码设计单层感知机，使其满足 y = σ(W x + b)。(其函数定义如以下代码，只需完成其实现)

```
class Layer():
    def __init__(self,input_dim,output_dim,bias=True):
        None
    def __call__(self,x,train=False):
        None
    def forward(self,x,train):
        None
    #反向传播函数
    def backward(self,error,eta):
        None
```

7. 在第6题的基础上，使用 Layer 类完成多层感知机的构造，其成员函数和layer一致，并用于对cos(2πx)函数的拟合。

思考题

5. 在第 4 题的基础上，使用不同的优化器并添加动量参数 momentum，观察收敛过程以及最后的收敛的结果。

Q&A

Questions and Answers