

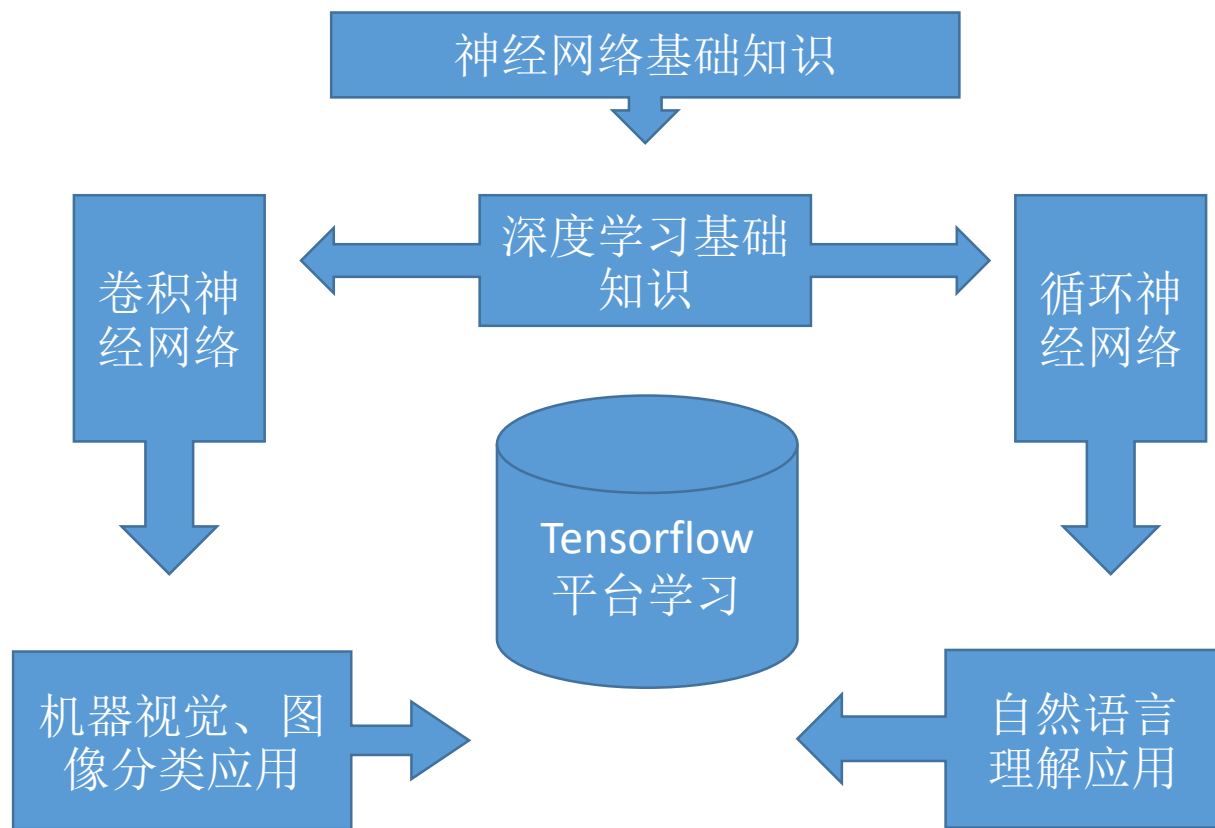
# 人工智能之深度学习

## 深度学习概述

上海育创网络科技有限公司

主讲人: Steven Tang

# 课程体系介绍



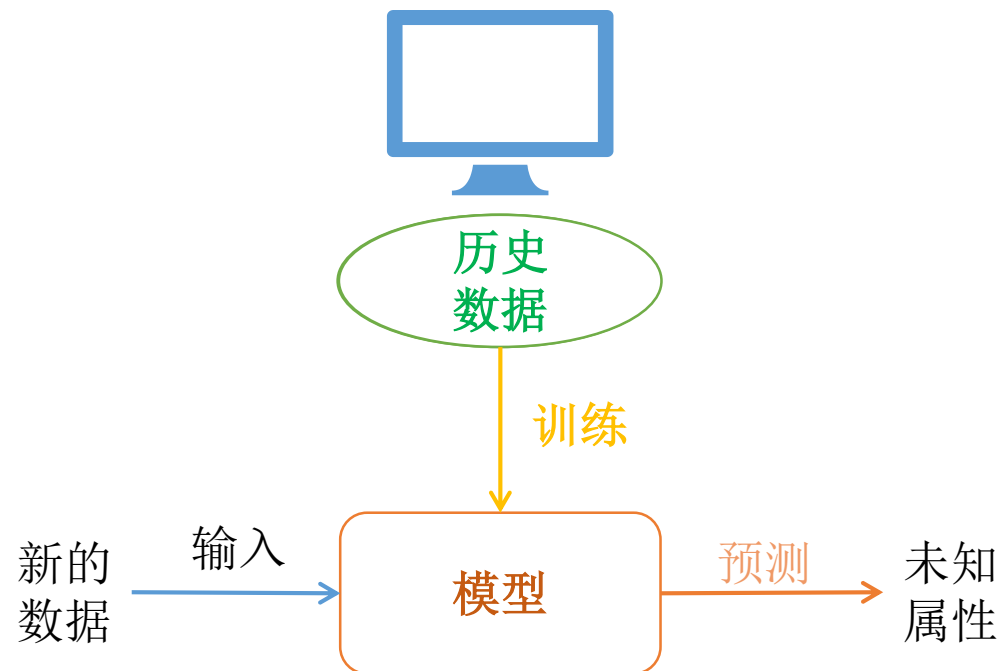
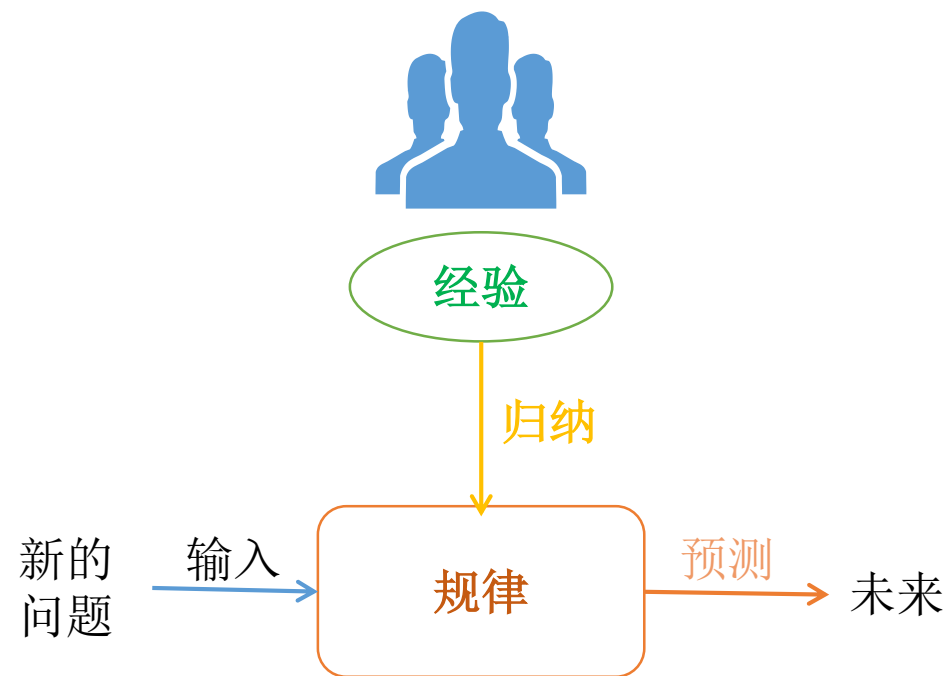
# 课程内容

- 深度学习概述
- 神经网络的发展
- 感知机
- 多层神经元网络
- BP神经网络

# 深度学习的应用

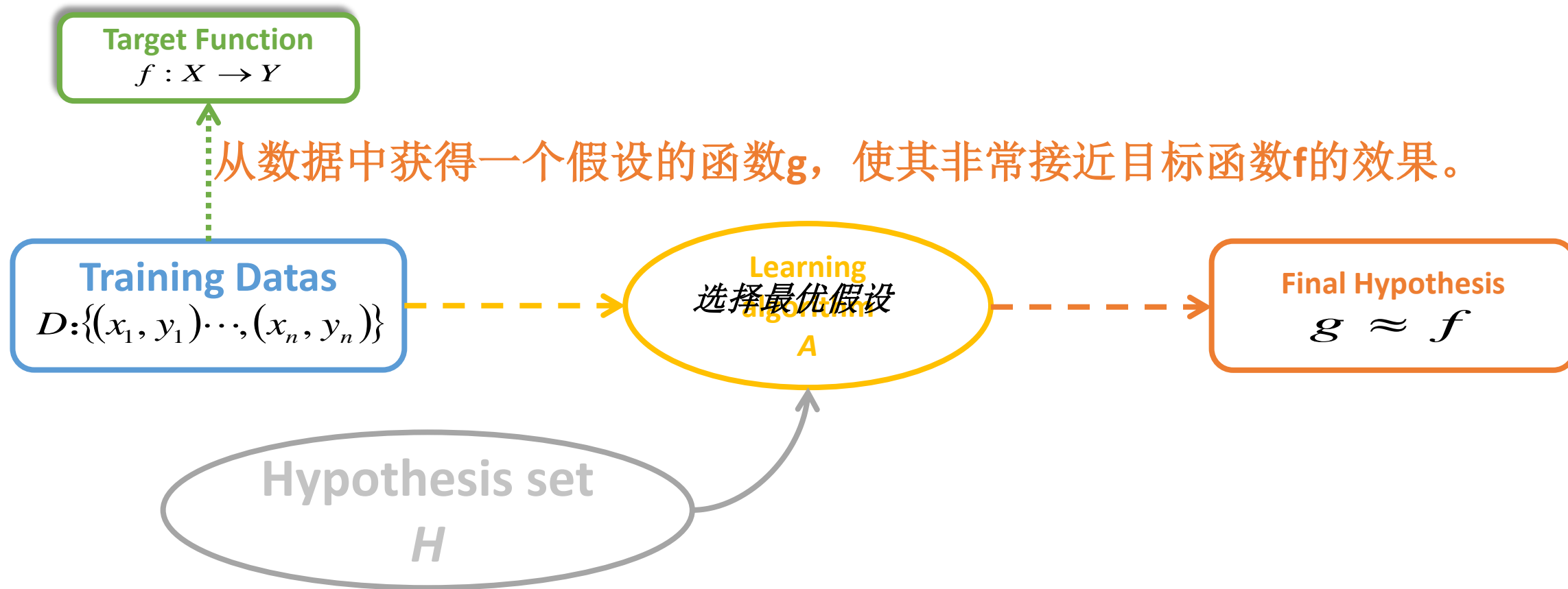
- 图像应用
  - 大规模(大数据量)图片识别(聚类/分类), 如**人脸识别**, **车牌识别**, OCR等
  - 以图搜图, 图像分割
  - **目标检测**, 如自动驾驶的行人检测, 安防系统的异常人群检测
- 自然语言处理
  - 语音识别, 语音合成, 自动分词, 句法分析, 语法纠错, **关键词提取**, 文本分类/聚类, 文本自动摘要, 信息检索 (ES,Solr)
  - 知识图谱, 机器翻译, **人机对话**, 机器写作
  - **推荐系统**, 高考机器人
  - **信息抽取**, **网络爬虫**, **情感分析**, 问答系统
- 数据挖掘, 风控系统, 广告系统等

# 机器学习回顾



# 机器学习回顾

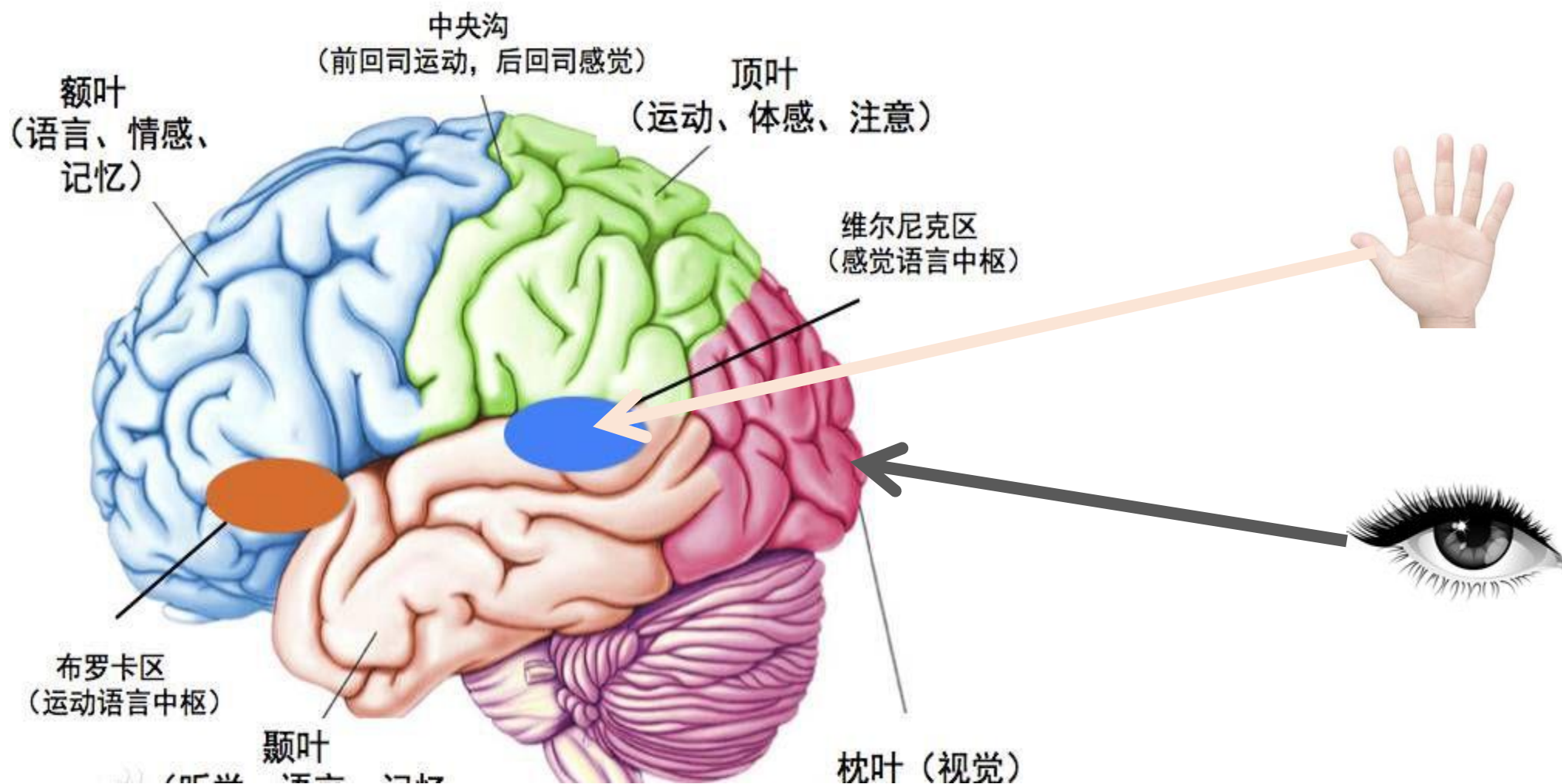
- 机器学习



# 深度学习vs机器学习

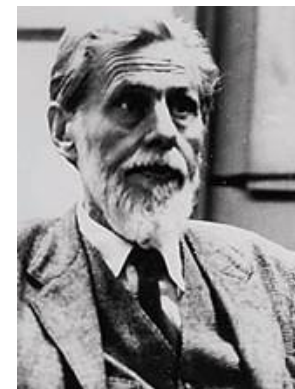
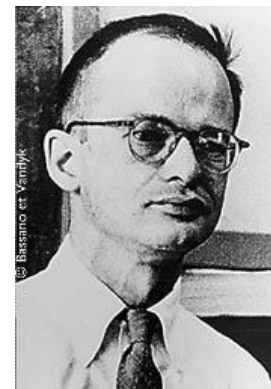
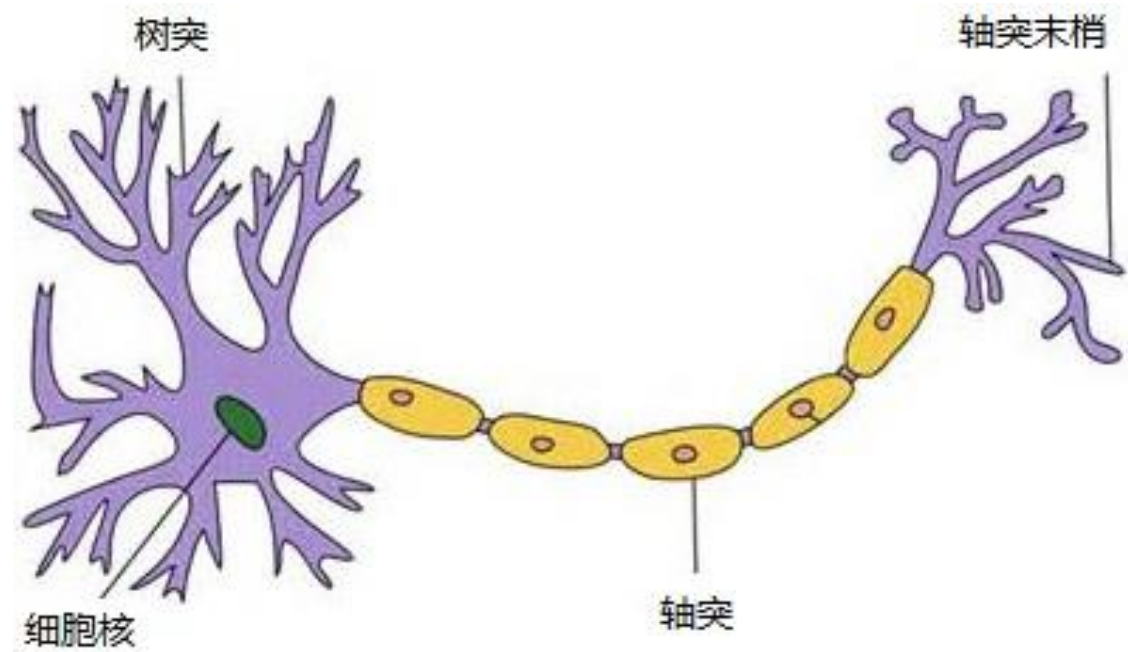
- 数据、特征、模型
- 端对端学习
- 真实数据vs抽象数据

# 神经网络来源之人的思考





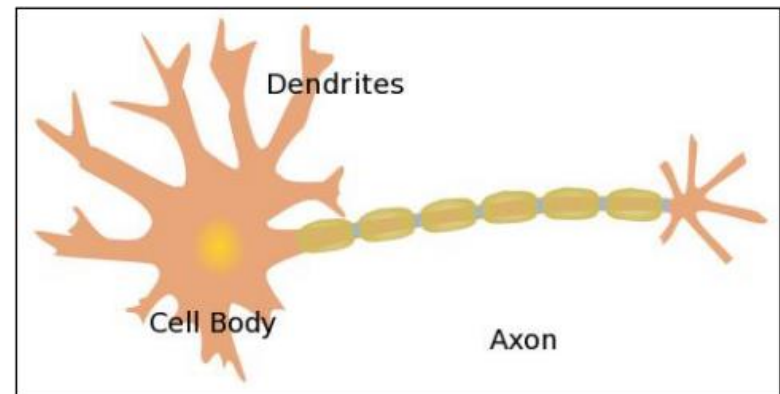
# 生物神经元



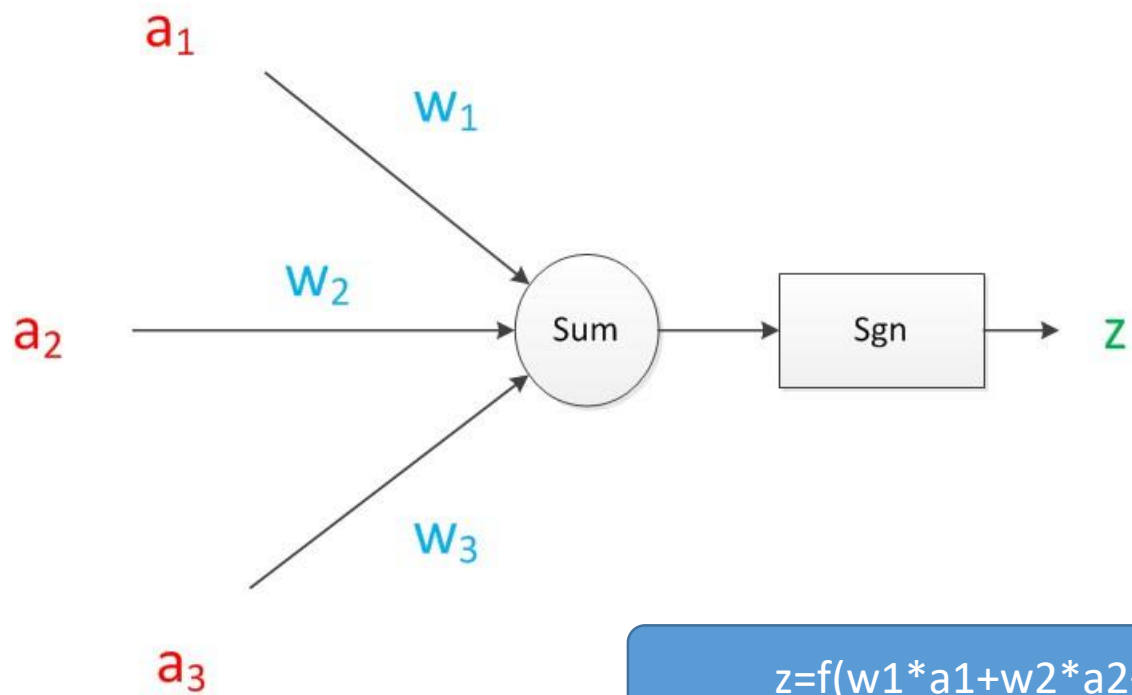
Warren McCulloch (右) 和 Walter Pitts (左)

## 神经网络来源之人的思考

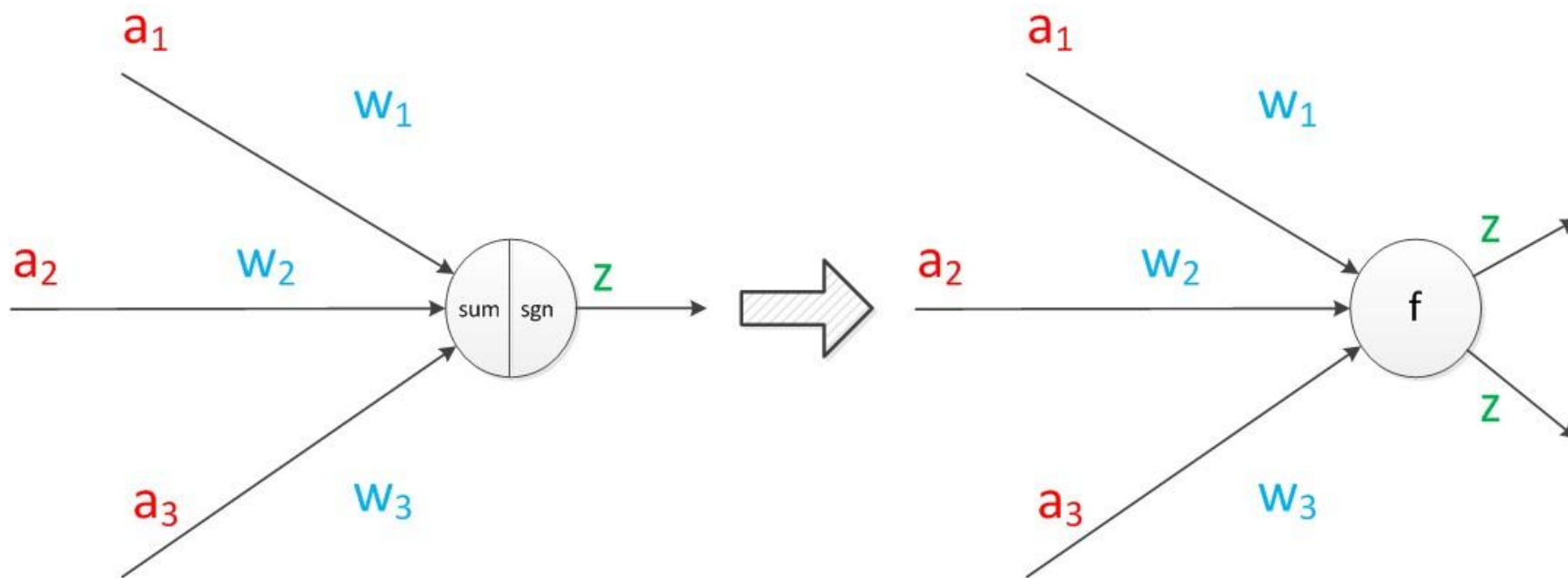
- 大脑是由处理信息的**神经元细胞**和连接神经元的细胞进行信息传递的**突触**构成的。树突(Dendrites)从一个神经元接受电信号，信号在**细胞核**(Cell Body)处理后，然后通过**轴突**(Axon)将处理的信号传递给下一个神经元。
- 一个神经元可以看作是将一个或多个输入处理成一个输出的计算单元。
- 通过多个神经元的传递，最终大脑会得到这个信息，并可以对这个信息给出一个合适的反馈。



# 人工神经元模型



# 人工神经元模型



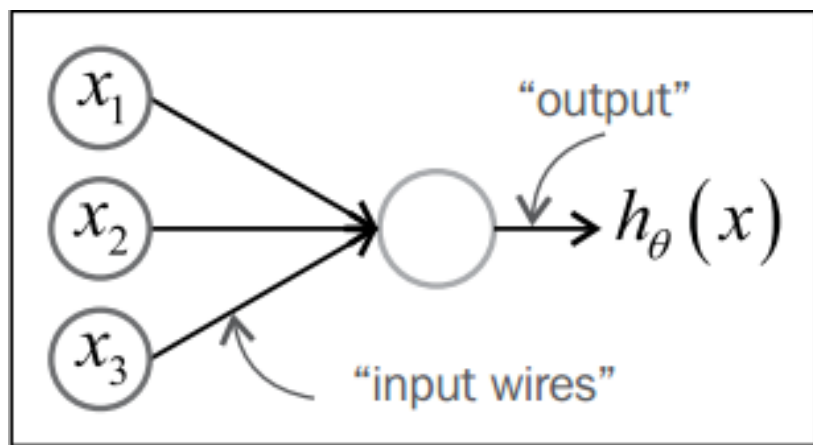
## 感知机模型

- 1958年，计算科学家 Rosenblatt 提出了由两层神经元组成的神经网络。他给它起了一个名字--“感知器”（Perceptron）（有的文献翻译成“感知机”，我们统一用“感知器”来指代）。



## 感知器模型

- 感知器是一种模拟人的神经元的一种算法模型，是一种研究单个训练样本的二元分类器，是SVM和人工神经网络(ANN, Artificial Neural Networks)的基础。
- 一个感知器接受几个二进制的输入，并产生一个二进制的输出，通常的表达方式如下：



$$output = \begin{cases} 0, & \text{if } \sum_j w_j x_j \leq threshold \\ 1, & \text{if } \sum_j w_j x_j > threshold \end{cases}$$

$$output = \begin{cases} 0, & \text{if } w \cdot x + b \leq 0 \\ 1, & \text{if } w \cdot x + b > 0 \end{cases}$$

## 感知器模型案例直观理解

- **感知器**可以看作是根据权重来做出决定的一个设备/单元，只要我们可以给定一个比较适合  
的权重以及阈值，那么感知器应该是能够对数据进行判断的/分类预测的。假定你现在在考  
虑是否换工作，也许你会考虑一下三个方面的因素：
  - 新工作的待遇会提高吗？
  - 你家庭经济压力大吗？
  - 新工作稳定吗？

## 一个最简单的例子（什么物体可以飞？）

- 手工选择的特征

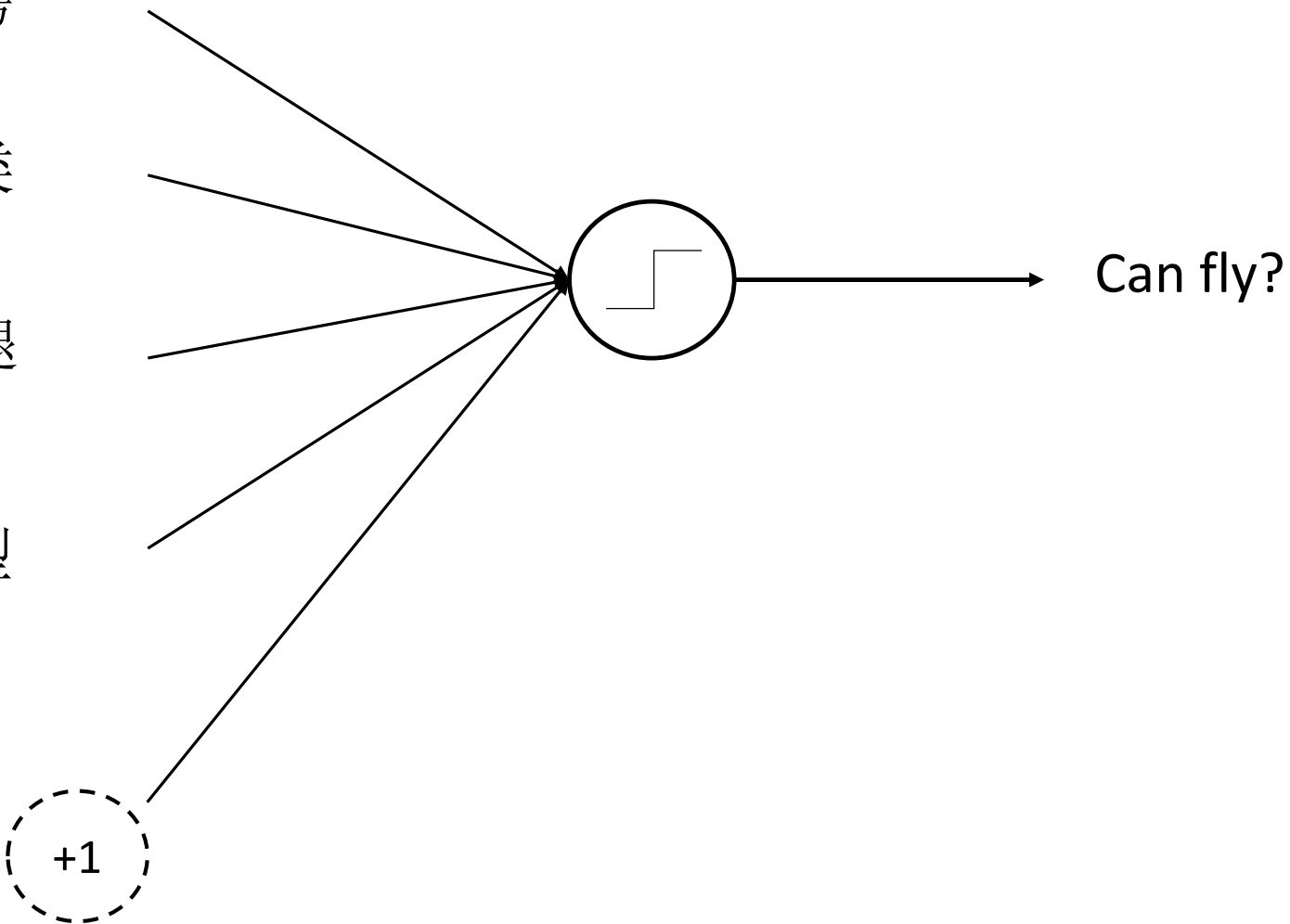
有翅膀

是鸟类

四条腿

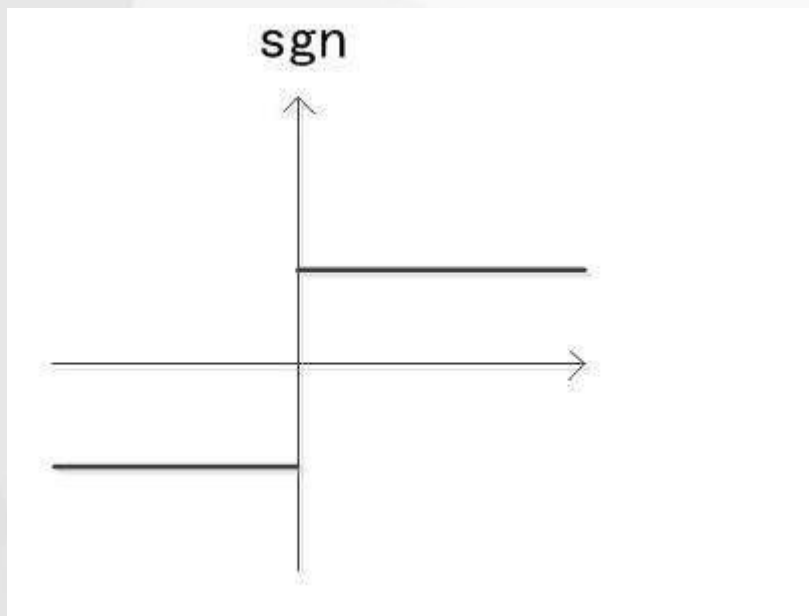
大体型

+1





# perceptron感知机:激活函数输出



$$f(x) = \text{sign}(x) = \begin{cases} 1 & x > 0 \\ -1 & x \leq 0 \end{cases}$$



$x0=[1,1,-1,-1]$   $y0=1$



$x1=[1,-1,-1,1]$   $y1=1$

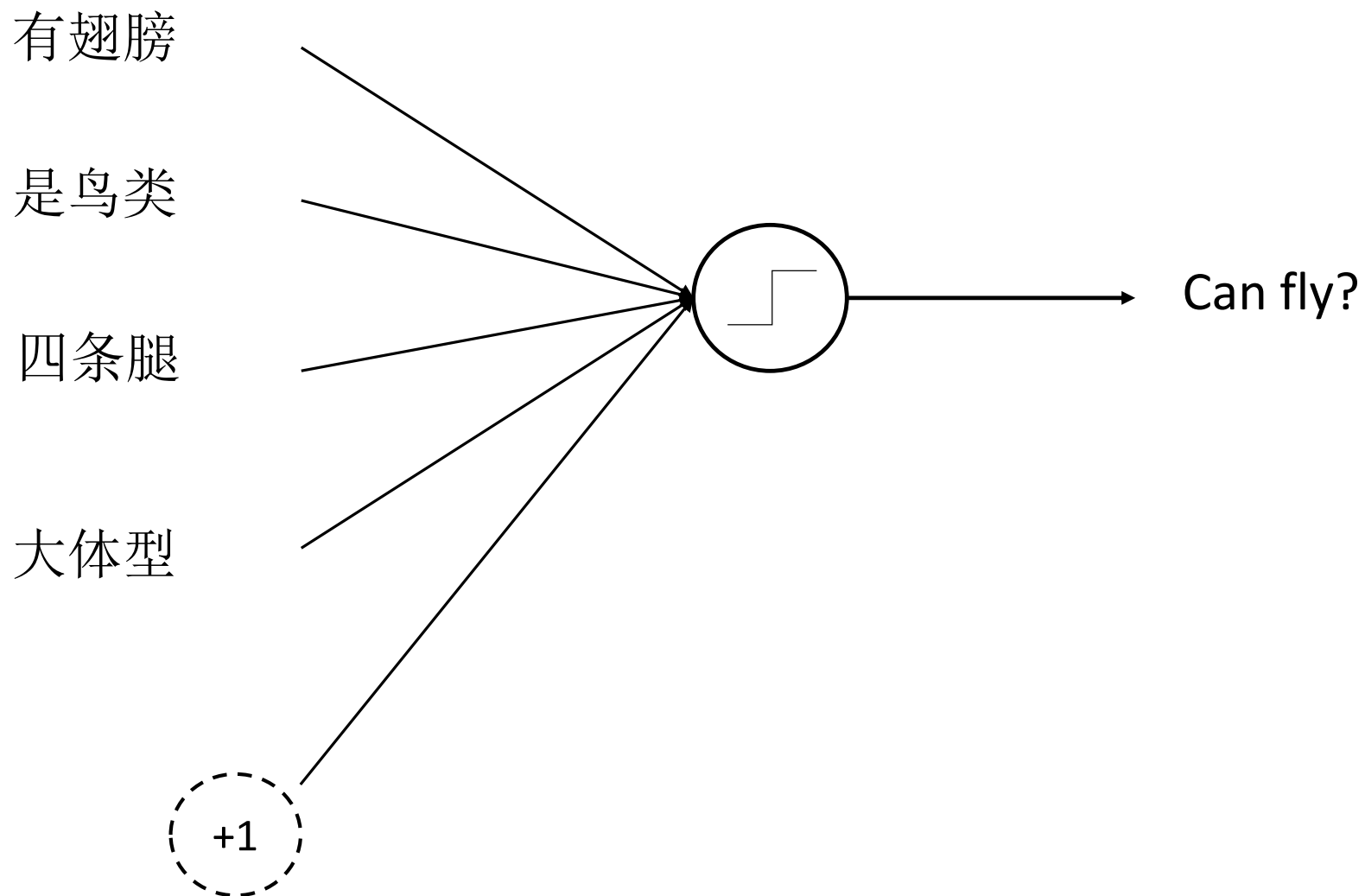


$x2=[1,1,-1,-1]$   $y2=-1$



$x3=[-1,-1,1,1]$   $y3=-1$

## 怎么来学习一组参数？



$$W = (0.3, 0.1, 0.2, -0.2)$$

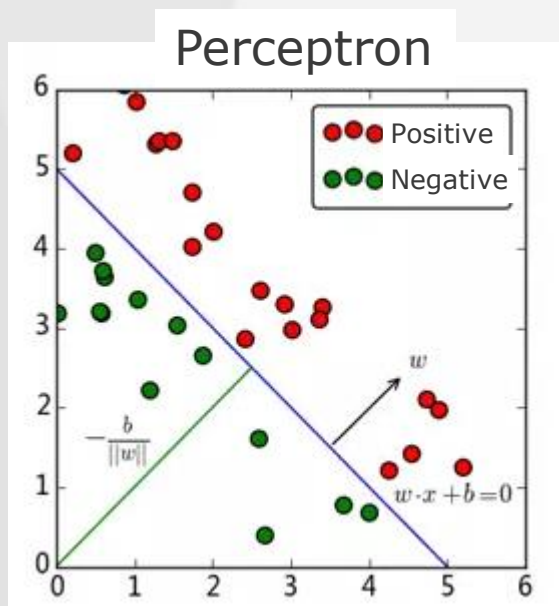
$$X = \begin{bmatrix} 1, 1, 1, -1 \\ 1, -1, -1, -1 \\ -1, -1, -1, 1 \\ -1, 1, -1, 1 \end{bmatrix}$$

$$b = -0.3$$

$$\eta = 0.1$$

# 感知机的损失函数

错分的样本到分界面的距离



$$L(w, b) = \arg \min_{w, b} \left( - \sum_{x_i \in M} y_i (w \cdot x_i + b) \right)$$

## 感知机的损失函数推导

- 首先任意点  $x_i$  到超平面  $S$  的距离: 
$$\frac{1}{\|w\|} |w \cdot x_0 + b|$$
- 当  $w^*x_i + b > 0$  时,  $y_i = -1$ , 而当  $w^*x_i + b < 0$  时,  $y_i = 1$ 。因此, 对于误分类的数据  $(x_i, y_i)$  来说,  $-y_i(w^*x_i + b) > 0$  成立。
- 另外, 误差分类点到超平面  $S$  的距离是 
$$\frac{1}{\|w\|} y_i (w \cdot x_i + b)$$
- 设  $M$  为超平面  $S$  的误分类点的集合, 则所有误分类点到超平面  $S$  的总距离为: 
$$\frac{1}{\|w\|} \sum_{x \in M} y_i (w \cdot x_i + b)$$

## 感知机的损失函数推导

- 给定一个训练数据集

$$T = \{(x_1, y_1), (x_2, y_2), \dots, (x_N, y_N)\}$$

$$L(w, b) = - \sum_{x \in M} y_i (w \cdot x_i + b)$$

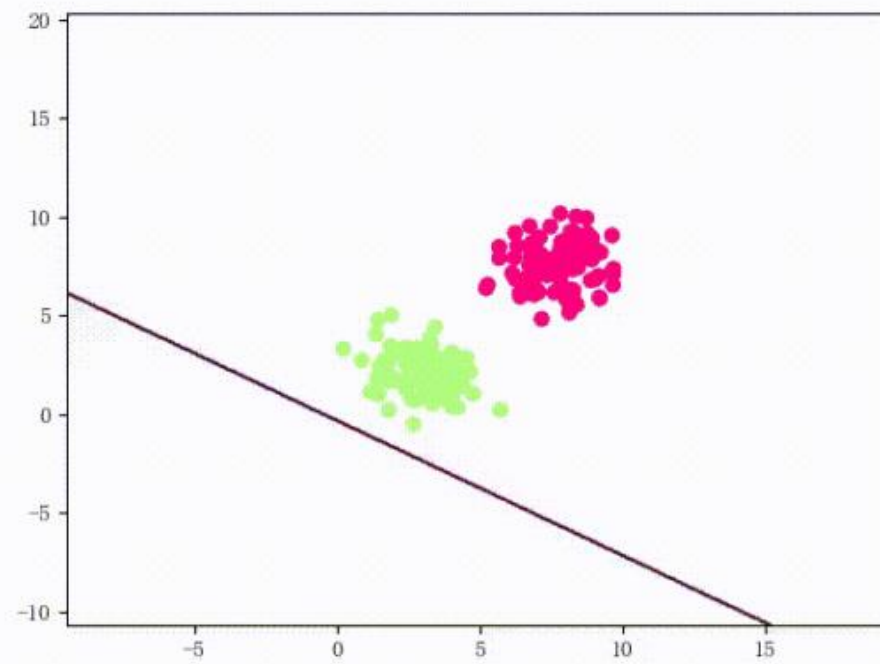
- 假设误分类点的集合为  $M$ ，那么损失函数  $L(w, b)$  的梯度为：

$$\nabla_w L(w, b) = - \sum_{x \in M} y_i x_i$$

$$\nabla_b L(w, b) = - \sum_{x \in M} y_i$$

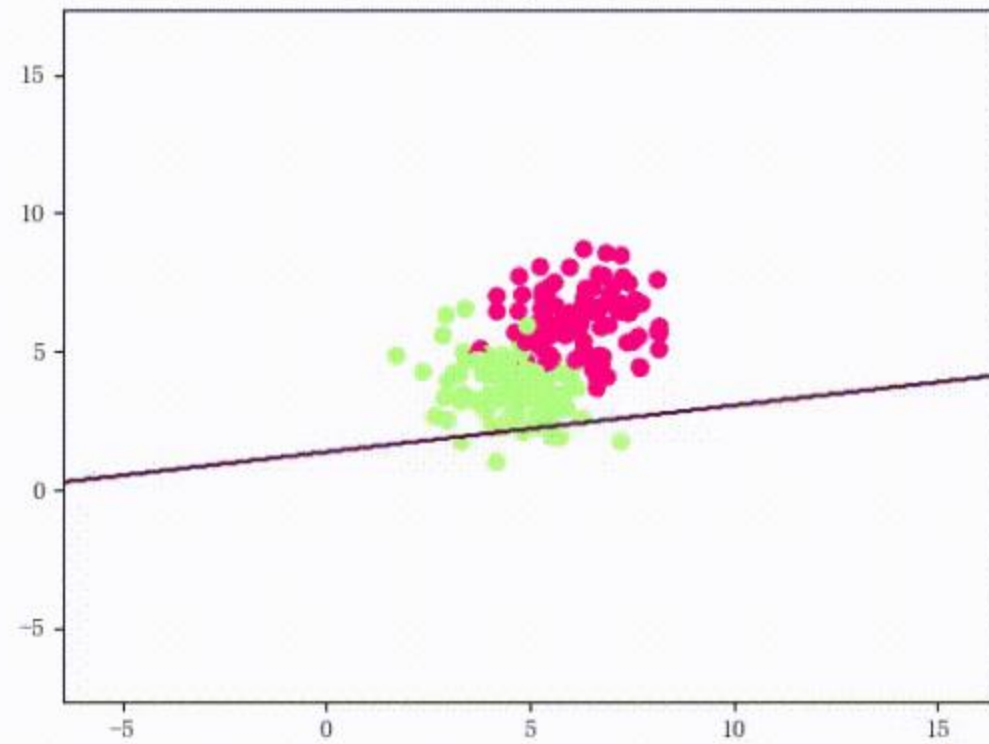
## 感知机训练过程

- 1. 随机初始化所有权重;
- 2. 随机选择一组训练样本  $\langle x_i, y_i \rangle$ ;
- 3. if  $y_i * f(W * x_i + b) < 0$   
    
$$W \leftarrow W + \eta y_i x_i$$
  
    
$$b \leftarrow b + \eta y_i$$
- 4. 继续到第二步直至所有样本被正确分类。



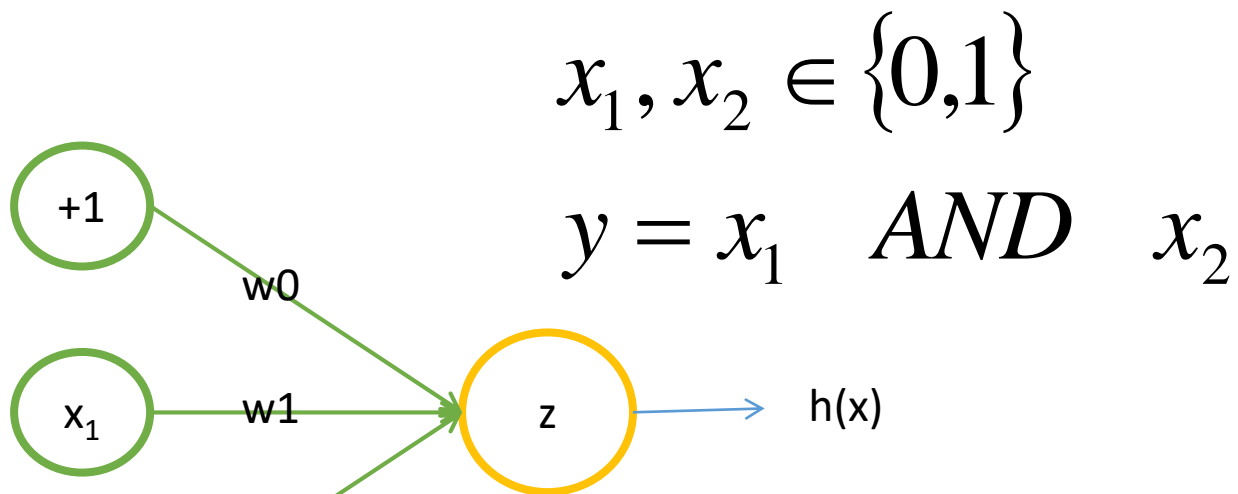


Perceptron



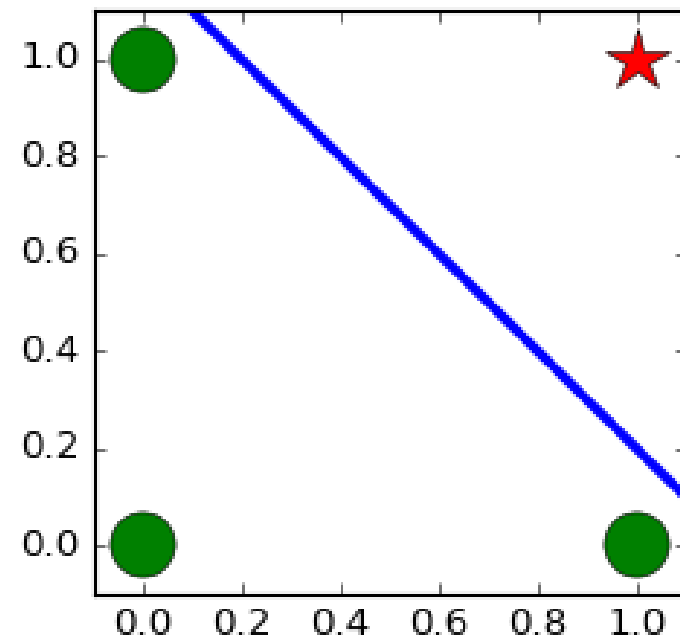
## 感知器神经元直观理解之逻辑与

- 单个神经元完成逻辑与功能



$$W = (-3, 2, 2)$$

$$h_w(z) = h(-3 + 2 \cdot x_1 + 2 \cdot x_2) = \begin{cases} 0, & z < 0 \\ 1, & z \geq 0 \end{cases}$$



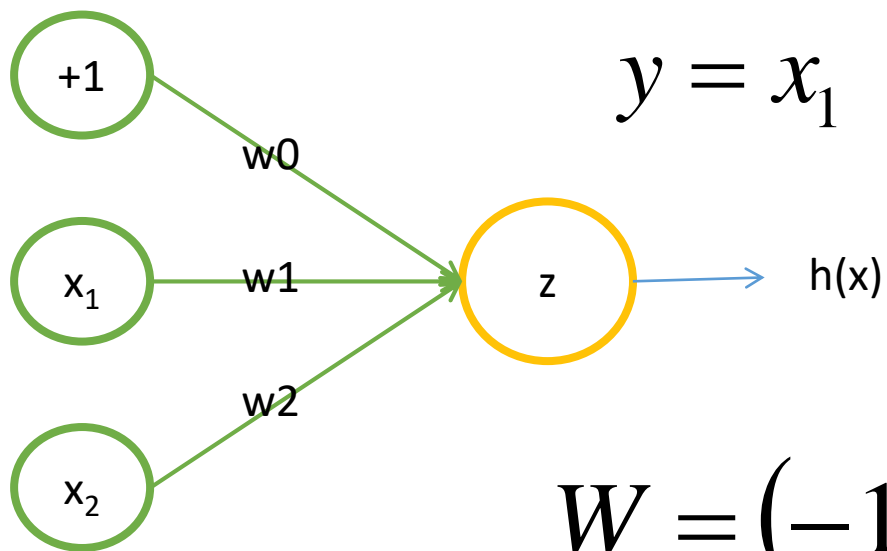
$x_1$	$x_2$	$h_{\Theta}(x)$
0	0	0
0	1	0
1	0	0
1	1	1

## 感知器神经元直观理解之逻辑或

- 单个神经元完成逻辑或功能

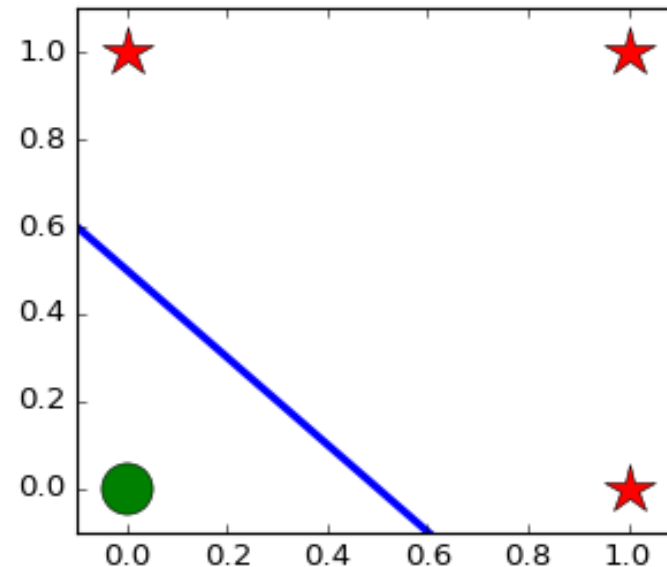
$$x_1, x_2 \in \{0, 1\}$$

$$y = x_1 \text{ OR } x_2$$



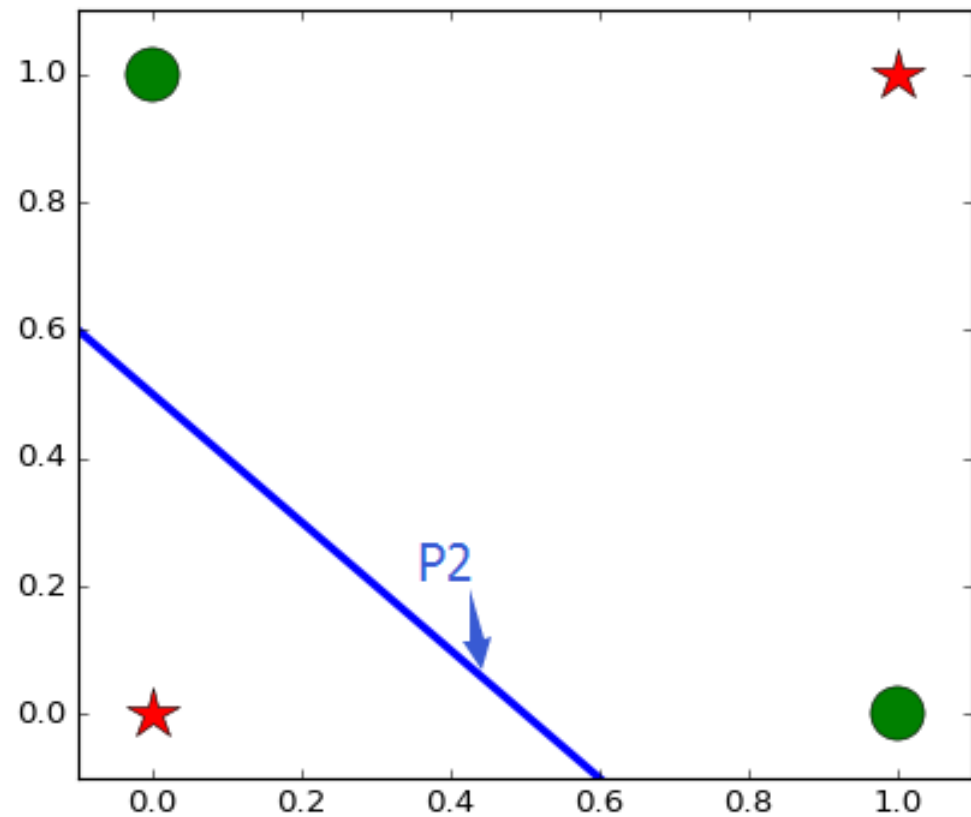
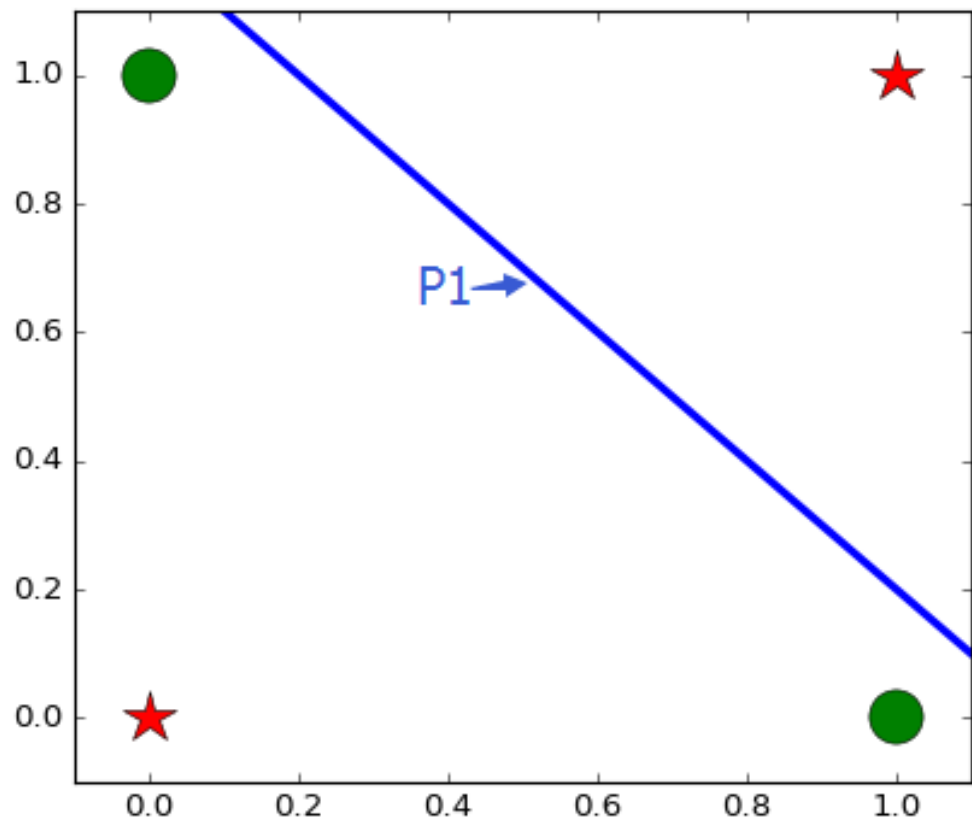
$$W = (-1, 2, 2)$$

$$h_w(z) = h(-1 + 2 \cdot x_1 + 2 \cdot x_2) = \begin{cases} 0, & z < 0 \\ 1, & z \geq 0 \end{cases}$$

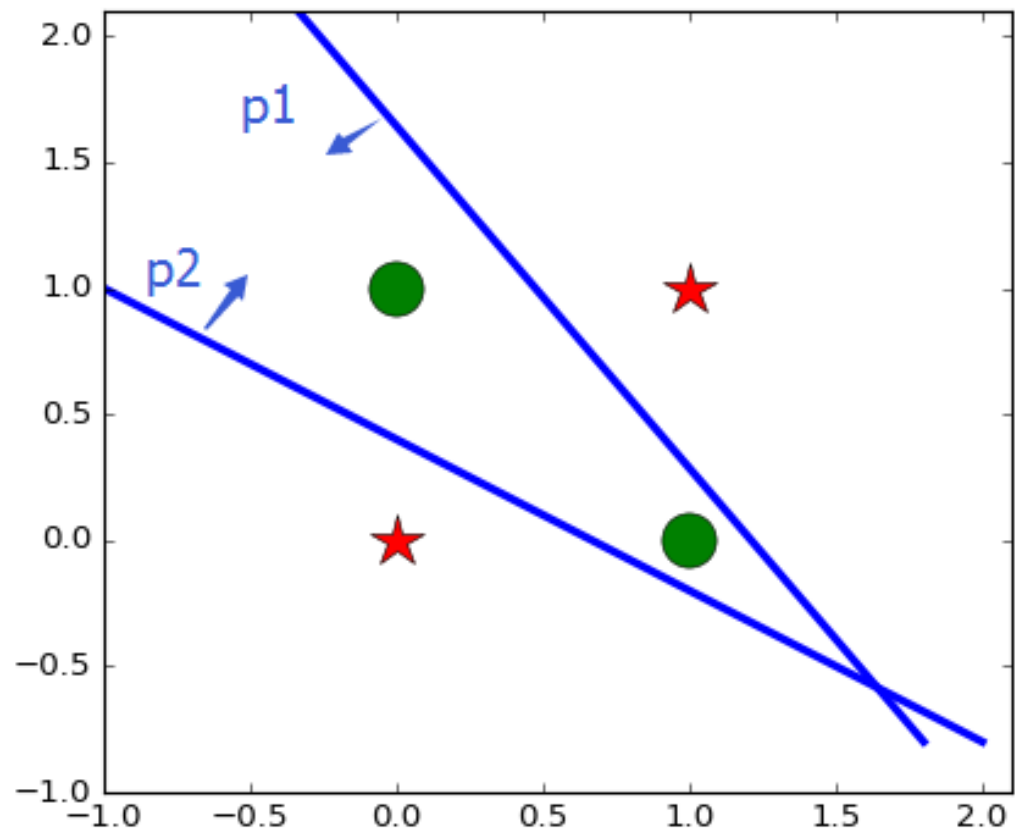
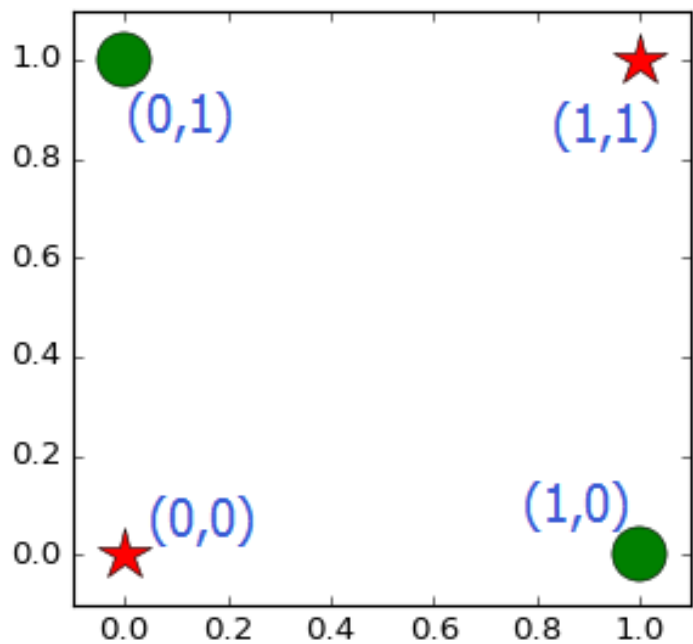


$x_1$	$x_2$	$h_{\Theta}(x)$
0	0	0
0	1	1
1	0	1
1	1	1

## 感知器神经元直观理解之非线性可分



## 感知器神经元直观理解之非线性可分



# 感知机的对偶形式

感知机学习算法的原始形式

对于输入空间，感知机通过以下函数将其映射至 $\{+1, -1\}$ 的输出空间

$$f(x) = \text{sign}(w \cdot x + b) \quad (1)$$

对于所有的错分类点 $i \in M_i$ ，都有 $-y_i(w \cdot x_i + b) > 0$ ，  
因此我们可以定义如下的损失函数作为优化准则：

$$L(w, b) = -\sum_{x_i \in M} y_i (w \cdot x_i + b)$$

然后我们可以得到梯度

$$\nabla_w L(w, b) = -\sum_{x_i \in M} y_i x_i$$

$$\nabla_b L(w, b) = -\sum_{x_i \in M} y_i$$

来进行更新：

$$w \leftarrow w + \eta y_i x_i$$

$$b \leftarrow b + \eta y_i$$

## 感知机的对偶形式

- 感知机梯度的更新是由错判样本的出现次数决定的。
- $N = (n_0, n_1, \dots, n_{L-1})$
- 如果按  $\mathbf{x}_1, \mathbf{x}_1, \mathbf{x}_2, \mathbf{x}_3, \mathbf{x}_3, \mathbf{x}_3, \mathbf{x}_4, \mathbf{x}_5, \mathbf{x}_5$  顺序学习,  
 $n_1 = 2, n_2 = 1, n_3 = 3, n_4 = 1, n_5 = 2$

## 感知机的对偶形式

- 现在 $w$ ,  $b$ 每次在感知到误判的时候更好的迭代思路是:

$$\mathbf{w}_{i+1} \leftarrow \mathbf{w}_i + \eta \cdot n_i \cdot \mathbf{y}_i \mathbf{x}_i$$

$$b \leftarrow b + \eta y_i$$

$$\text{令 } \alpha_i = \eta \cdot n_i$$

如果给定一个默认参数  $\mathbf{w}_0 = \mathbf{0}$  开始迭代, 则有:

$$w_1 = w_0 + \eta \cdot n_1 \cdot y_1 x_1 = 0 + \eta \cdot n_1 \cdot y_1 x_1 = \alpha_1 \cdot y_1 x_1$$

$$w_2 = w_1 + \eta \cdot n_2 \cdot y_2 x_2 = \alpha_1 \cdot y_1 x_1 + \eta \cdot n_2 \cdot y_2 x_2 = \alpha_1 \cdot y_1 x_1 + \alpha_2 \cdot y_2 x_2$$

$$w_3 = w_2 + \eta \cdot n_3 \cdot y_3 x_3 = w_2 + \alpha_3 \cdot y_3 x_3 = \alpha_1 \cdot y_1 x_1 + \alpha_2 \cdot y_2 x_2 + \alpha_3 \cdot y_3 x_3$$

$$w_i = \sum_c^i \alpha_c \cdot y_c x_c$$



## 感知机的对偶形式

$$\alpha_i \leftarrow \eta(n_i + 1) = \eta n_i + \eta = \alpha_i + \eta \text{ 即}$$

$$\alpha_i \leftarrow \alpha_i + \eta$$

所以，我们的对偶问题变为，

$$\alpha_i \leftarrow \alpha_i + \eta$$

$$b \leftarrow b + \eta y_i$$

## 感知机的对偶形式

- 最后我们可以得到

$$w = \sum_{i=1}^N n_i \eta y_i x_i$$

$$b = \sum_{i=1}^N n_i \eta y_i$$

$$f(x) = \text{sign}(w \cdot x + b) = \text{sign}\left(\sum_{j=1}^N n_j \eta y_j x_j \cdot x + \sum_{j=1}^N n_j \eta y_j\right)$$

相应地，训练过程变为：

1. 初始时  $\forall n_i = 0$ 。
2. 在训练集中选取数据  $(x_i, y_i)$
3. 如果  $y_i \left( \sum_{j=1}^N n_j \eta y_j x_j \cdot x_i + \sum_{j=1}^N n_j \eta y_j \right) \leq 0$ ，更新： $n_i \leftarrow n_i + 1$
4. 转至2直至没有误分类数据。

# 感知机的对偶形式

## Gram矩阵

$$G=[x_i * x_j]_{N*N}$$

定义： $n$ 维欧式空间中任意 $k(k \leq n)$ 个向量 $\alpha_1, \alpha_2, \dots, \alpha_k$ 的内积所组成的矩阵

$$\Delta(\alpha_1, \alpha_2, \dots, \alpha_k) = \begin{pmatrix} (\alpha_1, \alpha_1) & (\alpha_1, \alpha_2) & \dots & (\alpha_1, \alpha_k) \\ (\alpha_2, \alpha_1) & (\alpha_2, \alpha_2) & \dots & (\alpha_2, \alpha_k) \\ \dots & \dots & \dots & \dots \\ (\alpha_k, \alpha_1) & (\alpha_k, \alpha_2) & \dots & (\alpha_k, \alpha_k) \end{pmatrix}$$

称为 $k$ 个向量 $\alpha_1, \alpha_2, \dots, \alpha_k$ 的格拉姆矩阵（Gram矩阵），它的行列式称为Gram行列式。

**例 2.2** 数据同例 2.1，正样本点是 $x_1 = (3,3)^T$ ， $x_2 = (4,3)^T$ ，负样本点是 $x_3 = (1,1)^T$ ，试用感知机学习算法对偶形式求感知机模型。

**解** 按照算法 2.2，

(1) 取 $\alpha_i = 0$ ， $i=1,2,3$ ， $b=0$ ， $\eta=1$

(2) 计算 Gram 矩阵

$$G = \begin{bmatrix} 18 & 21 & 6 \\ 21 & 25 & 7 \\ 6 & 7 & 2 \end{bmatrix}$$

## 感知机的对偶形式

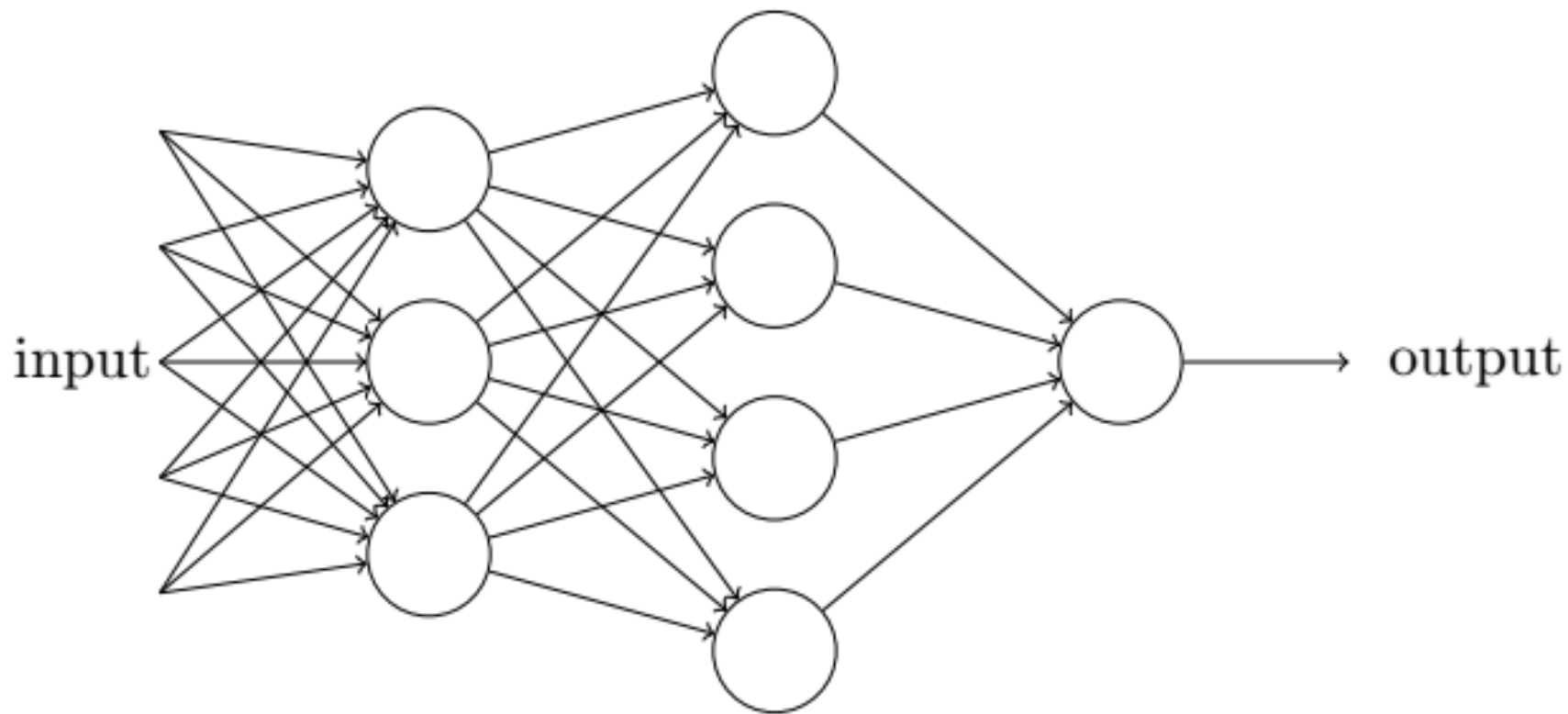
- 对偶形式的用途
- 当样本数量少、样本维度高时，能够节约计算资源。

## 神经网络游乐场初见

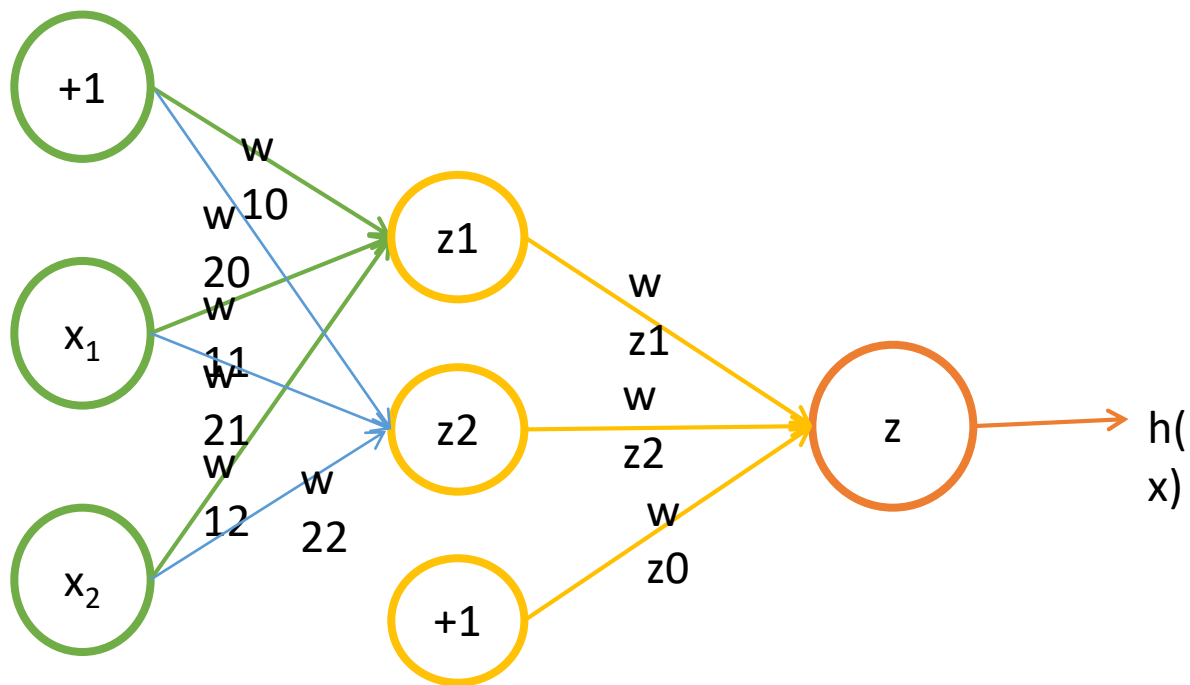
- <http://playground.tensorflow.org/#activation=linear&batchSize=10&dataset=gauss&regDataset=reg-plane&learningRate=0.0001&regularizationRate=0&noise=0&networkShape=1&seed=0.47753&showTestData=false&discretize=false&percentTrainData=50&x=true&y=true&xTimesY=false&xSquared=false&ySquared=false&cosX=false&sinX=false&cosY=false&sinY=false&collectStats=false&problem=classification&initZero=false&hideText=false>

## 多层感知器(人工神经网络)

- 将多个感知器进行组合，我们就可以得到一个多层感知器的网络结构，网络中的每一个节点我们叫做**神经元**。



## 感知器神经元直观理解之非线性可分



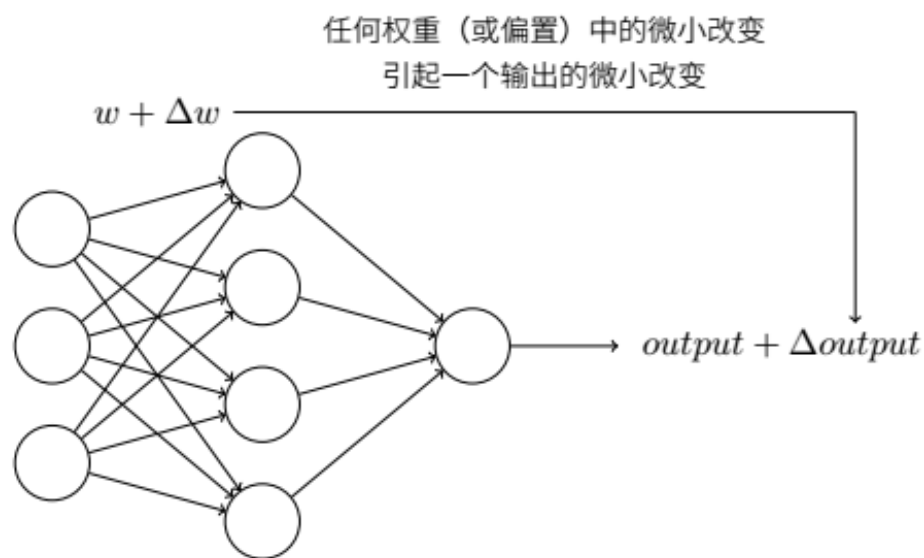
$x_1$	$x_2$	$z_1$	$z_2$	$h(x)$
0	0	0	0	1
0	1	0	1	0
1	0	0	1	0
1	1	1	1	1

$$W_1 = (-3, 2, 2) \quad W_2 = (-1, 2, 2) \quad W_z = (1, 3, -3)$$

$$h_w(z) = h(Wx) = \begin{cases} 0, & z < 0 \\ 1, & z \geq 0 \end{cases}$$

## 感知器网络理解以及S型神经元

- 其实只要将网络中的权重或者偏置项稍微的做一点小的改动，都会导致最终的输出发生一定的变化。但是在感知器神经网络中，单个感知器上的权重或者偏置项发现一点小的变化，最终的输出要不不变，要不完全翻转(因为只有两种取值0和1)，这种翻转会导致接下来的感知器可能发生复杂的完全没法控制的变化，这样会导致我们的网络很难得到最终的逼近结果。





# 感知器网络理解以及S型神经元

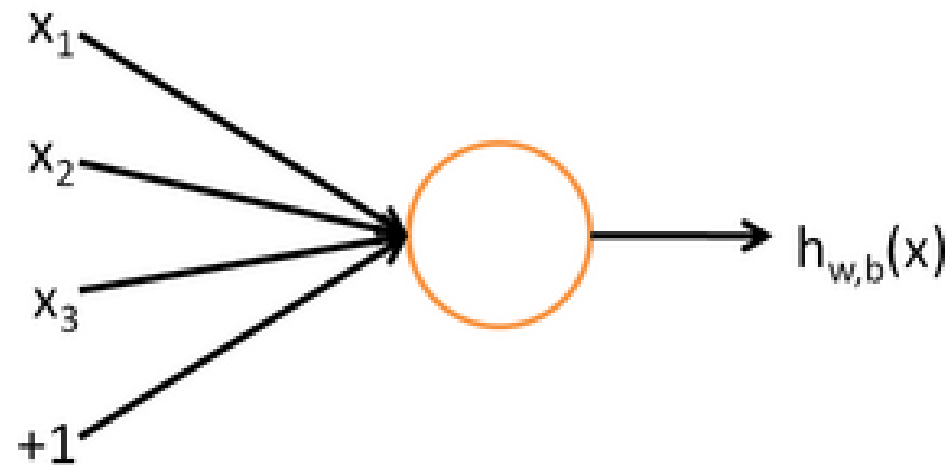
- 针对感知器网络的这种很难学习的问题，引入S型神经元来代替感知器，从而解决这个问题。
- 从感知器模型中，我们可以将单个神经元的计算过程看成下列两个步骤：
  - 先计算权重 $w$ 和输入值 $x$ 以及偏置项 $b$ 之间的线性结果值 $z$ ： $z=wx+b$
  - 然后对结果值 $z$ 进行一个数据的sign函数(变种)转换，得到一个离散的0/1值： $y=\text{int}((\text{sign}(z)+1)/2)$
- 在S型神经元中，和感知器神经元的区别在于：
  - 对于结果值 $z$ 的转换，采用的不是sign函数进行转换，是采用平滑类型的函数进行转换，让输出的结果值 $y$ 最终是一个连续的，S型神经元转指使用的是sigmoid函数。

## 神经网络来源之“神经元”

- 输出：函数 $h_{w,b}(x)$ ，其中 $w$ 权重和 $b$ 偏置项是参数
  - 输入： $x_1$ 、 $x_2$ 、 $x_3$ 和截距+1

$$h_{w,b}(x) = f(W^T x, b) = f\left(\sum_{i=1}^3 W_i x_i + b\right)$$

- 注意：函数 $f$ 被称为“**激活函数**”；常用/最好出现激活函数有  
sigmoid(逻辑回归函数)和tanh(双曲正切函数)



$$\tanh(z) = f(z) = \frac{e^z - e^{-z}}{e^z + e^{-z}}; \quad f'(z) = 1 - (f(z))^2$$

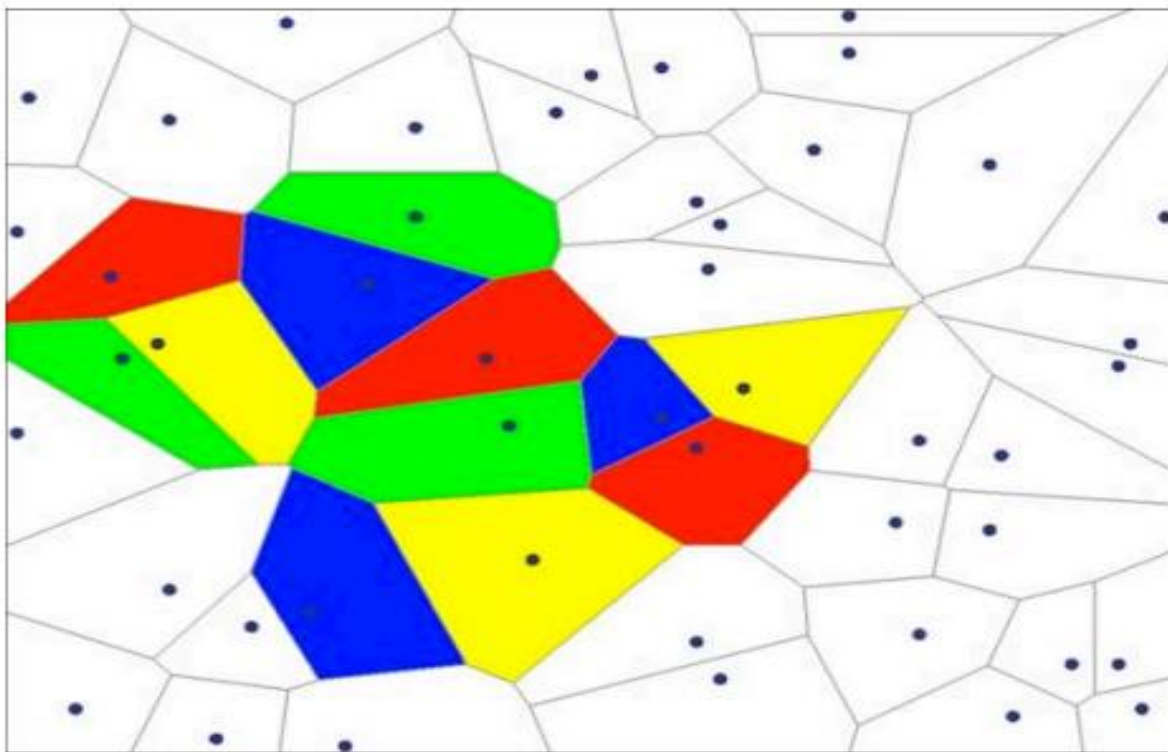
$$\text{sigmoid}(z) = f(z) = \frac{1}{1 + e^{-z}}; \quad f'(z) = f(z)(1 - f(z))$$

# 激活函数


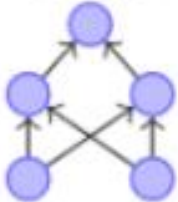


- 激活函数的主要作用是提供网络的非线性建模能力。如果没有激活函数，那么该网络仅能够表达线性映射，此时即便有再多的隐藏层，其整个网络跟单层神经网络也是等价的。因此也可以认为，只有加入了激活函数之后，深度神经网络才具备了分层的非线性映射学习能力。激活函数的主要特性是：可微性、单调性、输出值的范围；
- 常见的**激活函数**：Sign函数、Sigmoid函数、Tanh函数、ReLU函数、P-ReLU函数、Leaky-ReLU函数、ELU函数、Maxout函数等

## 神经网络之非线性可分

- 对线性分类器的**与**和**或**的组合可以完成非线性可分的问题；即通过多层的神经网络中加入激活函数的方式可以解决非线性可分的问题。

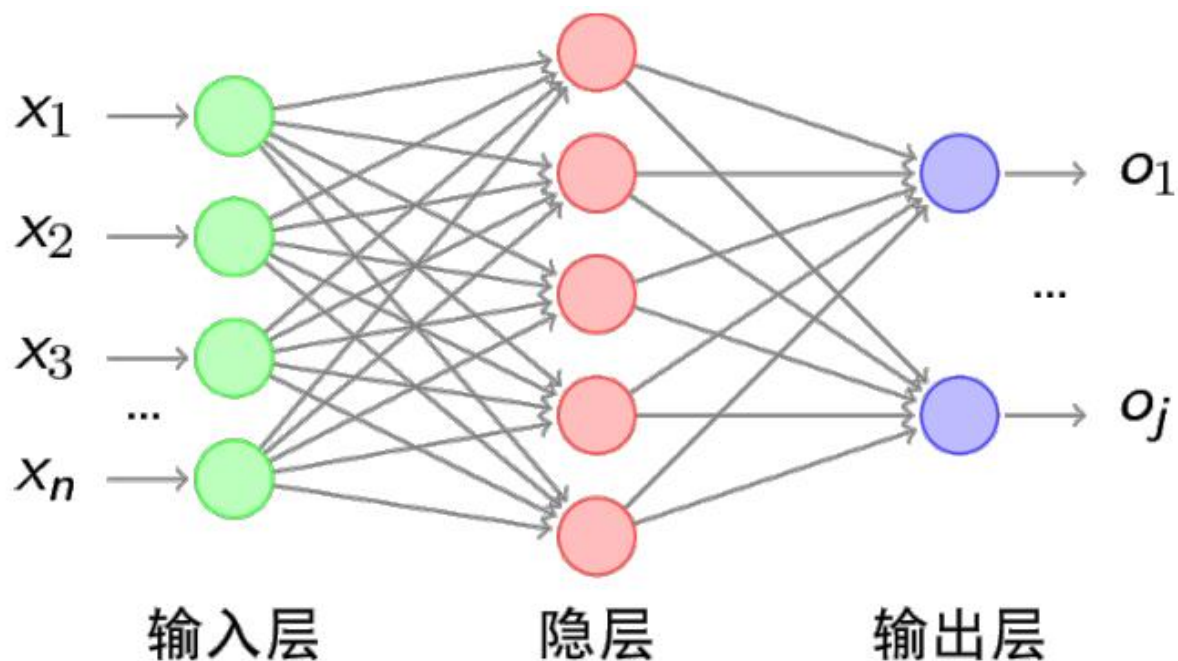


# 神经网络

结构	决策区域类型	区域形状	异或问题
无隐层 	由一超平面分成两个		
单隐层 	开凸区域或闭凸区域		
双隐层 	任意形状（其复杂度由单元数目确定）		

## 神经网络之BP算法

- BP算法也叫做 $\delta$ 算法
- 以三层的感知器为例（假定现在隐层和输出层均存在相同类型的激活函数）



## 神经网络之BP算法

- 输出层误差

$$E = \frac{1}{2} (d - O)^2 = \frac{1}{2} \sum_{k=1}^{\ell} (d_k - O_k)^2$$

- 隐层的误差

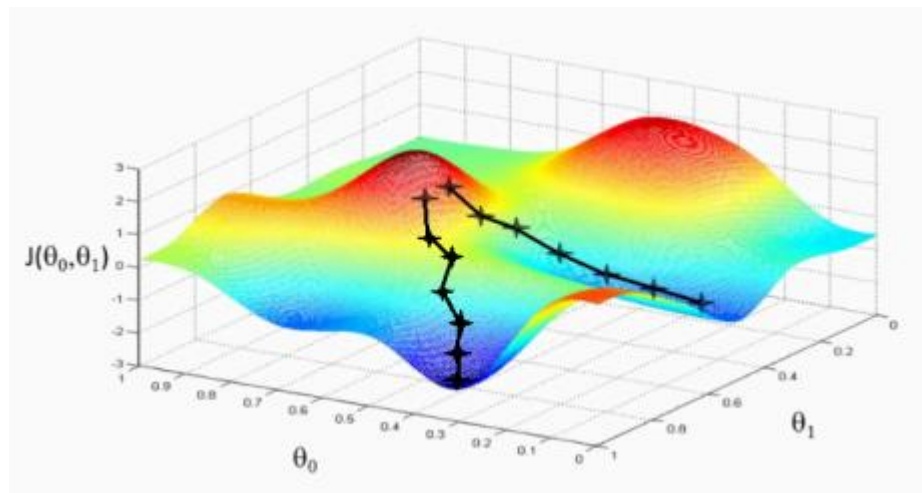
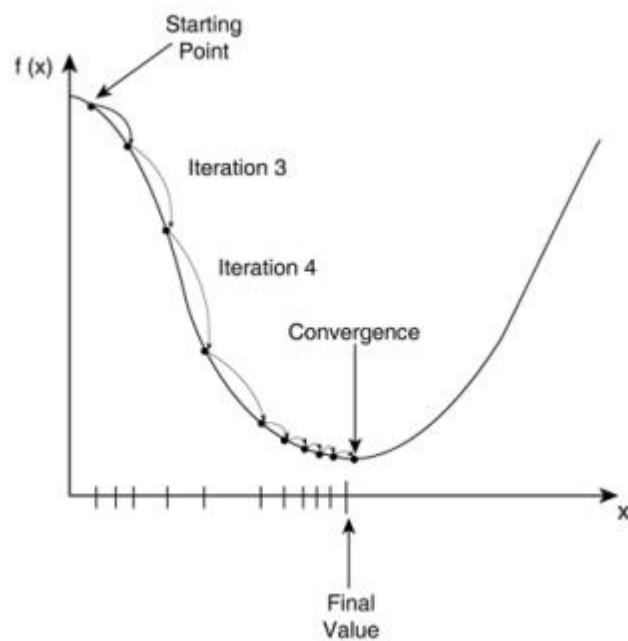
$$E = \frac{1}{2} \sum_{k=1}^{\ell} (d_k - f(net_k))^2 = \frac{1}{2} \sum_{k=1}^{\ell} \left( d_k - f \left( \sum_{j=1}^m w_{jk} y_j \right) \right)^2$$

- 输入层误差

$$E = \frac{1}{2} \sum_{k=1}^{\ell} \left( d_k - f \left[ \sum_{j=0}^m w_{jk} f(net_j) \right] \right)^2 = \frac{1}{2} \sum_{k=1}^{\ell} \left( d_k - f \left[ \sum_{j=0}^m w_{jk} f \left( \sum_{i=1}^n v_{ij} x_i \right) \right] \right)^2$$

# 神经网络之SGD

- 误差E有了，那么为了使误差越来越小，可以采用随机梯度下降的方式进行 $\omega$ 和 $v$ 的求解，即求得 $\omega$ 和 $v$ 使得误差E最小。



$$\Delta \omega_{j\kappa} = -\eta \frac{\partial E}{\partial \omega_{j\kappa}} \quad j = 0, 1, 2, \dots, m; \quad \kappa = 1, 2, \dots, \ell$$

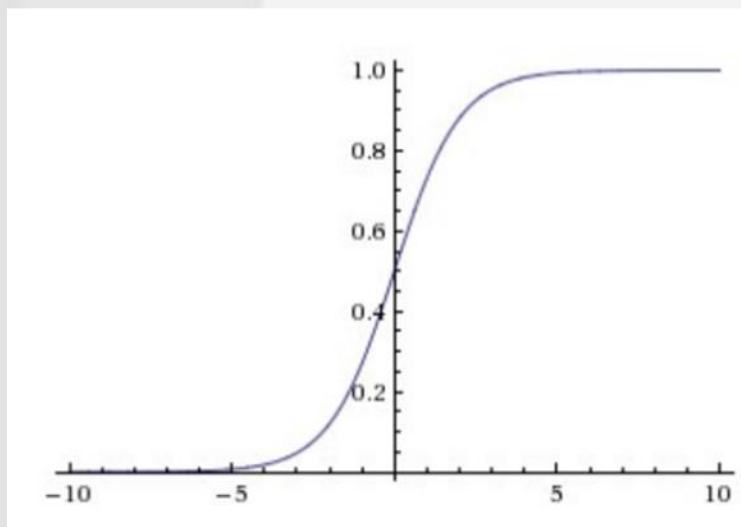
$$\Delta v_{ij} = -\eta \frac{\partial E}{\partial v_{ij}} \quad i = 0, 1, 2, \dots, n; \quad j = 1, 2, \dots, m$$



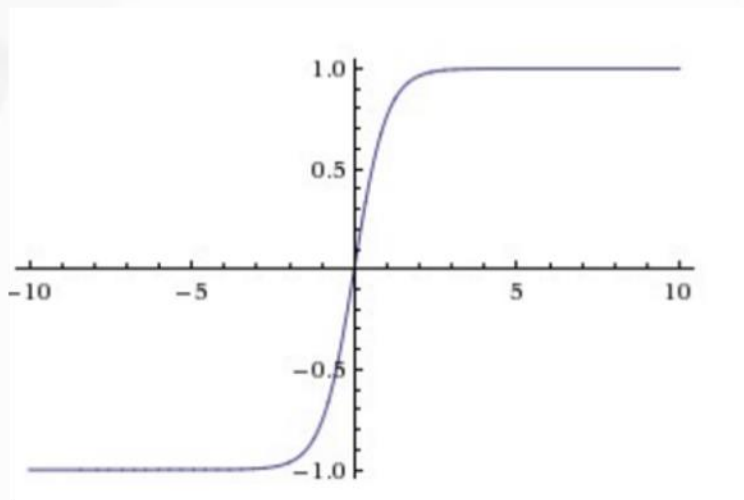
# 激活函数

激活函数的作用就是把组合函数得到的结果再进行一次非线性的变换

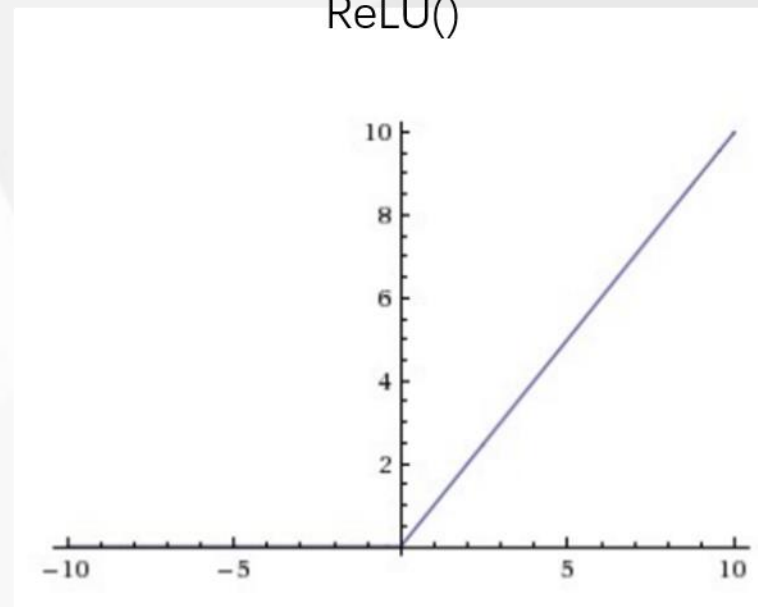
Sigmoid()



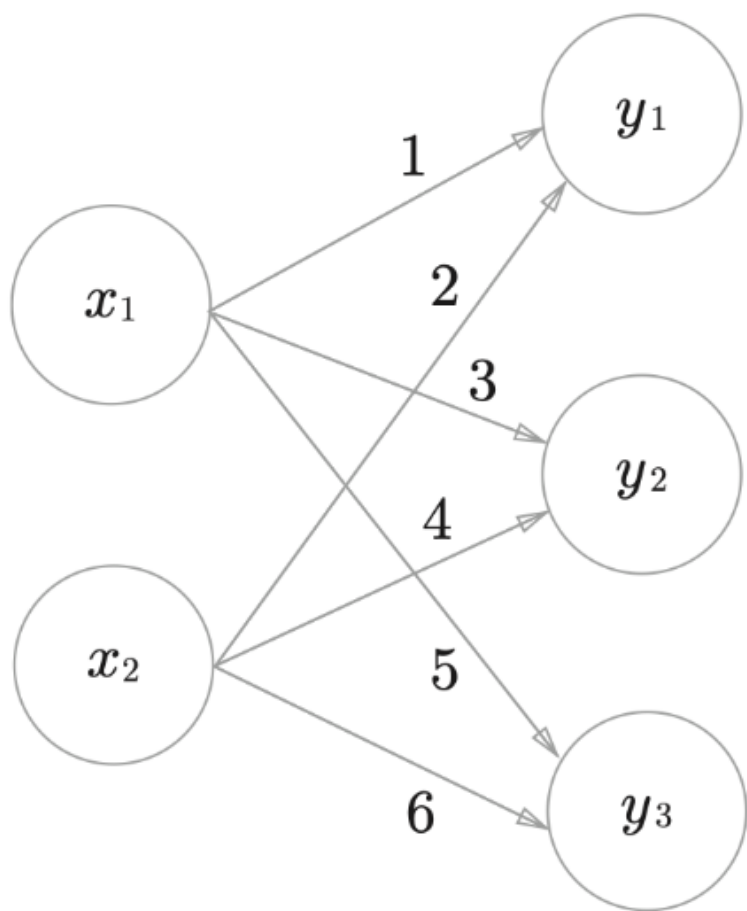
Tanh()



ReLU()



# 神经网络与矩阵乘法



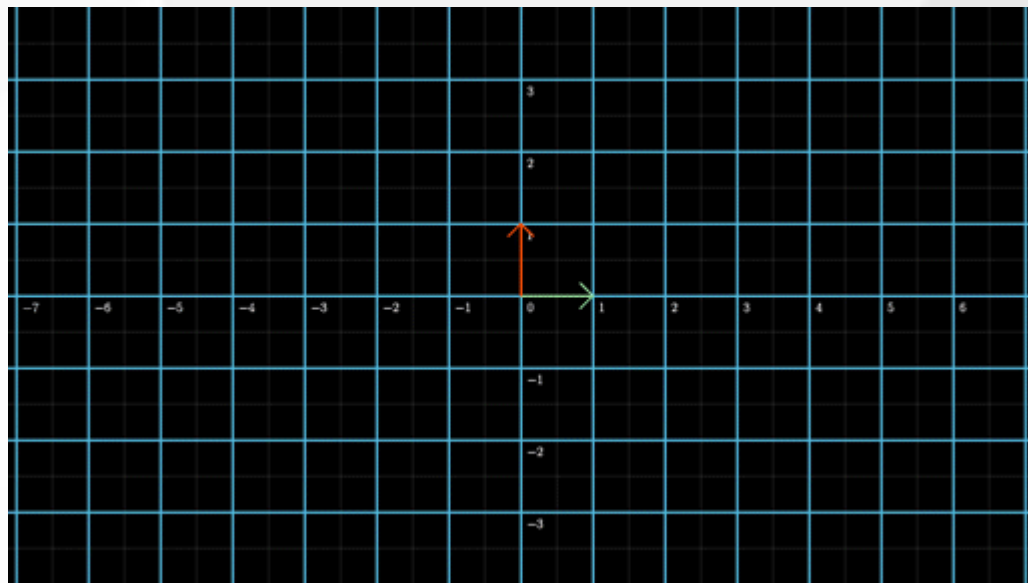
$$\begin{matrix} \mathbf{X} & \mathbf{W} & = & \mathbf{Y} \\ 2 & 2 \times 3 & & 3 \\ \hline & \text{一致} & & \end{matrix}$$

The diagram illustrates the matrix multiplication  $\mathbf{X}\mathbf{W} = \mathbf{Y}$ . The input matrix  $\mathbf{X}$  has a dimension of 2, the weight matrix  $\mathbf{W}$  has a dimension of  $2 \times 3$ , and the output matrix  $\mathbf{Y}$  has a dimension of 3. The dimensions are consistent, as indicated by the label "一致" (consistent) under the  $2$  and  $2 \times 3$  terms.

在这张动图的开始的阶段，绿色向量代表  $\begin{bmatrix} 0 \\ 1 \end{bmatrix}$

而红色向量代表  $\begin{bmatrix} 1 \\ 0 \end{bmatrix}$

我们盯住这两个基向量，观察到在动图的末尾，这两个向量分别落在了  $\begin{bmatrix} 1 \\ -2 \end{bmatrix}$  和  $\begin{bmatrix} 3 \\ 0 \end{bmatrix}$  那么，这两个基向量组成的坐标系也随着这两个基向量的变换而线性变换，形成了动图末尾中蓝色直线组成的二维坐标。假设经历了上图的坐标变换，原来的向量  $\begin{bmatrix} -1 \\ 2 \end{bmatrix}$  现在到何处了呢？



原来：  $\begin{bmatrix} -1 \\ 2 \end{bmatrix} = -1 * \begin{bmatrix} 1 \\ 0 \end{bmatrix} + 2 * \begin{bmatrix} 0 \\ 1 \end{bmatrix}$  变换后：  $\begin{bmatrix} 5 \\ 2 \end{bmatrix} = -1 * \begin{bmatrix} 1 \\ -2 \end{bmatrix} + 2 * \begin{bmatrix} 3 \\ 0 \end{bmatrix}$

这其中的区别就是基向量不一样了，而线性组合的系数 -1 与 2 保持固定不变。

我们把变换后的基向量放在一起，变为矩阵：

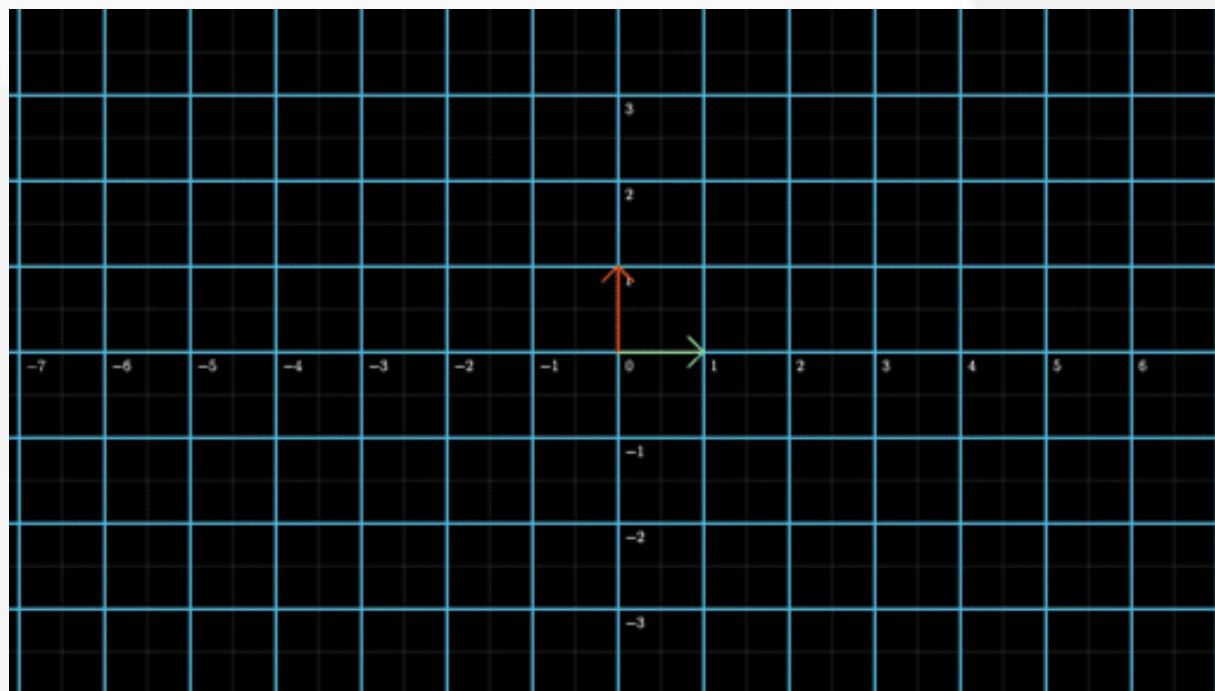
$$\left[ \begin{bmatrix} 1 \\ -2 \end{bmatrix} \quad \begin{bmatrix} 3 \\ 0 \end{bmatrix} \right] \rightarrow \begin{bmatrix} 1 & 3 \\ -2 & 0 \end{bmatrix}$$

这就是矩阵的由来，其实质就是将坐标整体线性变换。向量  $\begin{bmatrix} -1 \\ 2 \end{bmatrix}$  在经过线性变换  $\begin{bmatrix} 1 & 3 \\ -2 & 0 \end{bmatrix}$

变为向量  $\begin{bmatrix} 5 \\ 2 \end{bmatrix}$  表示形式为：

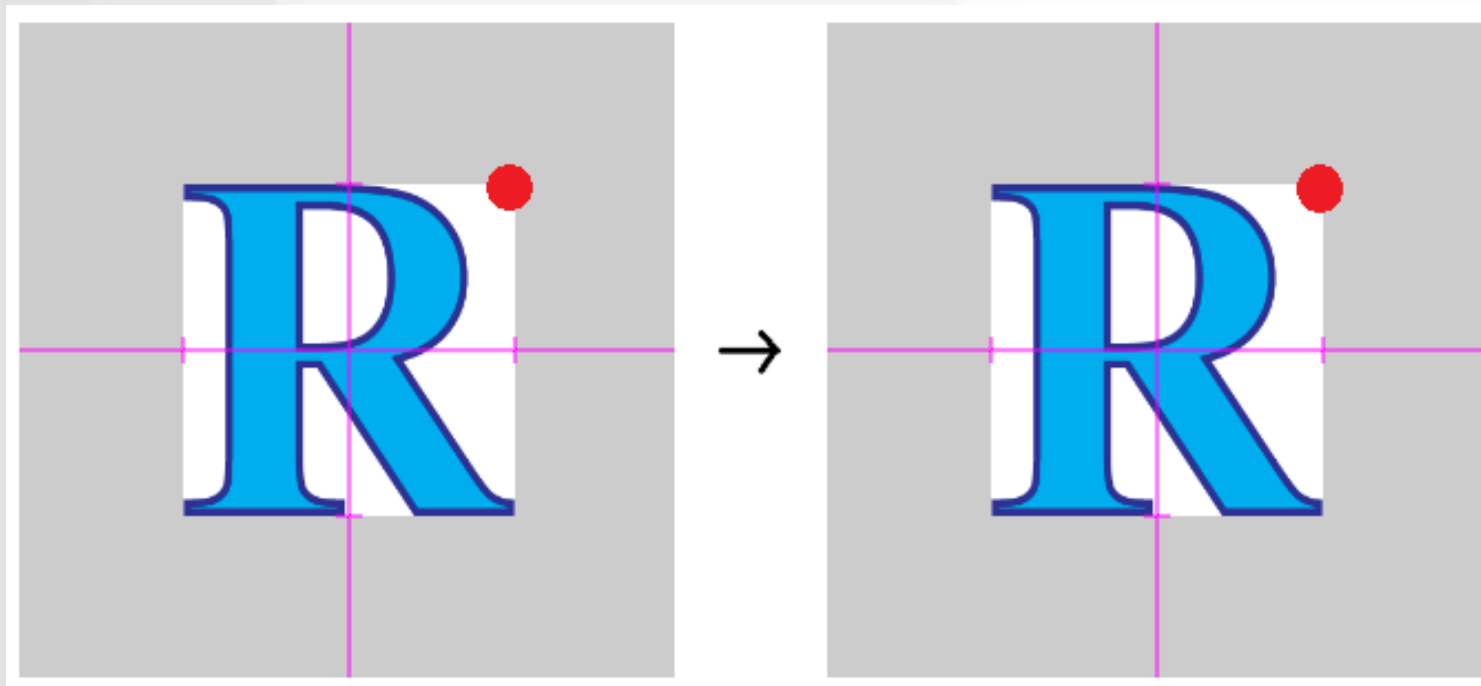
$$\begin{bmatrix} 1 & 3 \\ -2 & 0 \end{bmatrix} \begin{bmatrix} -1 \\ 2 \end{bmatrix} = \begin{bmatrix} 5 \\ 2 \end{bmatrix} \text{ (注意：这里的表示顺序为变换矩阵在左，向量为列向量在右侧。)}$$

## 练习



# 线性变换

## 单位矩阵表示恒等变换(identity)

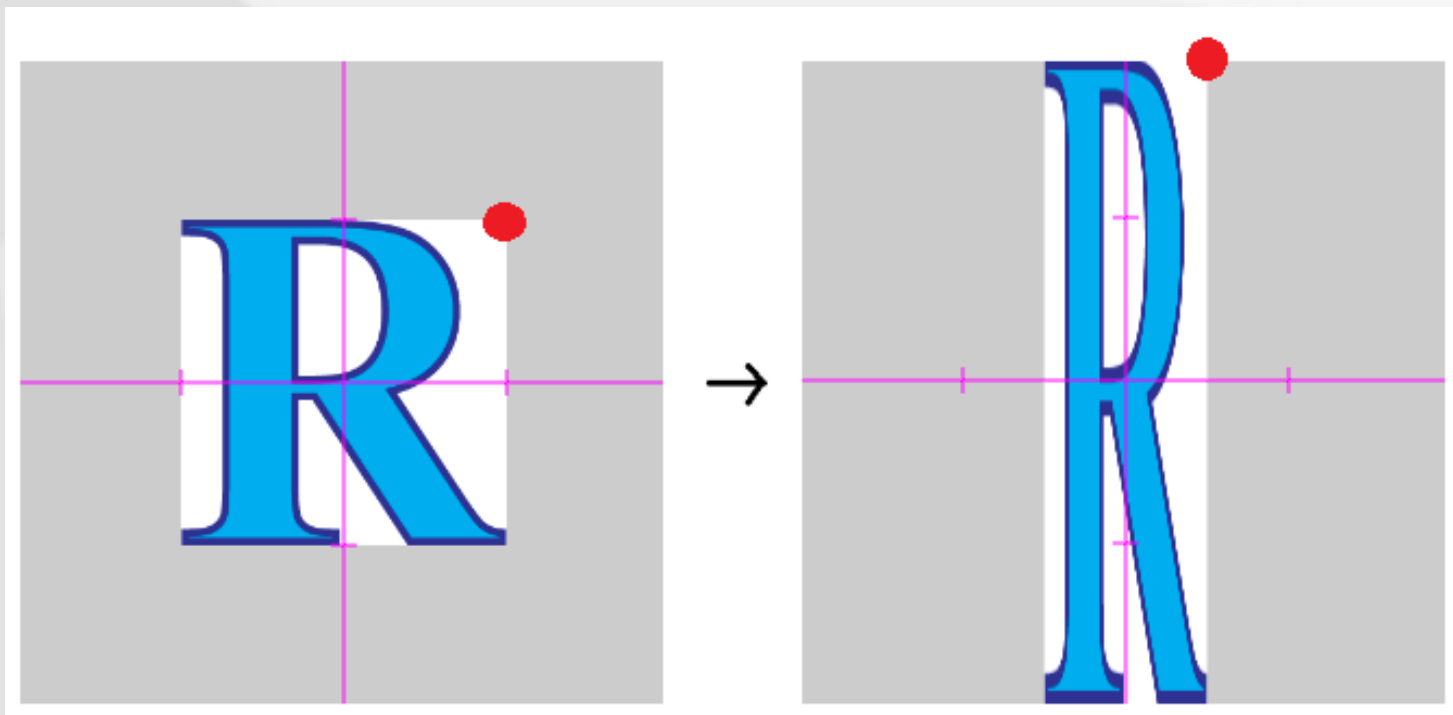


矩阵:  $I = \begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix}$

变换:  $\begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix} \begin{bmatrix} x \\ y \end{bmatrix}$

对右上角的点进行变换:  $\begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix} \begin{bmatrix} 1 \\ 1 \end{bmatrix} = \begin{bmatrix} 1 \\ 1 \end{bmatrix}$

## 对角矩阵表示伸缩变换(scale)

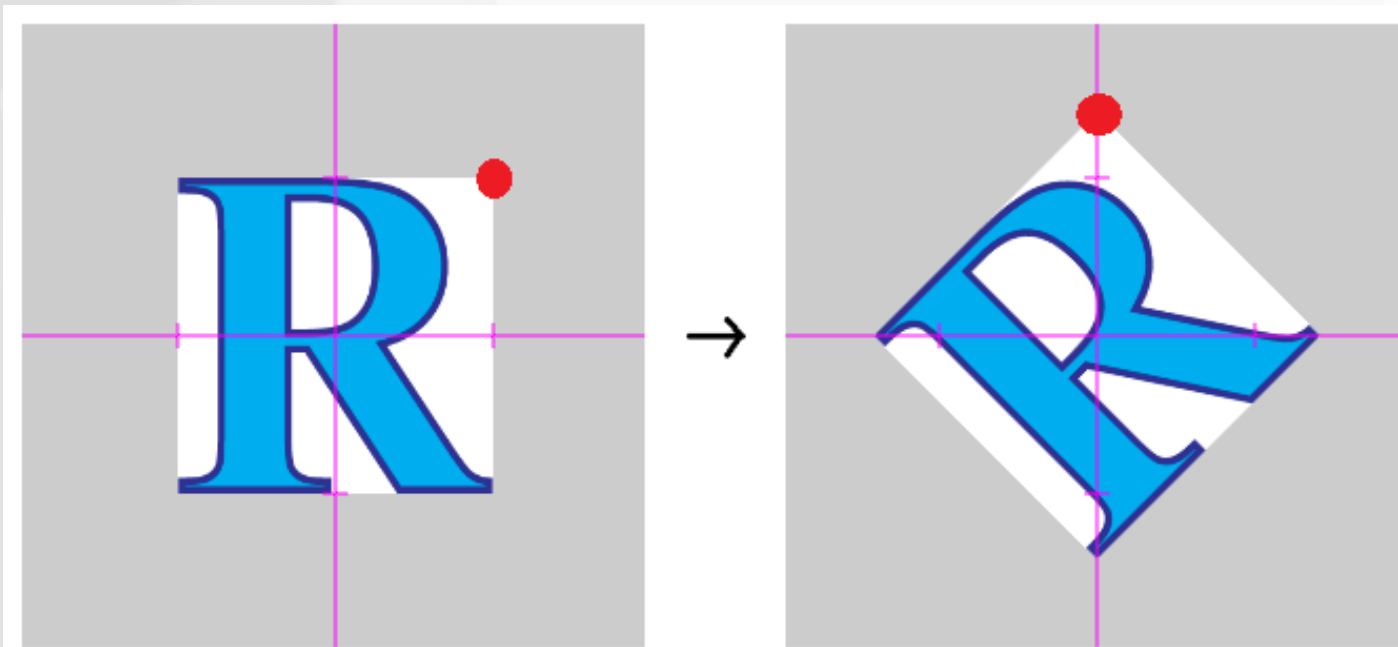


矩阵: 
$$S = \begin{bmatrix} s_x & 0 \\ 0 & s_y \end{bmatrix}$$

变换: 
$$\begin{bmatrix} 1/2 & 0 \\ 0 & 2 \end{bmatrix} \begin{bmatrix} x \\ y \end{bmatrix}$$

对右上角的  
点进行变换: 
$$\begin{bmatrix} 1/2 & 0 \\ 0 & 2 \end{bmatrix} \begin{bmatrix} 1 \\ 1 \end{bmatrix} = \begin{bmatrix} 1/2 \\ 2 \end{bmatrix}$$

## 旋转变换(rotation)(了解)(旋转变换是线性变换)



矩阵:  $R_{\alpha} = \begin{bmatrix} \cos(\alpha) & -\sin(\alpha) \\ \sin(\alpha) & \cos(\alpha) \end{bmatrix}$

变换:  $\begin{bmatrix} \cos(\frac{\pi}{4}) & -\sin(\frac{\pi}{4}) \\ \sin(\frac{\pi}{4}) & \cos(\frac{\pi}{4}) \end{bmatrix} \begin{bmatrix} x \\ y \end{bmatrix}$

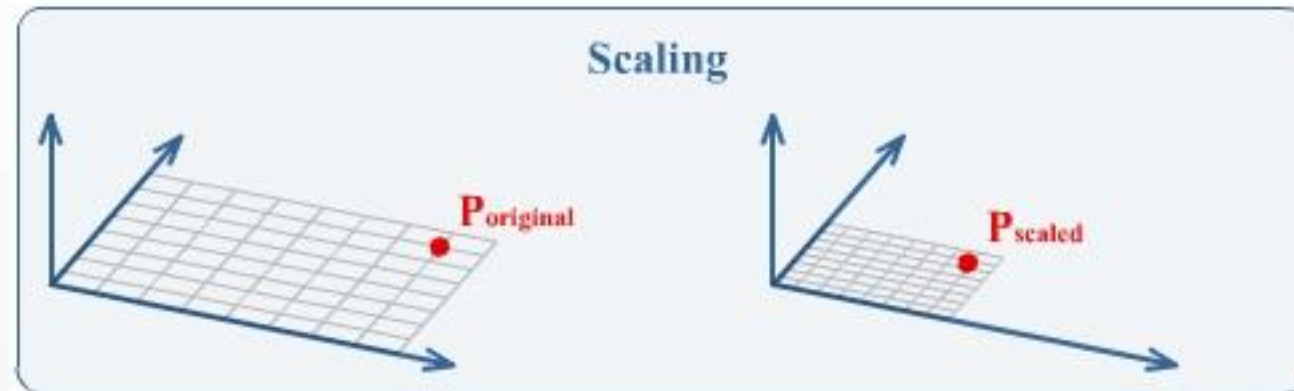
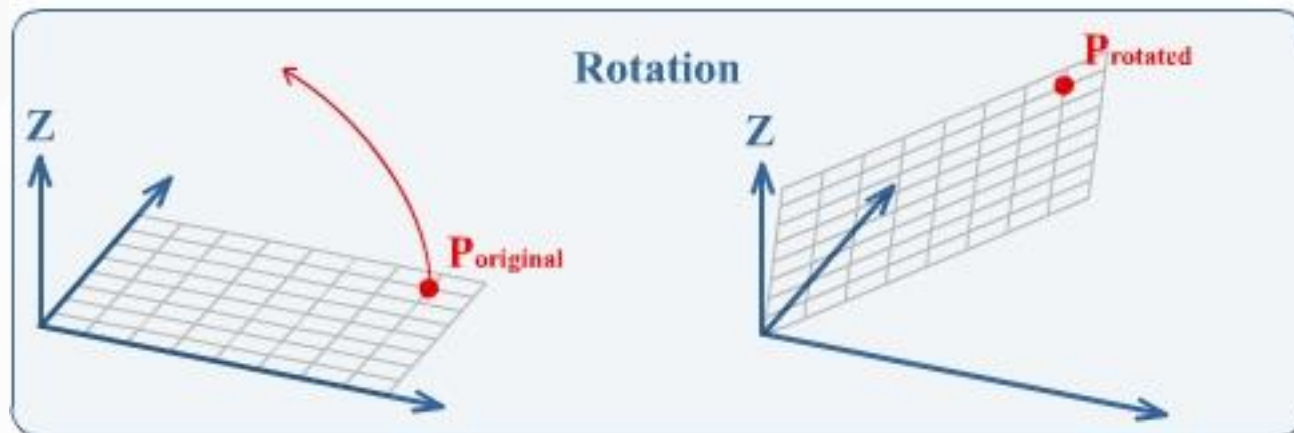
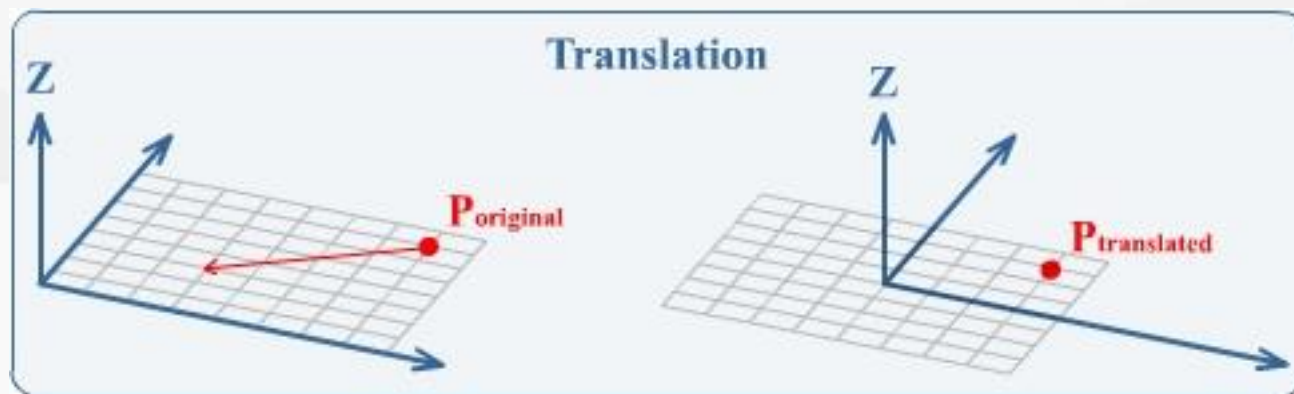
对右上角的  
点进行变换:  $\begin{bmatrix} \cos(\frac{\pi}{4}) & -\sin(\frac{\pi}{4}) \\ \sin(\frac{\pi}{4}) & \cos(\frac{\pi}{4}) \end{bmatrix} \begin{bmatrix} 1 \\ 1 \end{bmatrix} = \begin{bmatrix} 0 \\ \sqrt{2} \end{bmatrix}$

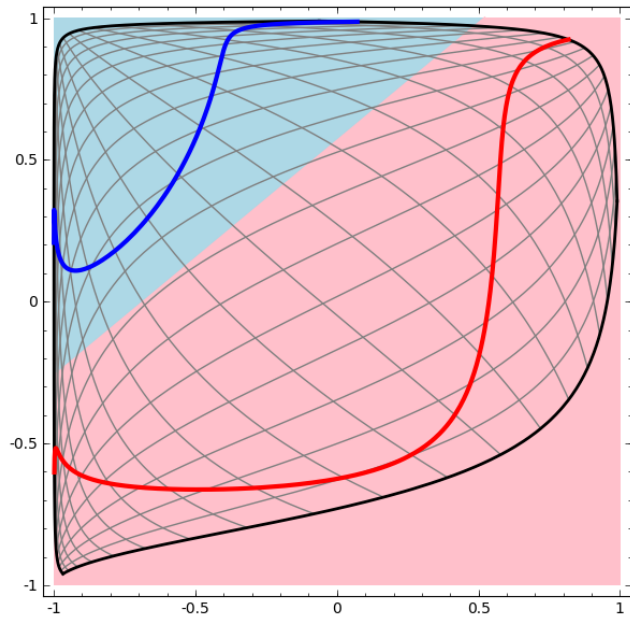
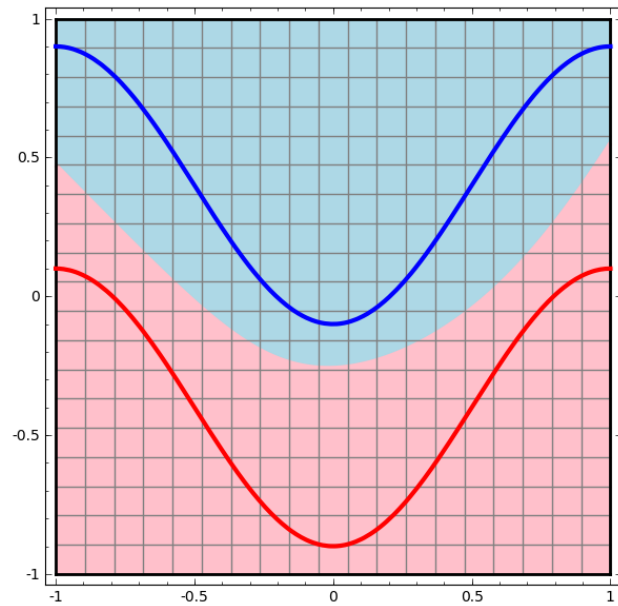
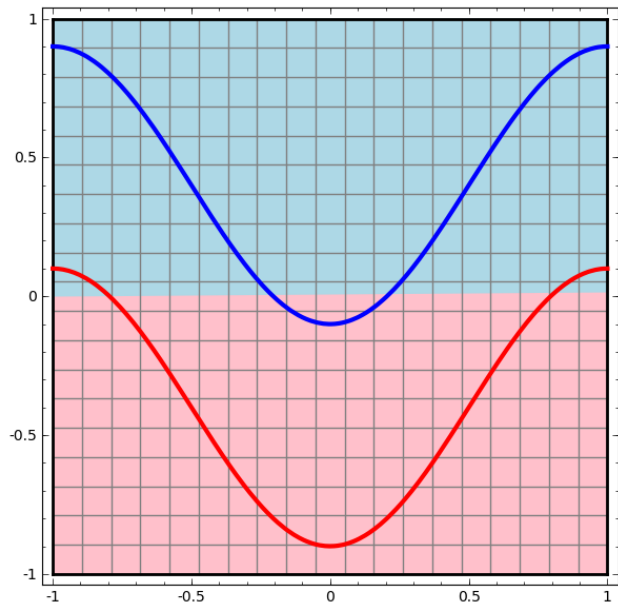
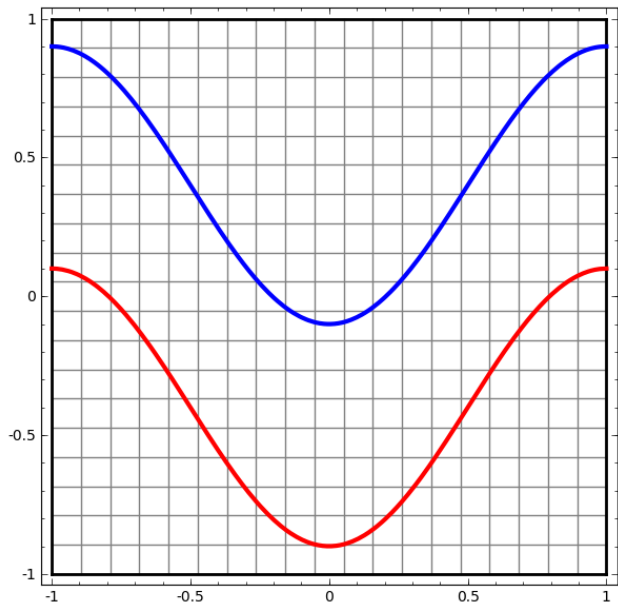


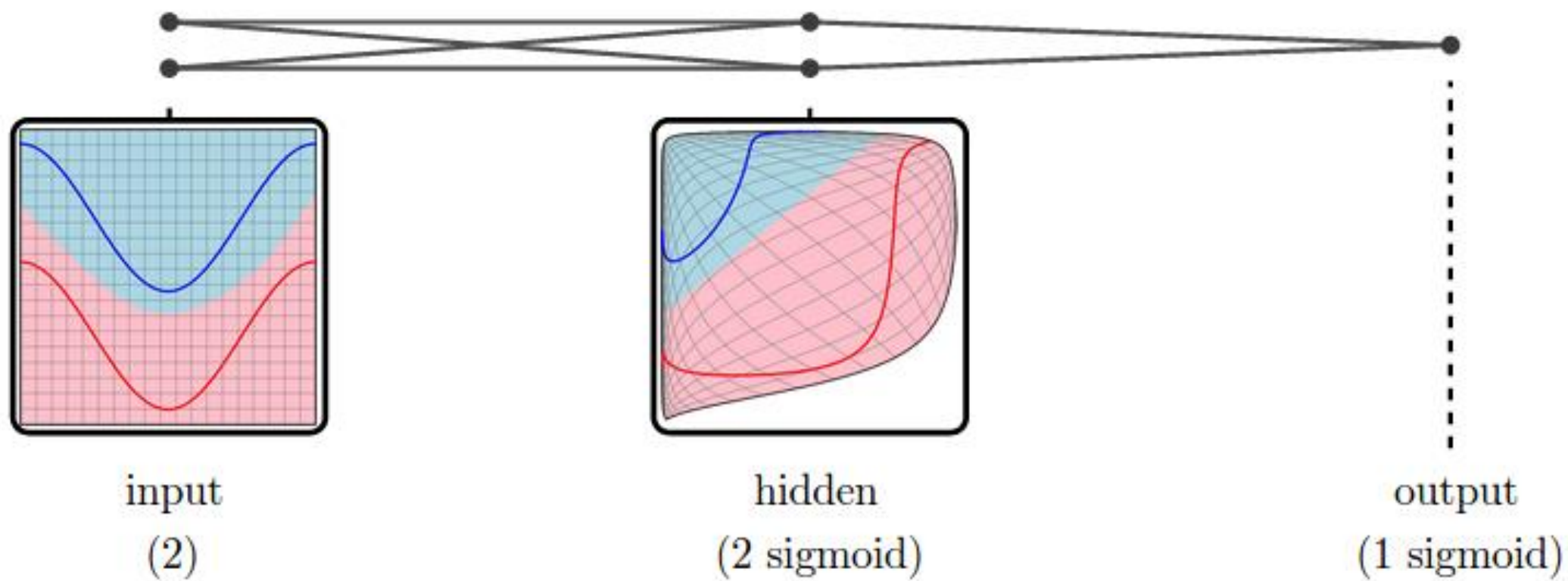
1) 我们有一个矩阵对应每个基本变换

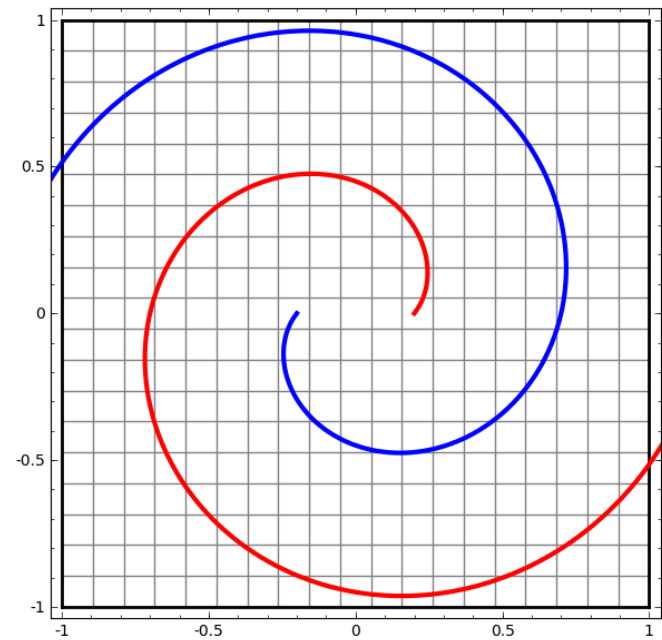
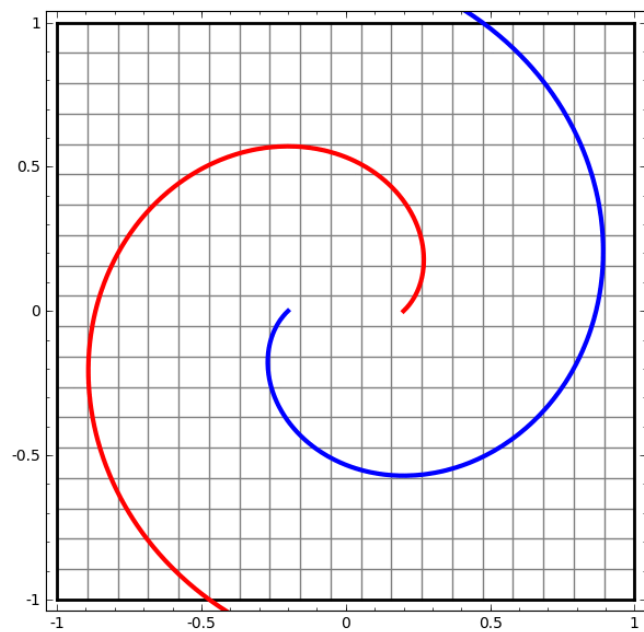
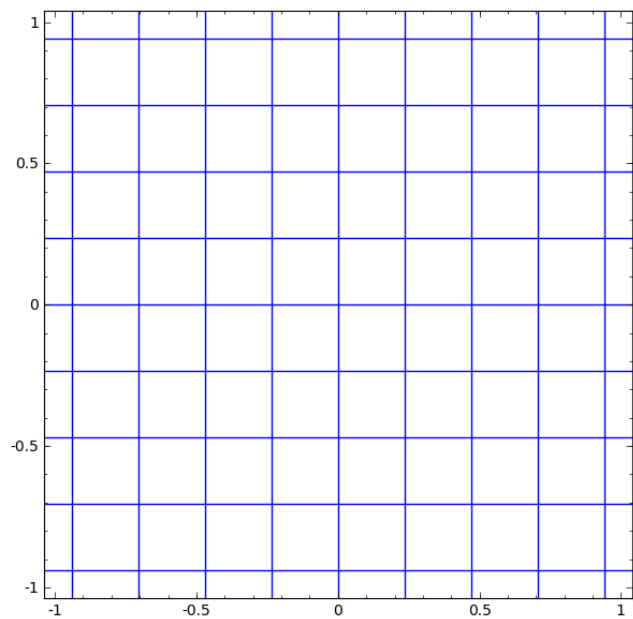
2) 将点的坐标乘以一个矩阵会获得变换后的坐标

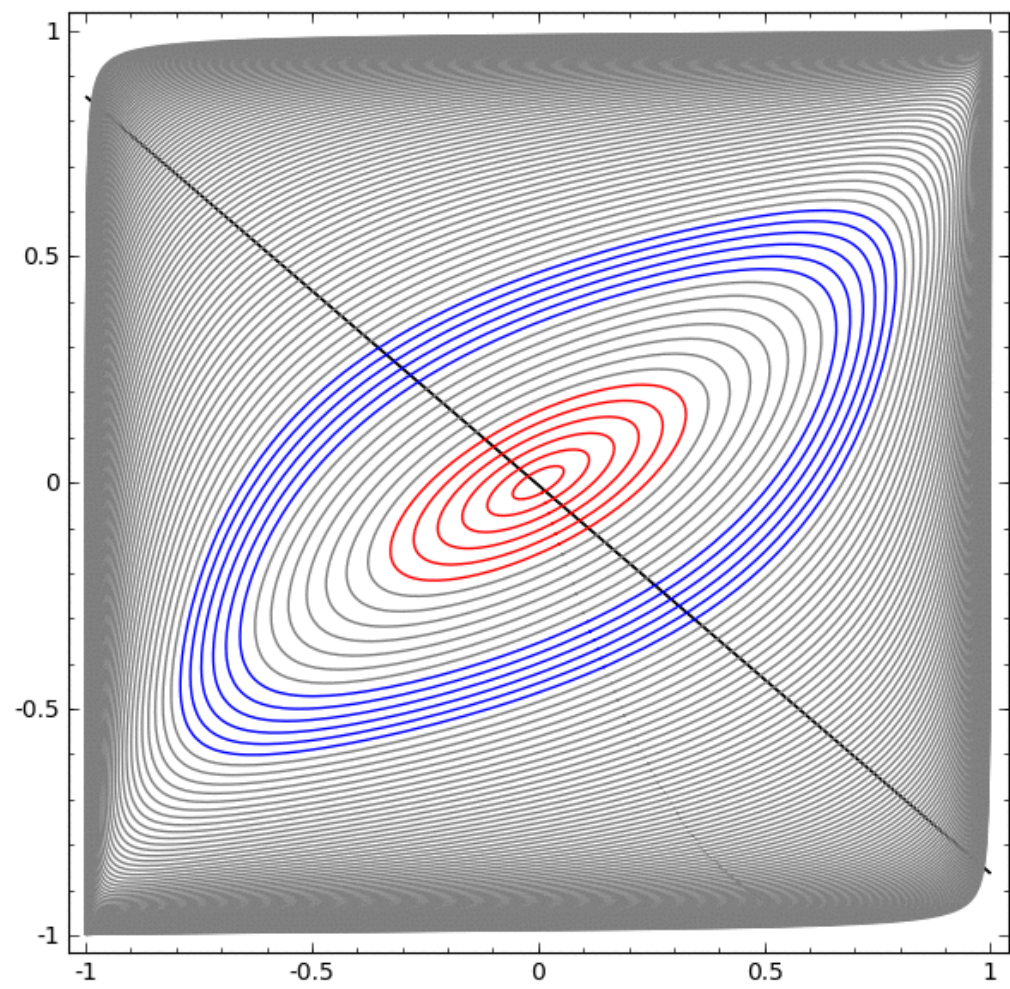
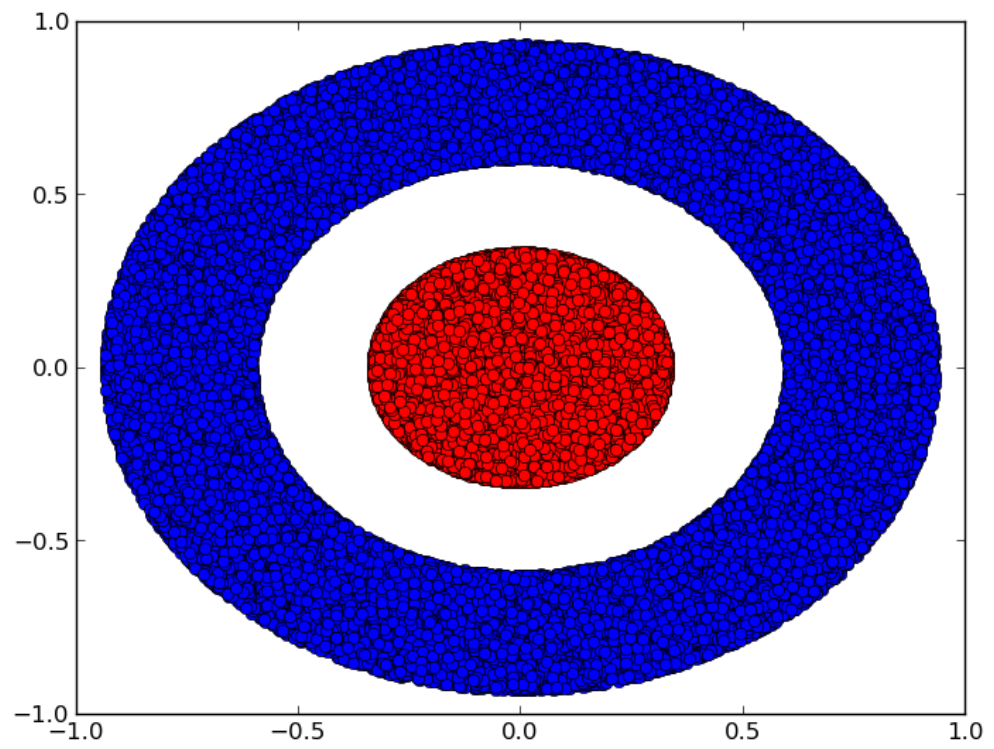
3) 两个矩阵相乘获得一个新矩阵，对应变换的组合



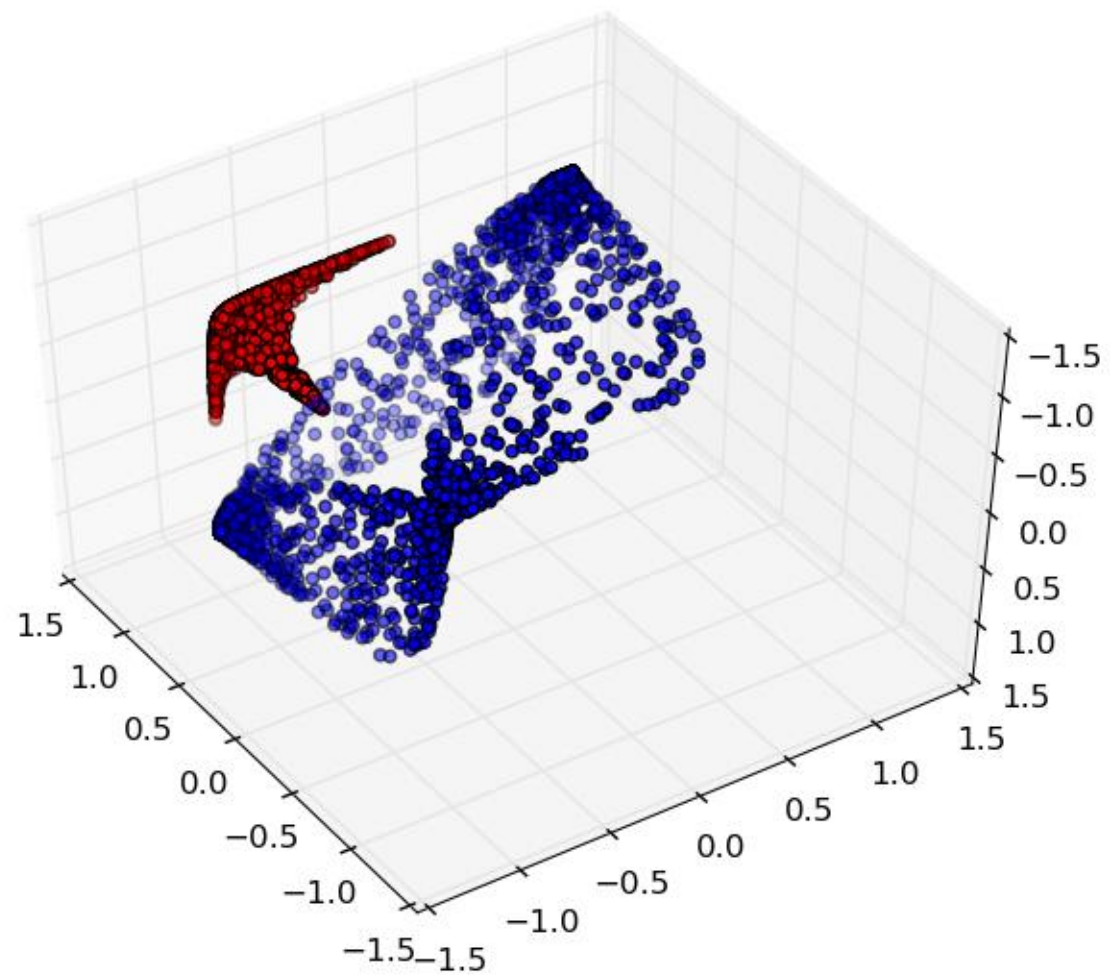




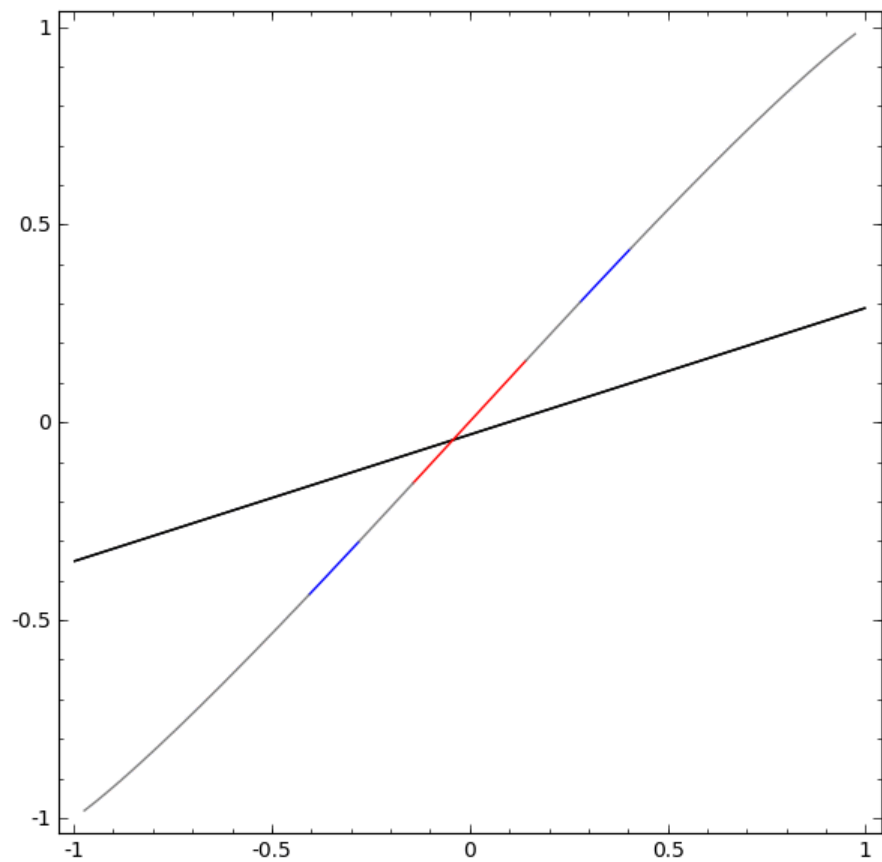
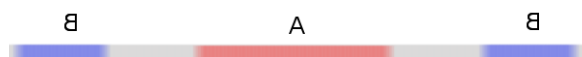




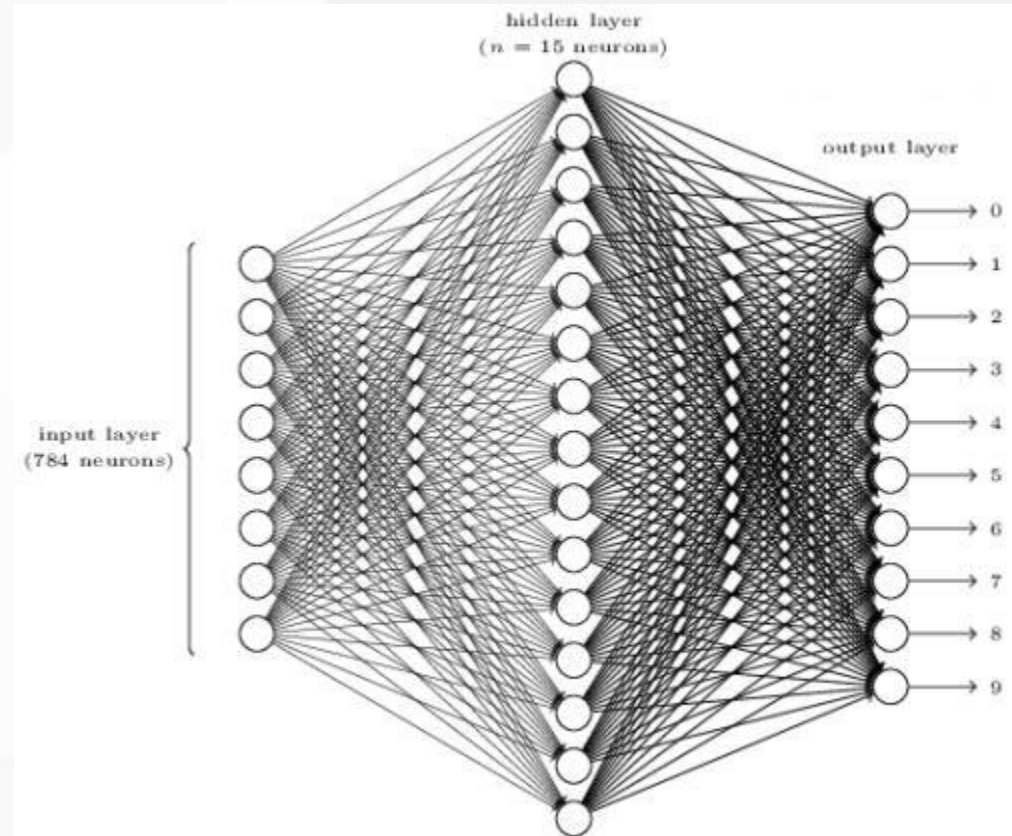
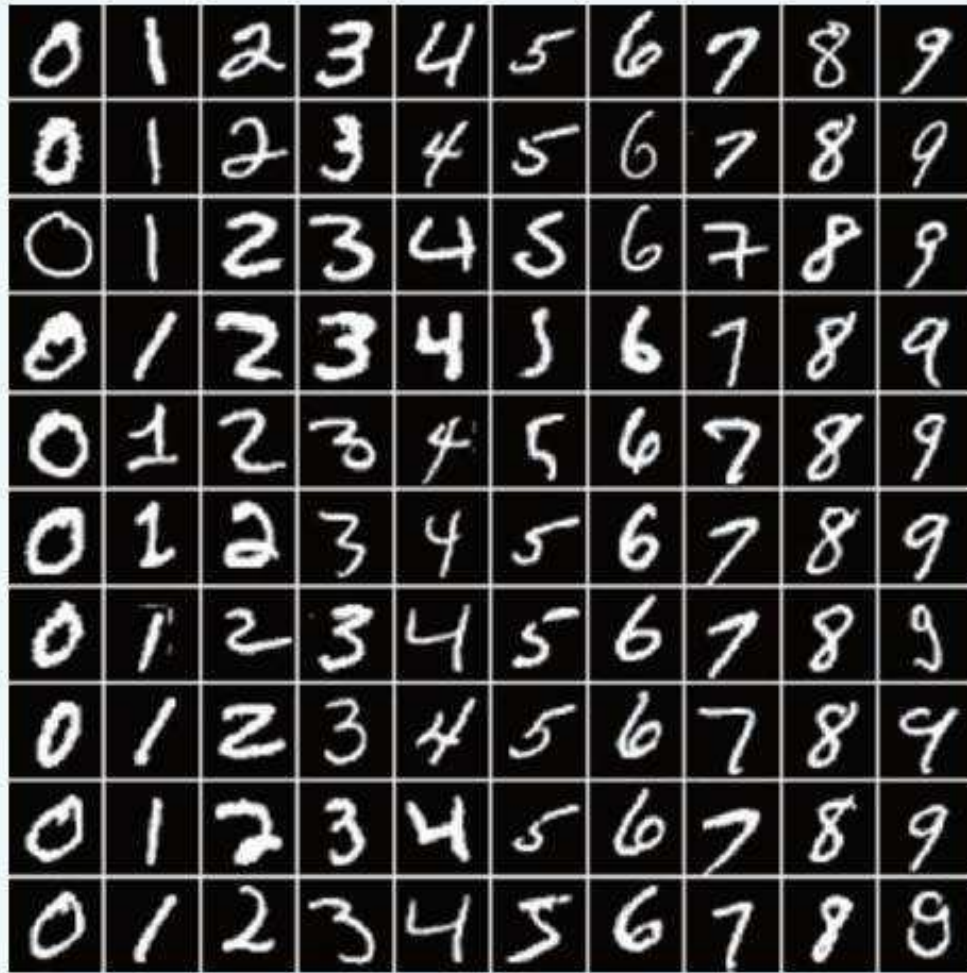




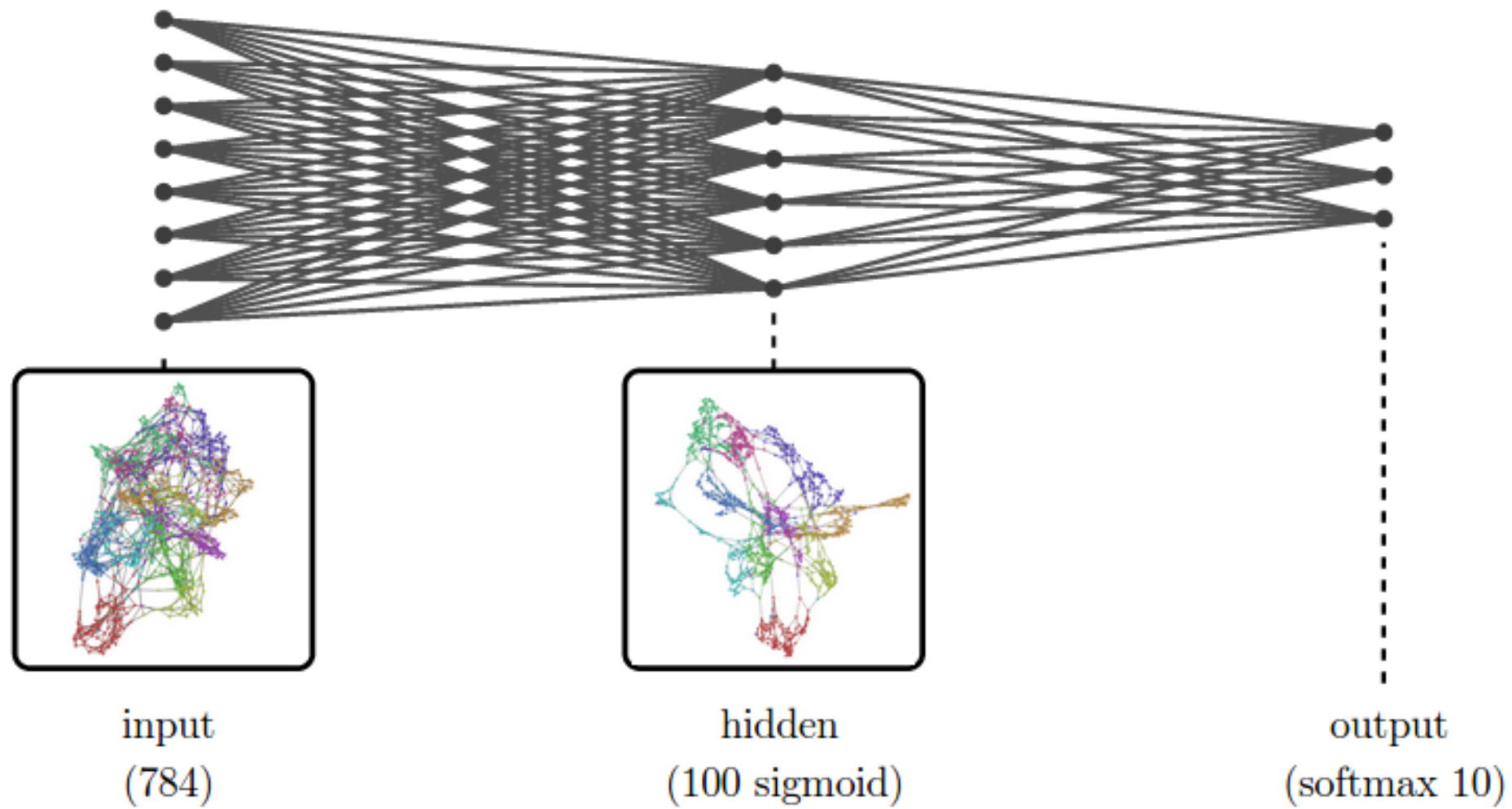
通过这种表示，我们可以使用超平面分离数据集。  
为了更好地理解，让我们考虑一个更简单的**1**维数据集：



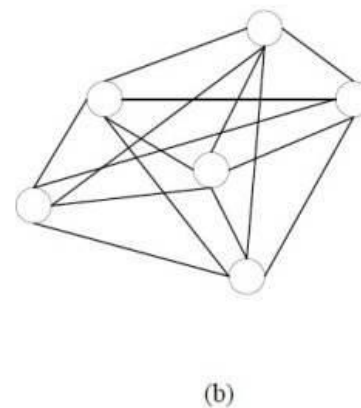
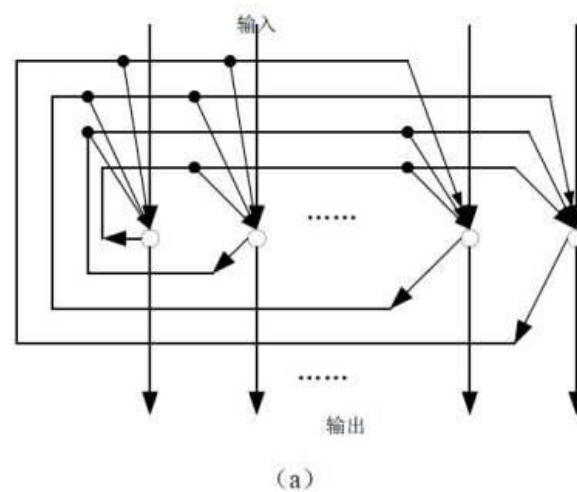
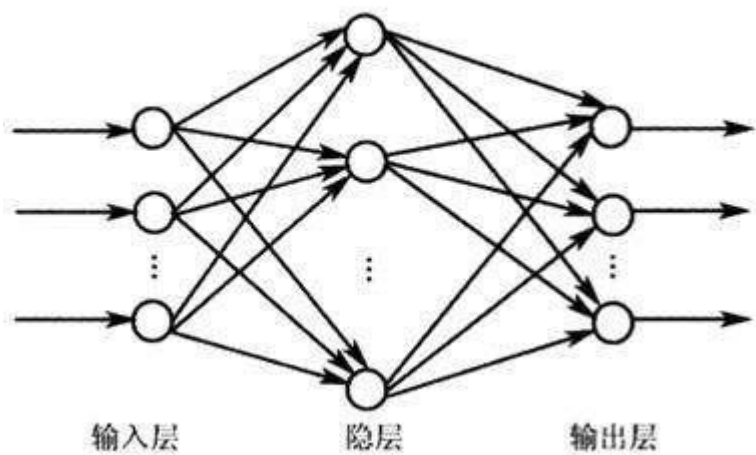
# MNIST dataset



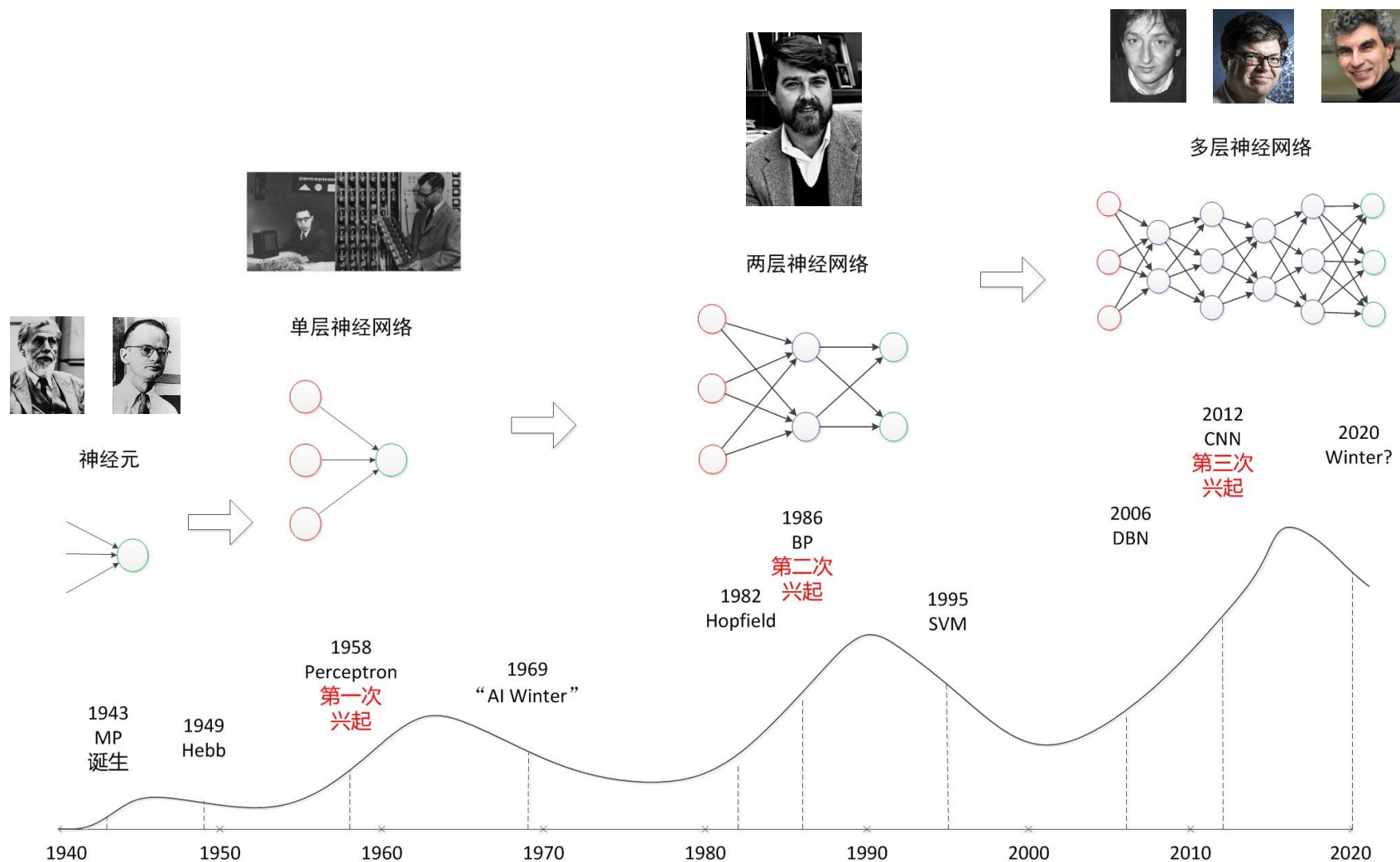




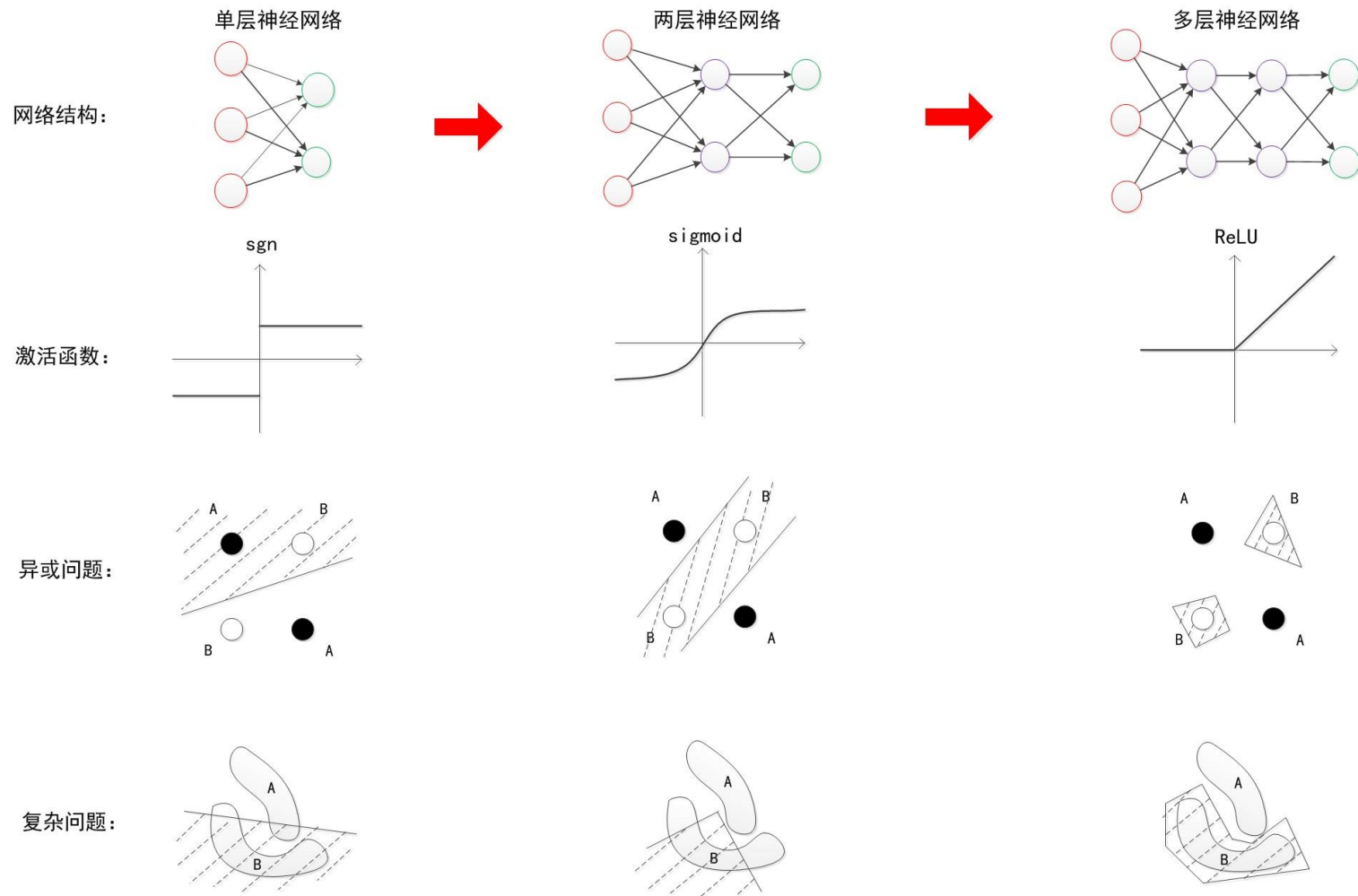
# 前馈神经网络vs反馈神经网络



# 神经网络发展历史

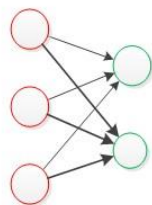


# 神经网络特征学习能力

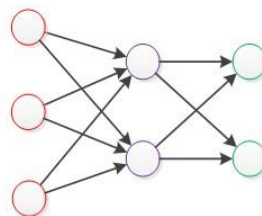


# 算力发展

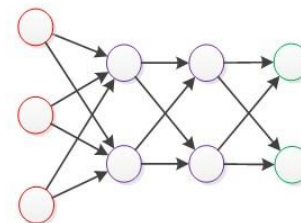
单层神经网络  
(60-70)



两层神经网络  
(85-95)



多层神经网络  
(2010-)



计算能力：

晶体管

CPU

集群或GPU

数据量：

1-10

1K-10K

1M-100M

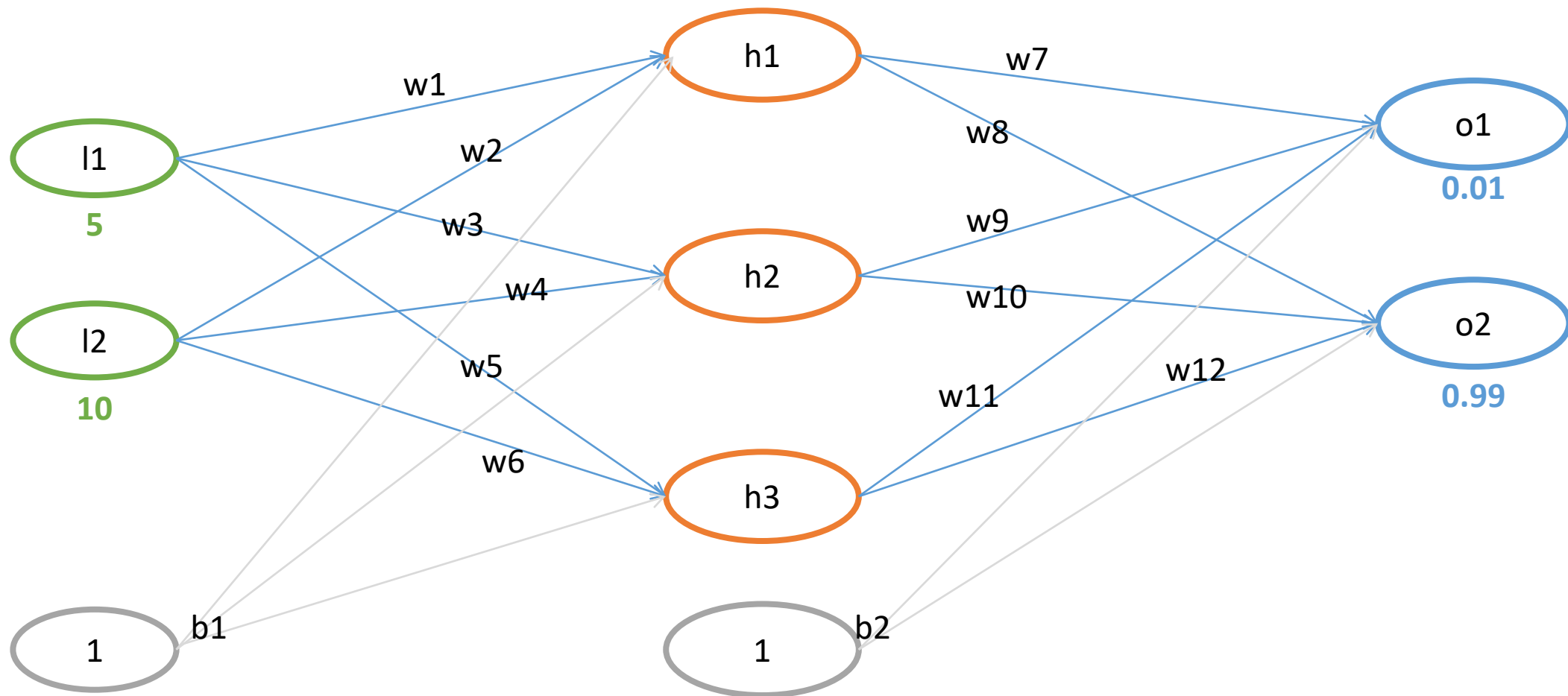
算法：

学习算法

BP算法

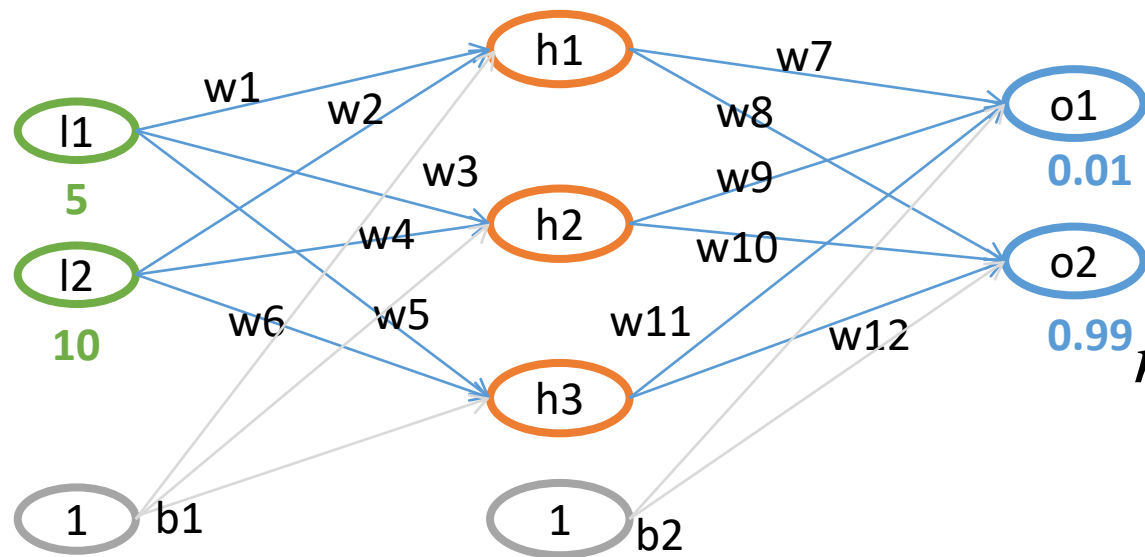
Pre-training,  
Dropout等方法

## BP算法例子



$$w = (0.1, 0.15, 0.2, 0.25, 0.3, 0.35, 0.4, 0.45, 0.5, 0.55, 0.6, 0.65)$$
$$b = (0.35, 0.65)$$

## BP算法例子-FP过程



$$b = (0.35, 0.65)$$

$$w = \begin{pmatrix} 0.1, 0.15, 0.2, 0.25, 0.3, 0.35, \\ 0.4, 0.45, 0.5, 0.55, 0.6, 0.65 \end{pmatrix}$$

$$net_{h1} = w_1 * l_1 + w_2 * l_2 + b_1 * 1$$

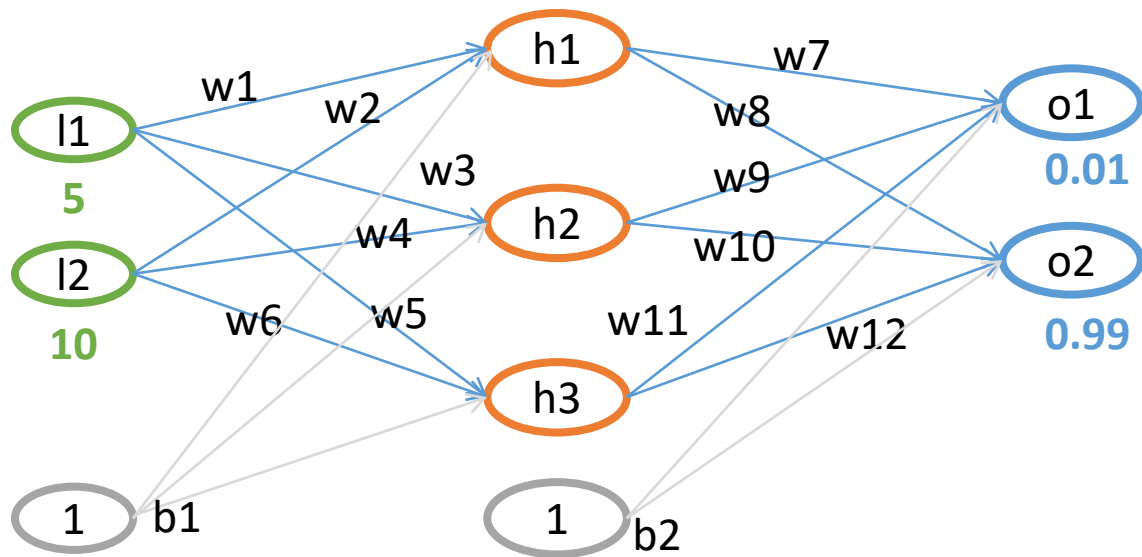
$$net_{h1} = 0.1 * 5 + 0.15 * 10 + 0.35 * 1 = 2.35$$

$$out_{h1} = \frac{1}{1 + e^{-net_{h1}}} = \frac{1}{1 + e^{-2.35}} = 0.912934$$

$$out_{h2} = 0.979164$$

$$out_{h3} = 0.995275$$

## BP算法例子-FP过程



$$net_{o1} = w_7 * out_{h1} + w_9 * out_{h2} + w_{11} * out_{h3} + b_2 * 1$$

$$net_{o1} = 0.4 * 0.912934 + 0.5 * 0.979164 + 0.6 * 0.995275 = 2.1019206$$

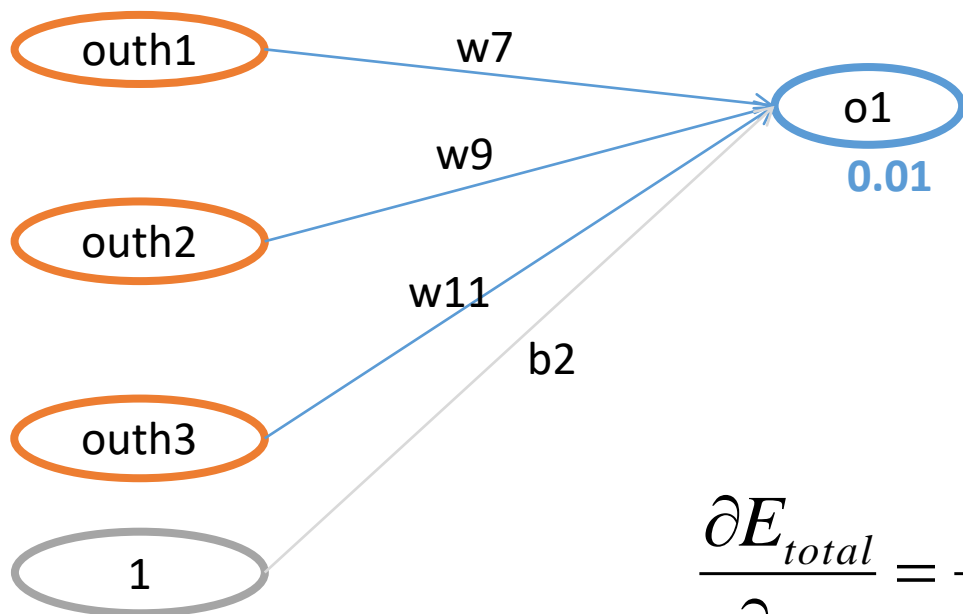
$$out_{o1} = \frac{1}{1 + e^{-net_{o1}}} = \frac{1}{1 + e^{-2.1019206}} = 0.891090$$

$$out_{o2} = 0.904330$$

$$E_{total} = E_{o1} + E_{o2} = \frac{1}{2} (0.01 - 0.891090)^2 + \frac{1}{2} (0.99 - 0.904330)^2 = 0.391829$$



## BP算法例子-BP过程(W7)



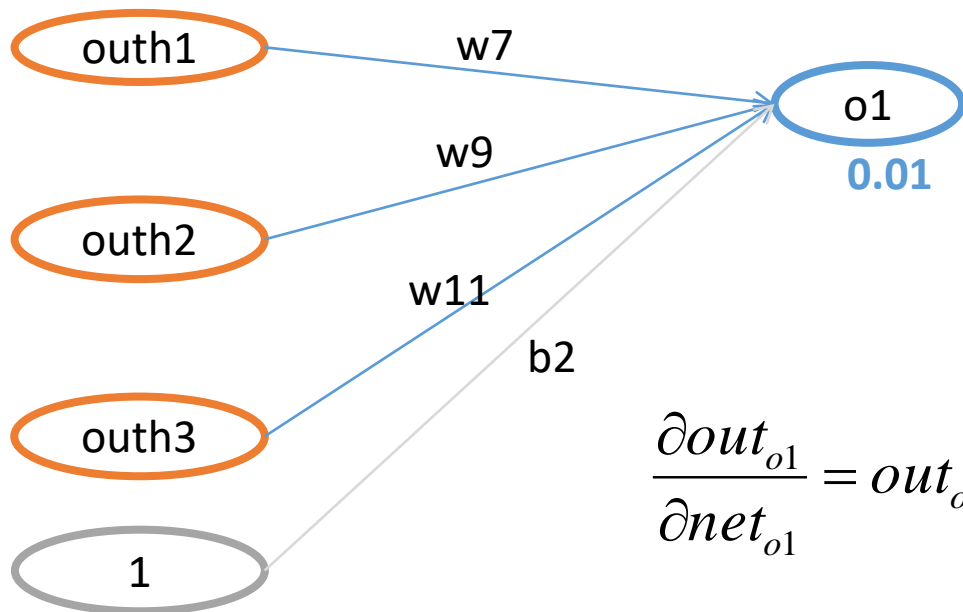
$$E_{o1} = \frac{1}{2} (\text{target}_{o1} - \text{out}_{o1})^2$$

$$E_{total} = E_{o1} + E_{o2}$$

$$\frac{\partial E_{total}}{\partial w_7} = \frac{\partial E_{total}}{\partial \text{out}_{o1}} * \frac{\partial \text{out}_{o1}}{\partial \text{net}_{o1}} * \frac{\partial \text{net}_{o1}}{\partial w_7}$$

$$\frac{\partial E_{total}}{\partial \text{out}_{o1}} = 2 * \frac{1}{2} (\text{target}_{o1} - \text{out}_{o1})^{2-1} * -1 + 0 = -(0.01 - 0.891090) = 0.88109$$

## BP算法例子-BP过程(W7)

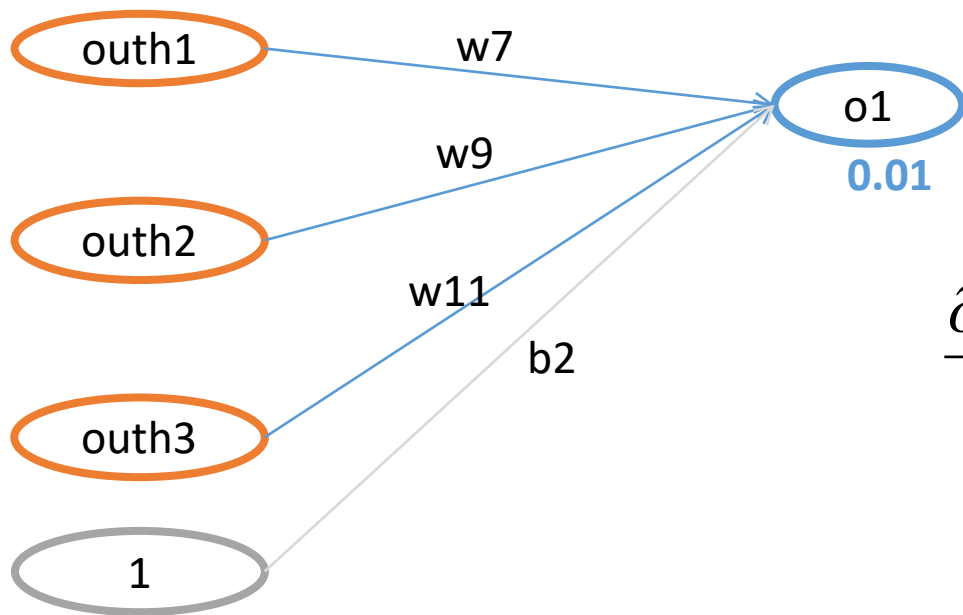


$$out_{o1} = \frac{1}{1 + e^{-net_{o1}}}$$

$$\frac{\partial out_{o1}}{\partial net_{o1}} = out_{o1}(1 - out_{o1}) = 0.891090(1 - 0.891090) = 0.097049$$

$$out'_{o1} = \frac{e^{-net}}{(1 + e^{-net})^2} = \frac{1 + e^{-net} - 1}{(1 + e^{-net})^2} = \frac{1}{1 + e^{-net}} - \frac{1}{(1 + e^{-net})^2} = out_{o1}(1 - out_{o1})$$

## BP算法例子-BP过程 (W7)

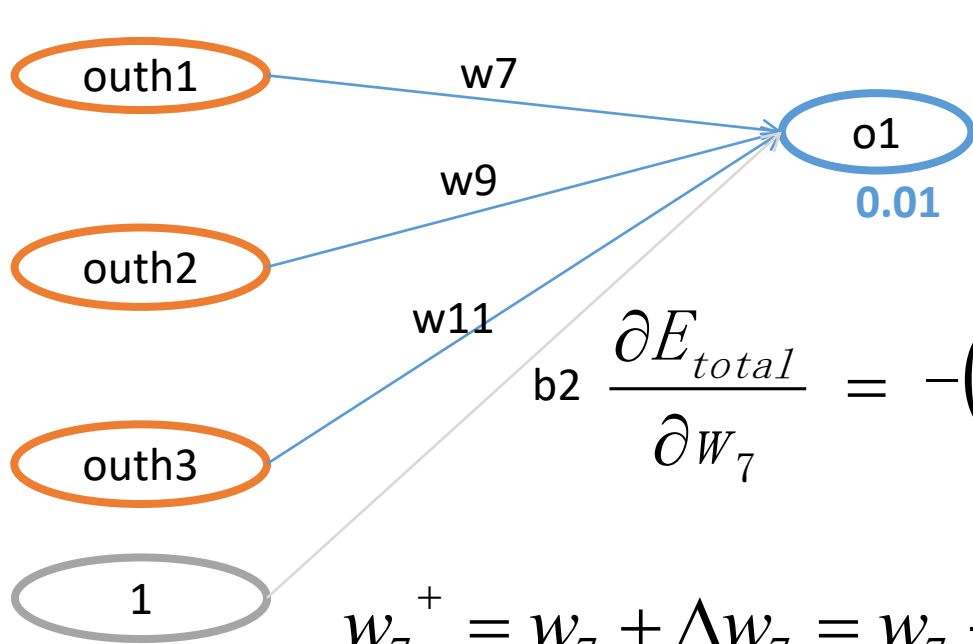


$$\frac{\partial net_{o1}}{\partial w_7} = 1 * out_{h1} * w_7^{(1-1)} + 0 + 0 + 0 = 0.912934$$

$$net_{o1} = w_7 * out_{h1} + w_9 * out_{h2} + w_{11} * out_{h3} + b_2 * 1$$

$$\frac{\partial E_{total}}{\partial w_7} = 0.88109 * 0.097049 * 0.912934 = 0.078064$$

## BP算法例子-BP过程(W7)



$$E_{o1} = \frac{1}{2} (\text{target}_{o1} - \text{out}_{o1})^2$$

$$E_{total} = E_{o1} + E_{o2}$$

$$\frac{\partial E_{total}}{\partial w_7} = \frac{\partial E_{total}}{\partial \text{out}_{o1}} * \frac{\partial \text{out}_{o1}}{\partial \text{net}_{o1}} * \frac{\partial \text{net}_{o1}}{\partial w_7}$$

$$\frac{\partial E_{total}}{\partial w_7} = -(\text{target} - \text{out}_{o1}) * \text{out}_{o1} * (1 - \text{out}_{o1}) * \text{out}_{h1}$$

$$w_7^+ = w_7 + \Delta w_7 = w_7 - \eta \frac{\partial E_{total}}{\partial w_7} = 0.4 - 0.5 * 0.078064 = 0.360968$$

$$w_8^+ = 0.453383$$

$$w_9^+ = 0.458137$$

$$w_{10}^+ = 0.553629$$

$$w_{11}^+ = 0.557448$$

$$w_{12}^+ = 0.653688$$

## BP算法例子-BP过程(W1)

$$\frac{\partial E_{total}}{\partial w_1} = \frac{\partial E_{total}}{\partial out_{h1}} * \frac{\partial out_{h1}}{\partial net_{h1}} * \frac{\partial net_{h1}}{\partial w_1} = \left( \frac{\partial E_{o1}}{\partial out_{h1}} + \frac{\partial E_{o2}}{\partial out_{h1}} \right) * \frac{\partial out_{h1}}{\partial net_{h1}} * \frac{\partial net_{h1}}{\partial w_1}$$

$$\frac{\partial E_{o1}}{\partial out_{h1}} = \frac{\partial E_{o1}}{\partial out_{o1}} * \frac{\partial out_{o1}}{\partial net_{o1}} * \frac{\partial net_{o1}}{\partial out_{h1}} = -(\text{target}_{o1} - out_{o1}) * out_{o1} * (1 - out_{o1}) * w_7$$

$$\frac{\partial E_{o1}}{\partial out_{h1}} = -(0.01 - 0.891090) * 0.891090 * (1 - 0.891090) * 0.360968 = 0.030866$$

## BP算法例子-BP过程(W1)

$$\frac{\partial E_{o2}}{\partial out_{h1}} = \frac{\partial E_{o2}}{\partial out_{o2}} * \frac{\partial out_{o2}}{\partial net_{o2}} * \frac{\partial net_{o2}}{\partial out_{h1}} = -(\text{target}_{o2} - out_{o2}) * out_{o2} * (1 - out_{o2}) * w8$$

$$\frac{\partial E_{total}}{\partial w_1} = 0.011204$$

$$w_1^+ = w_1 + \Delta w_1 = w_1 - \eta \frac{\partial E_{total}}{\partial w_1} = 0.1 - 0.5 * 0.011204 = 0.094534$$

## BP算法例子-BP过程

$$w_1^+ = 0.094534$$

$$w_2^+ = 0.139069$$

$$w_3^+ = 0.198211$$

$$w_4^+ = 0.246422$$

$$w_5^+ = 0.299497$$

$$w_6^+ = 0.348993$$

$$w_7^+ = 0.360968$$

$$w_8^+ = 0.453383$$

$$w_9^+ = 0.458137$$

$$w_{10}^+ = 0.553629$$

$$w_{11}^+ = 0.557448$$

$$w_{12}^+ = 0.653688$$

$$b_1 = 0.35$$

$$b_2 = 0.65$$

## BP算法例子-FP多次迭代效果

- 第10次迭代结果:  $O = (0.662866, 0.908195)$
- 第100次迭代结果:  $O = (0.073889, 0.945864)$
- 第1000次迭代结果:  $O = (0.022971, 0.977675)$

$$w^0 = \begin{pmatrix} 0.1, 0.15, 0.2, 0.25, \\ 0.3, 0.35, 0.4, 0.45, \\ 0.5, 0.55, 0.6, 0.65 \end{pmatrix} \quad w^{1000} = \begin{pmatrix} 0.214925, 0.379850, 0.262855, \\ 0.375711, 0.323201, 0.396402, \\ -1.48972, 0.941715, -1.50182, \\ 1.049019, -1.42756, 1.151881 \end{pmatrix}$$



## 神经网络之DNN问题

- 一般来讲，可以通过增加神经元和网络层次来提升神经网络的学习能力，使其得到的模型更加能够符合数据的分布场景；但是实际应用场景中，神经网络的层次一般情况不会太大，因为太深的层次有可能产生一些求解的问题
- 在DNN的求解中有可能存在两个问题：**梯度消失**和**梯度爆炸**；我们在求解梯度的时候会使用到链式求导法则，实际上就是一系列的连乘，如果每一层都小于1的话，则梯度越往前乘越小，导致梯度消失，而如果连乘的数字在每层都是大于1的，则梯度越往前乘越大，导致梯度爆炸。



谢谢