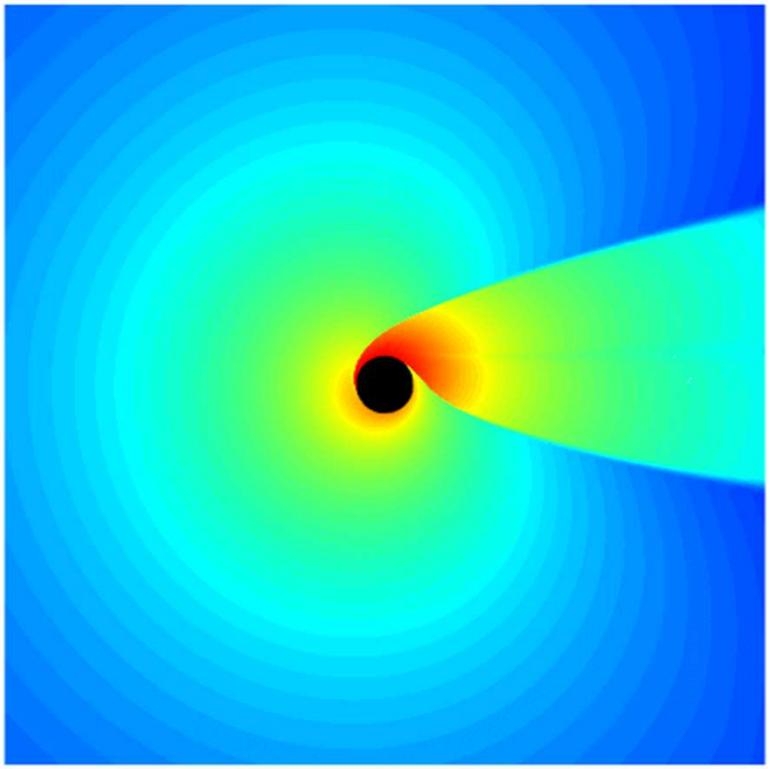
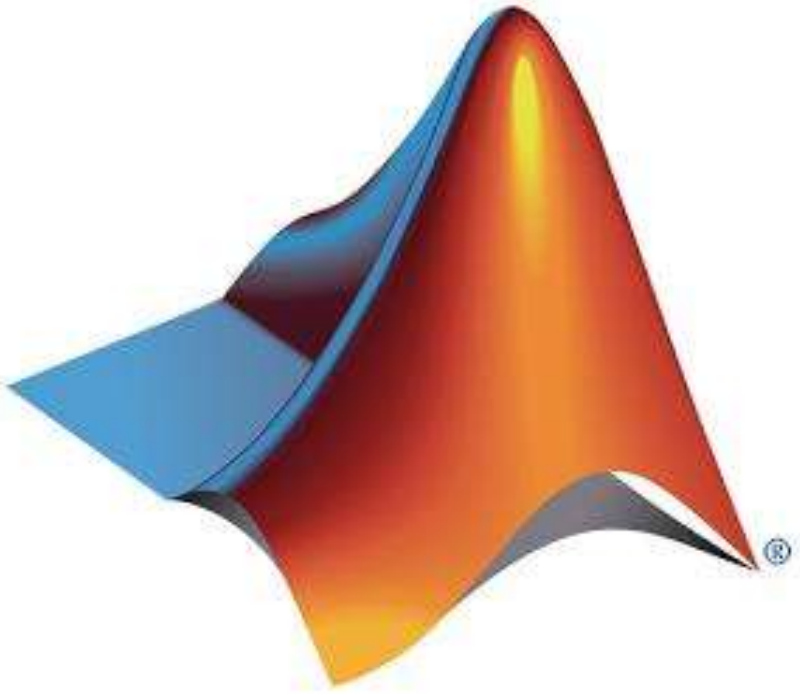
数学实验
Mathematical Experiments





实验十三：
智能优化与机器学习
Intelligent optimization and machine learning

实验1：智能优化算法

◆实验的背景

经典优化算法的缺陷

经典数学优化算法，如Newton法、最速下降法等，以追求算法的速率为目标，对函数的性质和初始解的要求较高，容易落入局部极值点的陷阱. 近几十年来，人工智能算法的兴起，为解决数学优化问题提供了一条新的途径，如

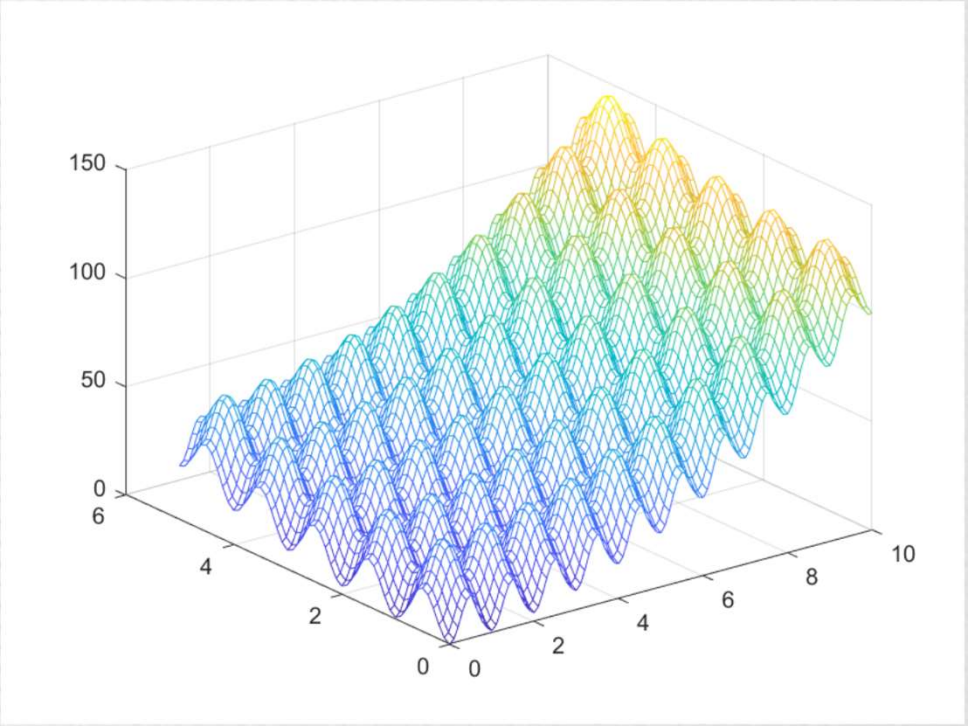
遗传算法、蚁群算法、模拟退火算法、粒子群算法等.

这些智能算法不依赖于精确的数学模型，能同时处理连续型和离散型变量，具有较好跳出局部极值点的能力.

◆实验的背景

例1 求解下列二元函数在 $0 \le x_1 \le 10, 0 \le x_2 \le 5$ 范围内的最大值:

$$\max f(x) = 20 + {x_1}^2 + {x_2}^2 - 10[\cos(2\pi x_1) + 10[\cos(2\pi x_2)].$$



作图可见该函数有很多极大值点.

```
x1=0:0.1:10;x2=0:0.1:5;
[x,y]=ndgrid(x1,x2);
z=20+x.^2+y.^2-10*(cos(2*pi*x)+cos(2*pi*y));
mesh(x,y,z)
```

◆实验的背景

解 尝试用fminsearch求解，先转化为最小值问题：

```
>>fun=@(x)-20-sum(x.^2)+10*sum(cos(2*pi*x));
>>[x,f]=fminsearch(fun,[0,0]) %原点作初值
x=0.5025  0.5025
f=-40.5025
>>[x,f]=fminsearch(fun,[5,2])%中点作初值
x=5.5281  2.5128
f=-76.6863
```

可见求解依赖于初值.事实上，以上都只是局部最优解，不是全局最优解.

◆实验的背景

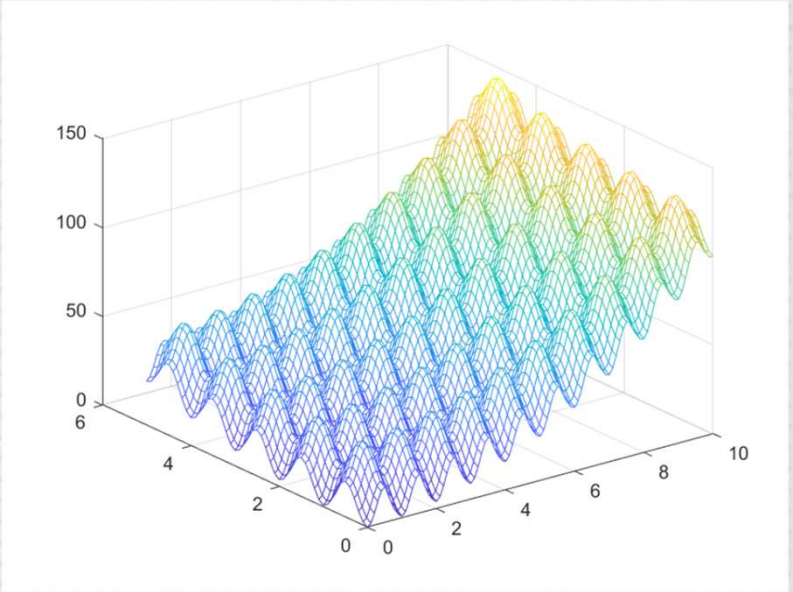
观察图可知，最优解应该靠近(10,5)，但是

```
>>[x,f]=fminsearch(fun,[9.9,4.9])
```

却出现了计算溢出.

若使用(9.5,4.5)作为初值，则

```
>>[x,f]=fminsearch(fun,[9.5,4.5])
x=9.5491  4.5230
f=-151.0665
```



终于可以找到最优解.问题是：怎样找到一个好的初值？怎样跳出局部极值点？

遗传算法的基本思想

◆遗传算法

1. 遗传算法的原理

遗传算法(genetic algorithm, 简称GA)是一种通过模拟自然进化过程搜索最优解的方法. 该领域最早是由美国Michigan大学的John Holland于1975年提出的. 遗传算法在解决复杂优化问题方面不断取得成功，受到人们关注. 另一方面，遗传算法本身并不要求对优化问题的性质作深入的数学分析，因而对不太熟悉数学理论和算法的使用者来说，无疑是方便的.

◆遗传算法

遗传算法的过程：

- 首先，随机产生一定数目的初始染色体（它们组成一个种群），种群中染色体的数目称为种群的大小或规模；
- 用评价函数来评价每一个染色体的优劣，即染色体对环境的适应程度（适应度），用来作为以后遗传操作的依据.
- 然后，进行选择过程：选择的目的是为了从当前种群中选出优良的染色体，判断染色体优良与否的准则是各自的适应度，即：
  - 染色体的适应度越高，其被选择的机会就越多；
- 通过选择产生的新的种群进行交叉操作.
- 进行变异操作：变异操作的目的是为了挖掘种群中个体的多样性，克服有可能陷入局部解的弊病；
- 最后，对新的种群（后代）重复进行选择、交叉、变异操作，经过给定次数的迭代处理以后，把最好的染色体作为优化问题的最优解.

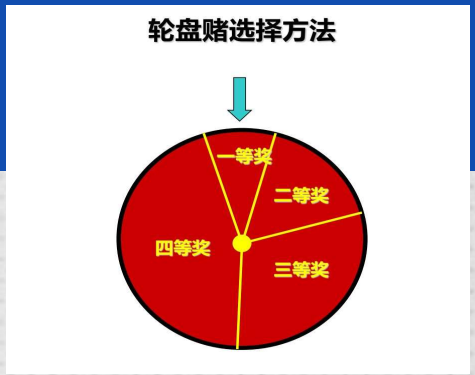
◆遗传算法

2. 遗传算法的步骤

(1) 初始化过程

定义整数n为染色体的个数，并随机产生n个初始染色体，一般情况下，由于优化问题的复杂性，解析地产生初始染色体有困难，可用下述方法产生初始染色体：在优化问题的可行集中选择一个内点V_0，给定一个足够大的数M，以保证遗传操作能遍及整个可行集.

◆遗传算法

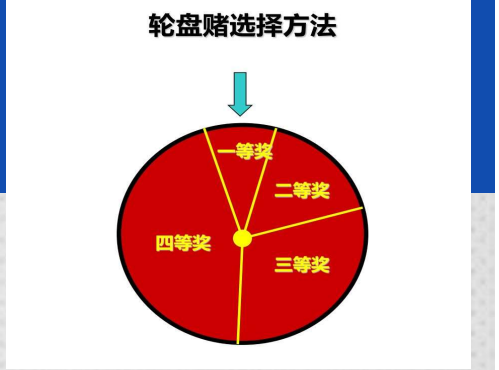


(2) 评价函数(适应度)

评价函数(用F(V)表示)用来对种群中每个染色体V设定一个概率，以使该染色体被选择的可能性与其种群中其他染色体的适应性成比例，评价函数有多种，比较常用的有基于序的评价函数和适应度定标设计的评价函数等.

以基于序的评价函数为例，通过轮盘赌，适应性强的染色体被选择产生后代的机会要大. 方法如下：设目前该代中的染色体为$V_1, V_2, \cdots, V_n$，可根据染色体的序进行再生分配，即**决策者可以作一合理假设，即在$V_1, V_2, \cdots, V_n$中给出一个序的关系**，使染色体由好到坏重排，即一个染色体的序号越小越好. 如给定参数$\alpha \epsilon (0,1)$，可定义基于序的评价函数为$F(V_i) = \alpha(1-\alpha)^{i-1}, i = 1,2,\cdots,n.$

◆遗传算法



(3) 选择过程

选择染色体的方式：以基于轮盘赌的选择过程为例，以旋转赌轮n次为基础，赌轮上的刻度是按每个染色体的适应度来划分的. 因此，刻度并不是平均分配在赌轮上的，染色体的适应度越大，该染色体在赌轮上所占的面积就越大，该染色体被选中的概率也就越大. 按每个染色体的适应度选择，每次旋转都为新的种群选择一个染色体.

◆遗传算法

无论使用何种评价函数，选择过程都可以写成如下形式：

对每个染色体V_i，其累计概率为q_i.

步骤1

$$\begin{cases} q_0 = 0, \\ q_i = \sum_{j=1}^{i} F(V_j), i = 1,2,\cdots,n; \end{cases}$$

步骤2　从区间$(0, q_n)$中产生一个随机数r；

步骤3　若$q_{i-1} < r \leq q_i$，则选择第i个染色体V_i；

步骤4　重复步骤2，步骤3共n次，这样可得到n个复制的染色体.

◆遗传算法

(4) 交叉操作

定义参数P_c作为交叉操作的概率，这个概率说明每次种群中平均有$P_c n$个染色体进行交叉操作.

为确定交叉父代，从i=1到n重复进行以下过程：从[0, 1]中产生一个随机数r, 若$r < P_c$，则选择V_i作为一个父代，否则V_i不被选择. 用V_1', V_2', V_3'表示上面选择的父代，并随机配对$(V_1', V_2'), (V_3', V_4'), (V_5', V_6'), \cdots$.

◆遗传算法

以(V_1', V_2')为例介绍交叉操作. 从$[0, 1]$产生一个随机数c，产生后代X和Y：

$$X = cV'_1 + (1-c)V'_2, Y = (1-c)V'_1 + cV'_2.$$

若X和Y均可行，则用它们代替其父代. 否则，重新选择c，重复上面的后代产生过程，直到产生可行的后代或重复到指定的次数为止. 总之，仅用可行的后代替换其父代.

◆遗传算法

(5) 变异操作

定义参数P_m作为遗传系统中的变异概率，这个概率说明每次种群中平均有$P_m n$个染色体进行变异操作.

为确定交叉父代，从i=1到n重复以下过程：从[0, 1]中产生一个随机数r，若$r < P_m$，则V_i作为一个变异父代，否则V_i不被选择. 对被选择变异的父代进行变异操作，用$V = V(x_1, x_2, \cdots, x_n)$表示上面选中的父代，对其按下列方法进行变异.

◆遗传算法

在R^n中随机选择变异方向d，检验$V+M\cdot d$是否可行. 如果不可行，就置M为0与M之间的随机数，重新检验，直到$V+M\cdot d$可行为止，其中M为初始化过程定义的一个足够大的数. 如果在预先给的迭代次数之内没有找到可行解，就置M为0，无论M为何值，总用$V+M\cdot d$代替父代染色体V.

遗传算法举例：
(以整数规划、二进制编码为例)

◆遗传算法

例 考虑二元函数

$f(x_1, x_2) = x_1 + {x_2}^2, x_1, x_2 \in \{0,1,\cdots,7\}.$

解 显然这个问题的最大值点在$x_1 = 7, x_2 = 7$. 我们用它来解释遗传算法解的寻优过程. 遗传算法的运算对象是表示个体的符号串，把变量x_1，x_2的所有可行解编码为一种二进制符号串，称为基因型. 例如，基因型011101对应x1=3，x2=5；而基因型111111对应$x_1 = 7, x_2 = 7$.

◆遗传算法

(1) 初始群体：取群体规模的大小为4，随机产生p(0)，如011101, 101011，011100, 111001.

(2) 适应度：利用目标函数值作为个体的适应度.

(3) 选择：采用轮盘赌选择方式，按与适应度成正比的概率选择个体复制的数量.

个体编号	初始群体 $p(0)$	x_1	x_2	适应度	占比	选择次数	选择结果
1	011101	3	5	28	0.4	2	011101
2	101011	5	3	14	0.2	1	011100
3	011100	3	4	19	0.28	1	011101
4	111001	7	1	8	0.12	0	101011

◆遗传算法

(4) 交叉(大概率)：先对群体进行随机配对，其次随机设置交叉点位置，再相互交换配对染色体之间的部分基因.

个体编号	选择结果	配对情况	交叉点位置	交叉结果
1	01 | 1101	1-2	1-2 : 2	01 1100
2	01 | 1100			01 1101
3	0111 | 01	3-4	3-4 : 4	0111 11
4	1010 | 11			1010 01

◆遗传算法

(5)变异(小概率)：首先确定出各个个体的基因变异位置，然后依照某一概率将变异点的原有基因值取反.

个体编号	结果	变异点	变异结果
1	01[1]100	3	01[0]100
2	011101	无	011101
3	011111	无	011111
4	101[0]01	4	101[1]01

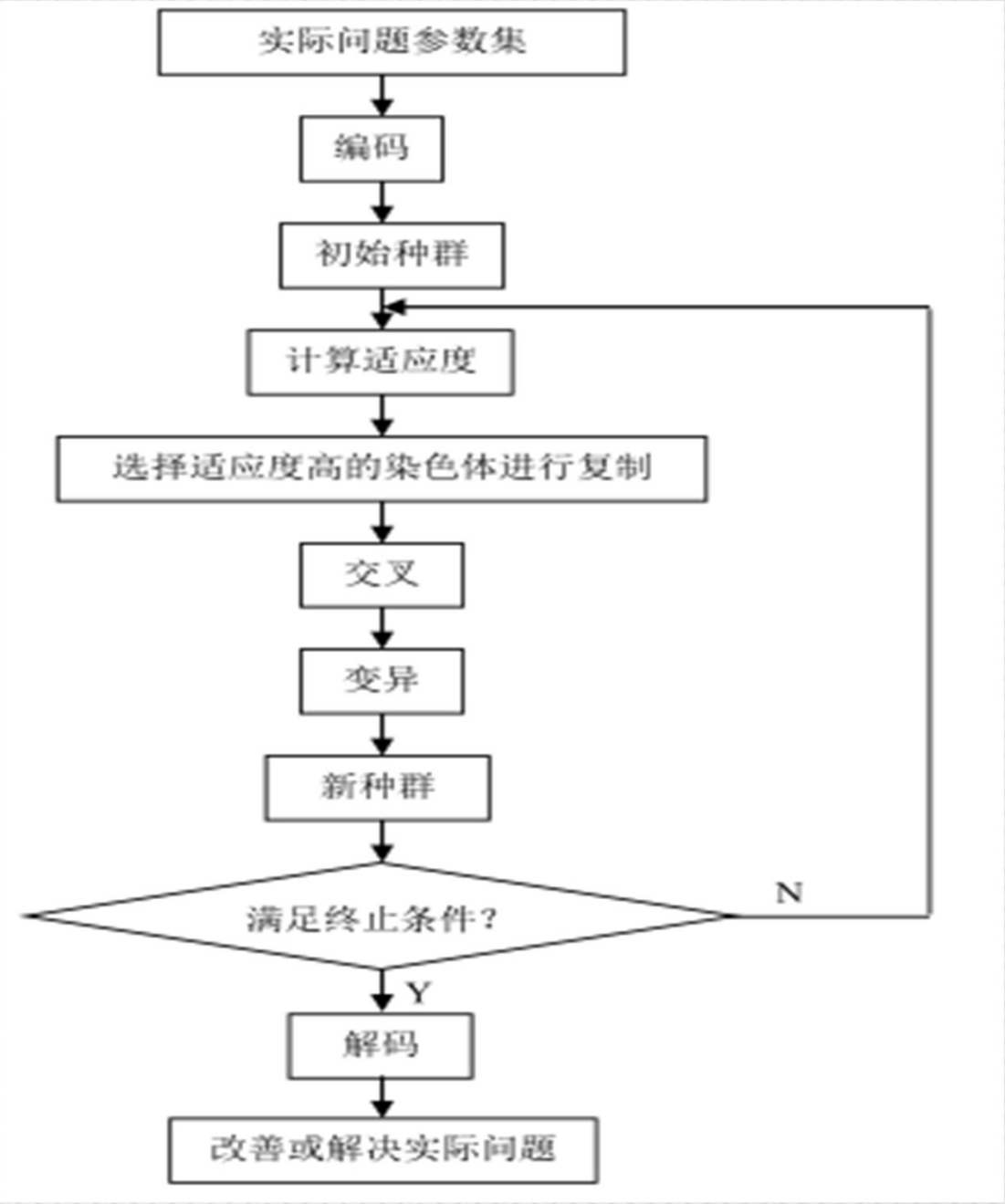
对群体p(0)进行一轮选择、交叉、变异运算之后可得到新一代的群体p(1).

◆遗传算法

个体编号	父代群体 $p(0)$	x_1 x_2	适应度	子代群体	x_1 x_2	适应度
1	011101	3 5	28	010100	2 4	18
2	101011	5 3	14	011101	3 5	28
3	011100	3 4	19	011111	3 7	52
4	111001	7 1	8	101101	5 5	30
均值			17.25			32

从上表中可以看出，群体经过一代进化之后，其适应度的最大值从28增加到52，适应度平均值从17.25增加到32，都得到了明显的改进. 继续迭代计算，对群体p(t)进行一轮选择、交叉、变异运算之后可得到新一代的群体p(t+1)，最终求得最优解$x_1 = 7, x_2 = 7$.

◆遗传算法


实际问题参数集
编码
初始种群
计算适应度
选择适应度高的染色体进行复制
交叉
变异
新种群
满足终止条件？
N
Y
解码
改善或解决实际问题

◆遗传算法

现在我们来用遗传算法求解例1的多极值点的二元函数最大值问题.

讲解程序Exp13_1.m

运行得到最优解

$x_1 = 9.5503, x_2 = 4.5303,$

目标函数最优值为151.0558. 注意：由于随机性，每次运行的结果可能不一致.

MATLAB的遗传算法函数

◆MATLAB的遗传算法函数

MATLAB全局优化工具箱提供了用遗传算法求约束优化问题的函数ga.

```
x=ga(fitnessfcn,nvars)
x=ga(fitnessfcn,nvars,options)
x=ga(fitnessfcn,nvars,A,b)
x=ga(fitnessfcn,nvars,A,b,Aeq,beq)
x=ga(fitnessfcn,nvars,A,b,Aeq,beq,lb,ub)
x=ga(fitnessfcn,nvars,A,b,Aeq,beq,lb,ub,nonlcon)
x=ga(fitnessfcn,nvars,A,b,Aeq,beq,lb,ub,nonlcon,options)
x=ga(fitnessfcn,nvars,A,b,[],[],LB,UB,nonlcon,IntCon)
x=ga(fitnessfcn,nvars,A,b,[],[],LB,UB,nonlcon,IntCon,options)
```

◆遗传算法函数

x=ga(fitnessfcn,nvars,A,b,[],[],LB,UB,nonlcon,IntCon,options)

其中nvars为解向量长度.IntCon为正整数向量，IntCon(i)=k表示x(k)是整数变量，显然i<=nvars.注意：当存在整数变量时，ga不接受等式约束。

fitnessfcn可用匿名函数方式直接调用，如@(x)sin(x);

也可写成M函数形式(fitnessfcn.m)：function f=fitnessfcn(x)f=f(x);

end

◆遗传算法函数

非线性约束条件写成如下的M函数形式

(nonlcon.m):

```
function[c,ceq]=nonlcon(x)
c=c(x);ceq=ceg(x);
end
```

◆遗传算法函数

求解优化问题(可以是混合整数规划)

min f(x)

s.t.A·x≤b,Aeq·x=beq,

c(x)≤0,ceq(x)=0,lb≤x≤ub,

当约束条件中缺A和b,Aeq和beq,lb或ub、nonlcon时，相关项可用[]代替以分别表示没有不等式约束、等式约束、下界或上界、非线性约束

算法参数选择可由函数optimoptions完成

[x,f]=ga(…) 同时返回解x处的函数值f

可调用help ga或doc ga来了解ga更多的用法

◆遗传算法函数

我们来用ga求解例1的多极值点的二元函数最大值问题. 它不需要特别考虑初值, 容易找到全局最优解. 注意由于ga求解最小值, 目标函数需要加负号.

```
>>fun=@(x)-20-sum(x.^2)+10*sum(cos(2*pi*x));
>>[x, f]=ga(fun, 2, [], [], [], [], [0  0], [10  5])
```

X=9.5491 4.5230

f=-151.0665

◆遗传算法函数

例3 用遗传算法计算约束非线性规划

$$\min f(x_1,x_2)=e^{x_1}(4{x_1}^2+2{x_2}^2+4x_1x_2+2x_2+1)$$

$$s.t.\begin{cases}1.5+x_1x_2-x_1-x_2\le 0,\\ 10+x_1x_2\ge 0.\end{cases}$$

解 编写约束函数eg14_4con. m, 注意需化为标准形式.

◆遗传算法函数

```
%M函数eg14_4con.m
function[c,ceg]=eg14_4con(x)
c=[1.5+x(1)*x(2)-x(1)-x(2);-10-x(1)*x(2)];
%非线性不等式约束
ceq=[];
end
```

计算如下：

```
fun=@(x)exp(x(1))*(4*x(1)^2+2*x(2)^2+4*x(1)*x(2)+2*x(2)+1);
>>[x,f]=ga(fun,2,[],[],[],[],[],[],@ eg14_4con)
x=-8.0257  1.2245
f=0.0735
```

◆遗传算法函数

由于遗传算法的随机性，每次运行结果可能不同，可多运行几次选取最好的结果.默认的算法参数可以通过下列命令观察：

```
>>optimoptions('ga')
```

可以通过增加初始种群数量提高计算精度：

```
>>options=optimoptions('ga','PopulationSize',200);
>>[x,f]=ga(fun,2,[],[],[],[],[],[],eg14_4con,[],options)
x=-9.3293 1.0684
f=0.0279
```

◆遗传算法函数

例4 用遗传算法求解混合整数规划

$$\max xyz$$
$$s.t.\begin{cases} -x+xy+2z\geq 0, \\ x+2y+2z\leq 72, \\ 10\leq y\leq 20 \\ x,y\text{为整数}. \end{cases}$$

解 将目标函数max化为min，不等式约束“≥”化为“≤”，决策变量记为x(1),x(2),x(3).在编辑器窗口编写M文件.

◆遗传算法函数

```
%M函数eg14_5con.m
function[g,geq]=eg14_5con(x)
g=x(1)-x(1)*x(2)-2*x(3);
geg=[];
end
```

◆遗传算法函数

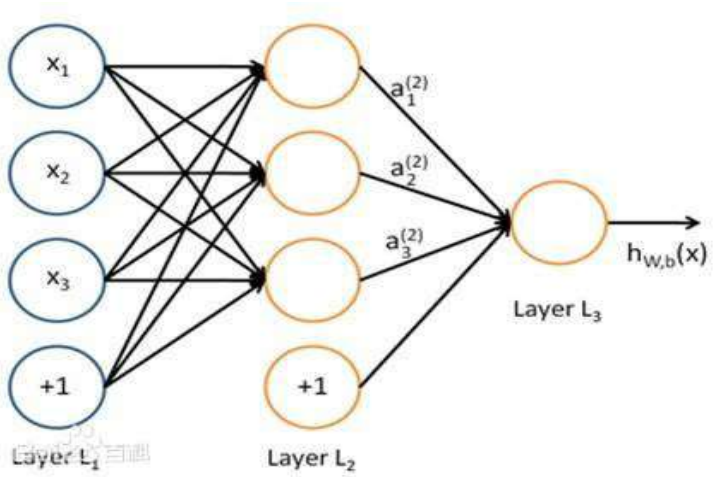
再在命令行窗口执行

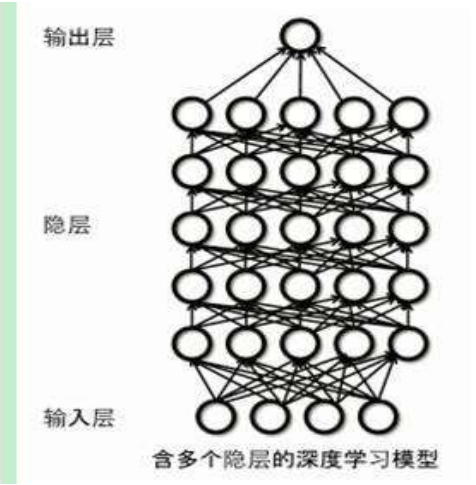
```
>>A=[1 2 2];b=72;fun=@(x)-x(1)*x(2)*x(3);
>>[x,fval]=ga(fun,3,A,b,[],[],[-inf,10,-
inf]',...[inf,20,inf]',@eg14_5con,[1 2])
```

求得最大值点x=24，y=12，z=12.0005，最大值3456.144.由于遗传算法的随机性，每次运行结果可能不同.

实验2：深度学习

深度学习



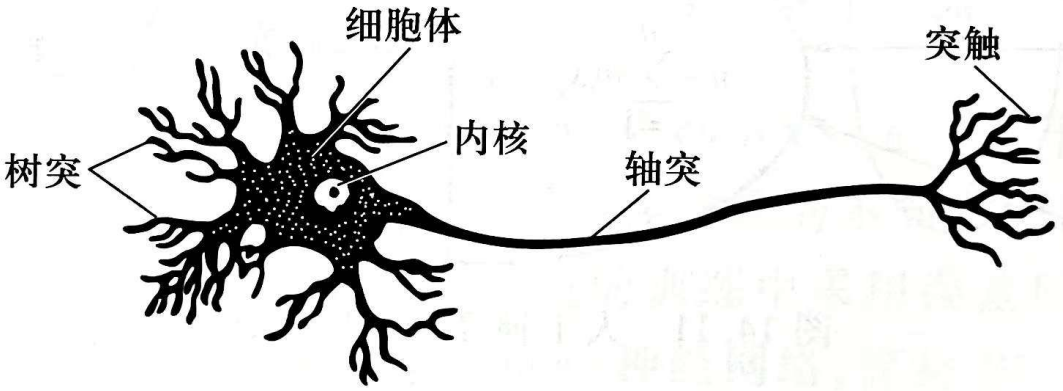


深度学习(DL, Deep Learning)是机器学习(ML, Machine Learning)领域中一个新的研究方向。

深度学习的概念源于人工神经网络（ANN,Artificial Neural Network）的研究，含多个隐藏层的多层感知器就是一种深度学习结构。深度学习通过组合低层特征形成更加抽象的高层表示属性类别或特征，以发现数据的分布式特征表示。研究深度学习的动机在于建立模拟人脑进行分析学习的神经网络，它模仿人脑的机制来解释数据，例如图像，声音和文本等。

生物神经网络

生物神经网络

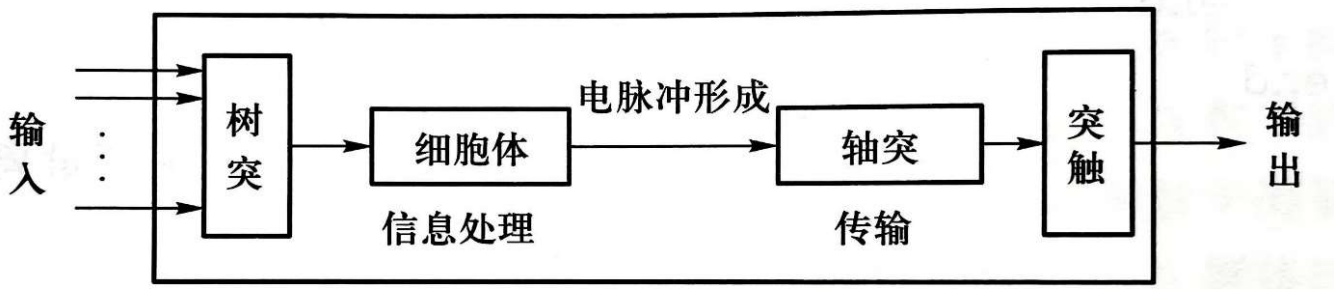


生物神经元结构简图

(生物神经网络)大脑的一个重要成分是神经网络.神经网络由相互关联的神经元组成.

每个神经元由细胞体(包含细胞内核)、树突、轴突(包含突触)组成.大多数神经元具有多个树突，每个树突都较短，分支较多，可扩大接受信息面积.每个神经元只有一个轴突，其末端分支多，终端末梢形成许多球形的突触小体，贴附于另一个神经元的树突，形成突触.树突的机能是接受其他神经元传来的神经冲动，并将冲动传到细胞体.轴突的机能主要是传导神经冲动，能将冲动经过突触传递到另一个神经元上.

生物神经网络

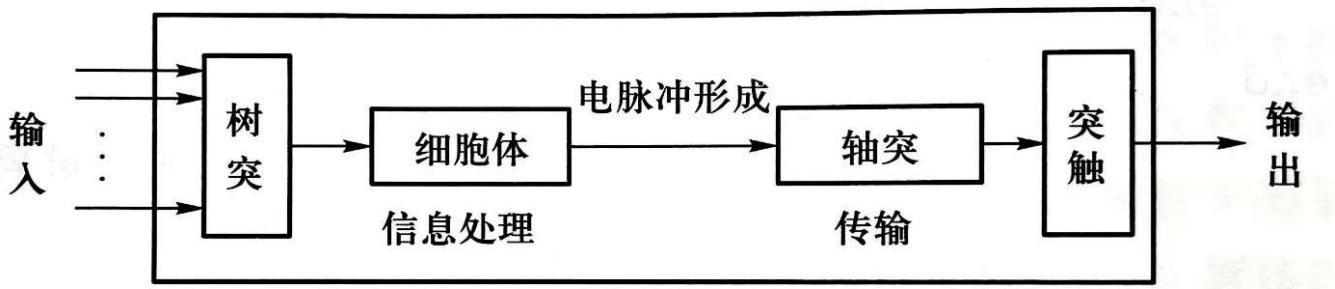


生物神经元功能模型

(生物神经网络)大脑的一个重要成分是神经网络.神经网络由相互关联的神经元组成.

一个神经元通过树突接收到一定的信息后，对这些信息进行处理，再通过突触传送给其他神经元.神经元可分为“兴奋”性或“抑制”性两种.当一个神经元接收的兴奋信息累计超过某一值时，这个神经元被激活并传递出信息给其他神经元，这个固定值称为阈值.这种传递信息的神经元为“兴奋”性的.而一个神经元虽然接收到信息，但没有向外传递信息，便是“抑制”性的神经元.上图是生物神经元的基本功能模型.

生物神经网络



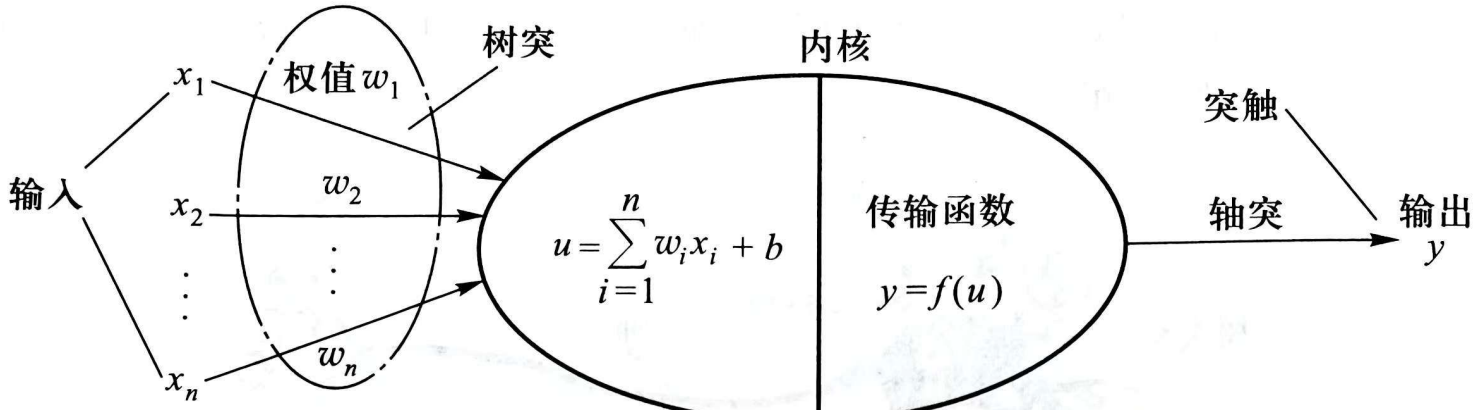
生物神经元功能模型

(生物神经网络)大脑的一个重要成分是神经网络.神经网络由相互关联的神经元组成.

单个神经元在突触地方彼此互相联系，从而形成一个巨大而又复杂的大脑信息处理网络.这些神经网络高度可变，因为神经元之间的突触可塑性较强，这种可塑性产生的变化也是学习的基础.

人工神经网络(artificial neural network,简称ANN)

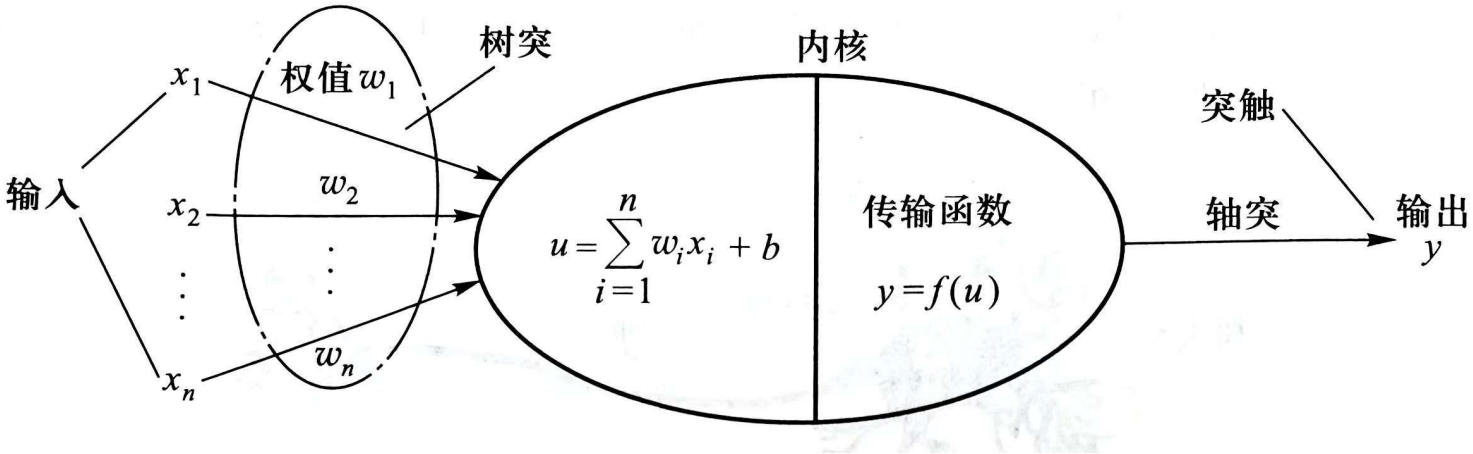
人工神经网络



人工神经元MP模型

人工神经元与神经网络1943年，美国心理学家W.McCulloch和数学家W.Pitts提出了一个简单的神经元模型，即MP模型.

人工神经网络



人工神经元MP模型

w_i为关联权值，表示神经元对第i个树突接收到信息的感知能力，其物理意义是输入信号的强度，若涉及多个神经元，则可以理解为神经元之间的连接强度. 神经元的权值w_i应该通过神经元对样本点反复的学习过程而确定，这样的学习过程在神经网络理论中称为训练. 传输函数也称为激活函数/激励函数（Activation Function），可以理解成对信号u的非线性映射，一般的激活函数应为单值函数，使得神经元是可逆的.

人工神经网络

McCulloch-Pitts 激活函数为

$$y = f(u) = sgn(\sum_{i=1}^{n} w_i x_i + b)$$

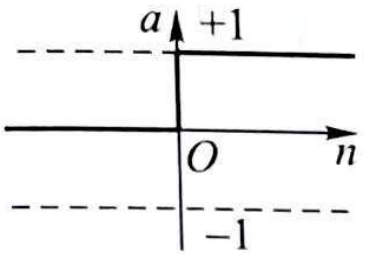
其中$sgn(x) = \begin{cases} 1, x > 0, \\ 0, 其他 \end{cases}$，b称其为偏置/阈值(bias).

除了上述符号函数,常用的传输函数是ReLU，Sigmoid函数(亦称为S型函数),例如，双曲正切Sigmoid 函数：

$f(u) = \frac{2}{1+e^{-2u}} - 1$，对数Sigmoid 函数：$f(u) = \frac{1}{1+e^{-u}}$,

对应的MATLAB命令分别是tansig和 logsig.

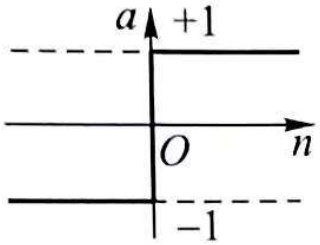
早期的几种激励函数



函数标记

$a=\text{hardlim}(n)$

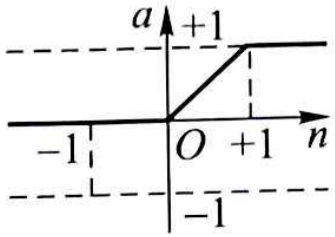
硬限幅传输函数



函数标记

$a=\text{hardlims}(n)$

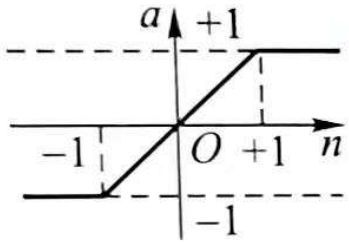
对称硬限幅传输函数



函数标记

$a=\text{satlin}(n)$

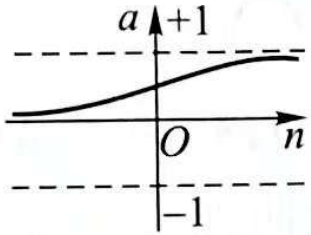
饱和线性传输函数



函数标记

$a=\text{satlins}(n)$

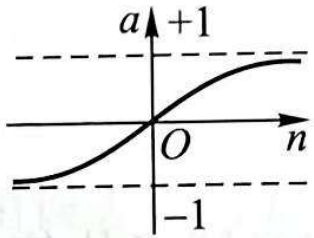
对称饱和线性传输函数



函数标记

$a=\text{logsig}(n)$

对数 Sigmoid 传输函数



函数标记

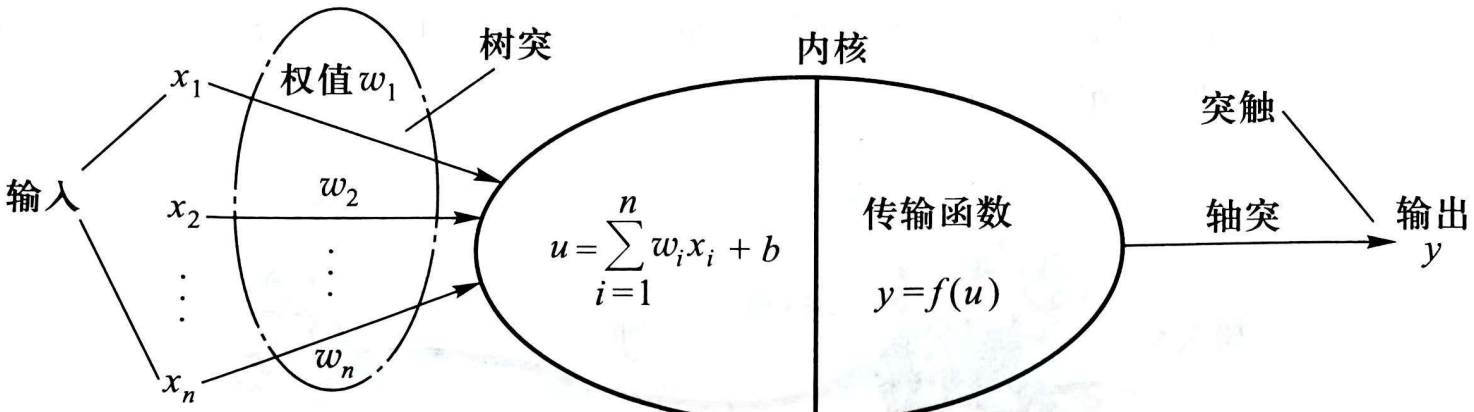
$a=\text{tansig}(n)$

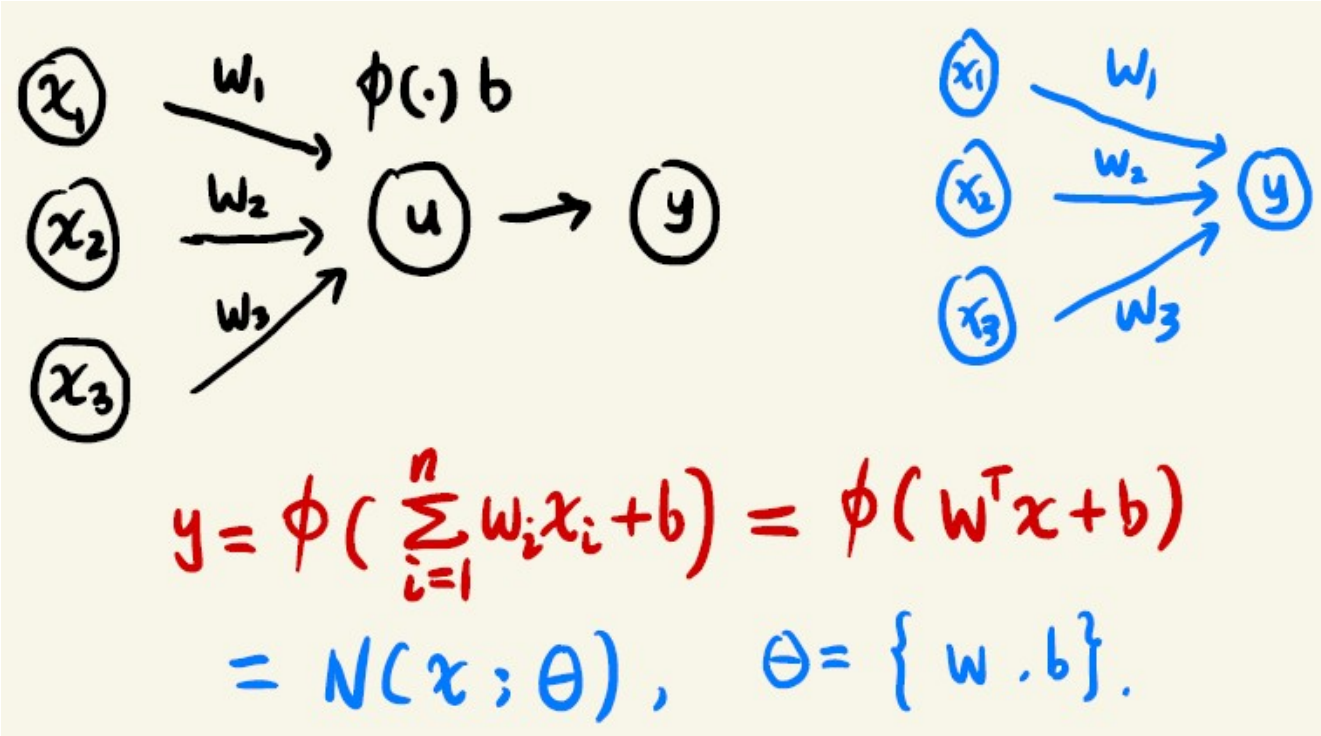
双曲正切 Sigmoid 传输函数

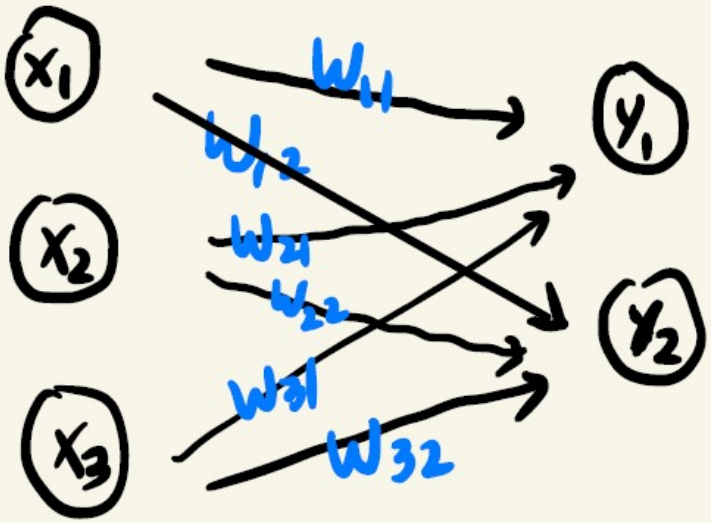
当前流行的几种激励函数

Name	Plot	Function, $f(x)$	Derivative of f, $f'(x)$	Range	Order of continuity
Identity		x	1	$(-\infty, \infty)$	C^∞
Binary step		$\begin{cases} 0 & \text{if } x < 0 \\ 1 & \text{if } x \geq 0 \end{cases}$	$\begin{cases} 0 & \text{if } x \neq 0 \\ \text{undefined} & \text{if } x = 0 \end{cases}$	$\{0, 1\}$	C^{-1}
Logistic, sigmoid, or soft step		$\sigma(x) = \frac{1}{1+e^{-x}}$	$f(x)(1-f(x))$	$(0, 1)$	C^∞
Hyperbolic tangent (tanh)		$\tanh(x) = \frac{e^x - e^{-x}}{e^x + e^{-x}}$	$1 - f(x)^2$	$(-1, 1)$	C^∞
Rectified linear unit (ReLU)[9]		$\begin{cases} 0 & \text{if } x \leq 0 \\ x & \text{if } x > 0 \end{cases}$ $= \max\{0, x\} = x\mathbf{1}_{x>0}$	$\begin{cases} 0 & \text{if } x < 0 \\ 1 & \text{if } x > 0 \\ \text{undefined} & \text{if } x = 0 \end{cases}$	$[0, \infty)$	C^0
Gaussian Error Linear Unit (GELU)[6]		$\frac{1}{2}x\left(1 + \operatorname{erf}\left(\frac{x}{\sqrt{2}}\right)\right)$ $= x\Phi(x)$	$\Phi(x) + x\phi(x)$	$(-0.17\ldots, \infty)$	C^∞
Softplus[10]		$\ln(1 + e^x)$	$\frac{1}{1+e^{-x}}$	$(0, \infty)$	C^∞
Exponential linear unit (ELU)[11]		$\begin{cases} \alpha(e^x - 1) & \text{if } x \leq 0 \\ x & \text{if } x > 0 \end{cases}$ with parameter α	$\begin{cases} \alpha e^x & \text{if } x < 0 \\ 1 & \text{if } x > 0 \\ 1 & \text{if } x = 0 \text{ and } \alpha = 1 \end{cases}$	$(-\alpha, \infty)$	$\begin{cases} C^1 & \text{if } \alpha = 1 \\ C^0 & \text{otherwise} \end{cases}$
Scaled exponential linear unit (SELU)[12]		$\lambda\begin{cases} \alpha(e^x - 1) & \text{if } x < 0 \\ x & \text{if } x \geq 0 \end{cases}$ with parameters $\lambda = 1.0507$ and $\alpha = 1.67326$	$\lambda\begin{cases} \alpha e^x & \text{if } x < 0 \\ 1 & \text{if } x \geq 0 \end{cases}$	$(-\lambda\alpha, \infty)$	C^0
Leaky rectified linear unit (Leaky ReLU)[13]		$\begin{cases} 0.01x & \text{if } x < 0 \\ x & \text{if } x \geq 0 \end{cases}$	$\begin{cases} 0.01 & \text{if } x < 0 \\ 1 & \text{if } x \geq 0 \end{cases}$	$(-\infty, \infty)$	C^0
Parametric rectified linear unit (PReLU)[14]		$\begin{cases} \alpha x & \text{if } x < 0 \\ x & \text{if } x \geq 0 \end{cases}$ with parameter α	$\begin{cases} \alpha & \text{if } x < 0 \\ 1 & \text{if } x \geq 0 \end{cases}$	$(-\infty, \infty)$	C^0
Sigmoid linear unit (SiLU,[6] Sigmoid shrinkage,[15] SiL,[16] or Swish-1[17])		$\frac{x}{1+e^{-x}}$	$\frac{1 + e^{-x} + xe^{-x}}{(1+e^{-x})^2}$	$[-0.278\ldots, \infty)$	C^∞
Mish [18]		$x\tanh(\ln(1+e^x))$	$\frac{(e^x(4e^{2x} + e^{3x} + 4(1+x) + e^x(6+4x)))}{(2+2e^x+e^{2x})^2}$	$[-0.308\ldots, \infty)$	C^∞
Gaussian		e^{-x^2}	$-2xe^{-x^2}$	$(0, 1]$	C^∞
Phish[19]		$x\tanh\left(\frac{1}{2}x\left(1 + \operatorname{erf}\left(\frac{x}{\sqrt{2}}\right)\right)\right)$		$(0, \infty)$	C^∞

深度BP神经网络


输入
x_1
x_2
x_n
权值 w_1
w_2
w_n
树突
内核
$u=\sum_{i=1}^{n} w_i x_i + b$
传输函数
$y=f(u)$
突触
轴突
输出
y


x_1
x_2
x_3
w_1
w_2
w_3
$\phi(\cdot)$ b
u
y
$y=\phi\left(\sum_{i=1}^{n} w_i x_i + b\right)=\phi(w^T x+b)$
$=N(x;\theta)$, $\theta=\{w, b\}$.

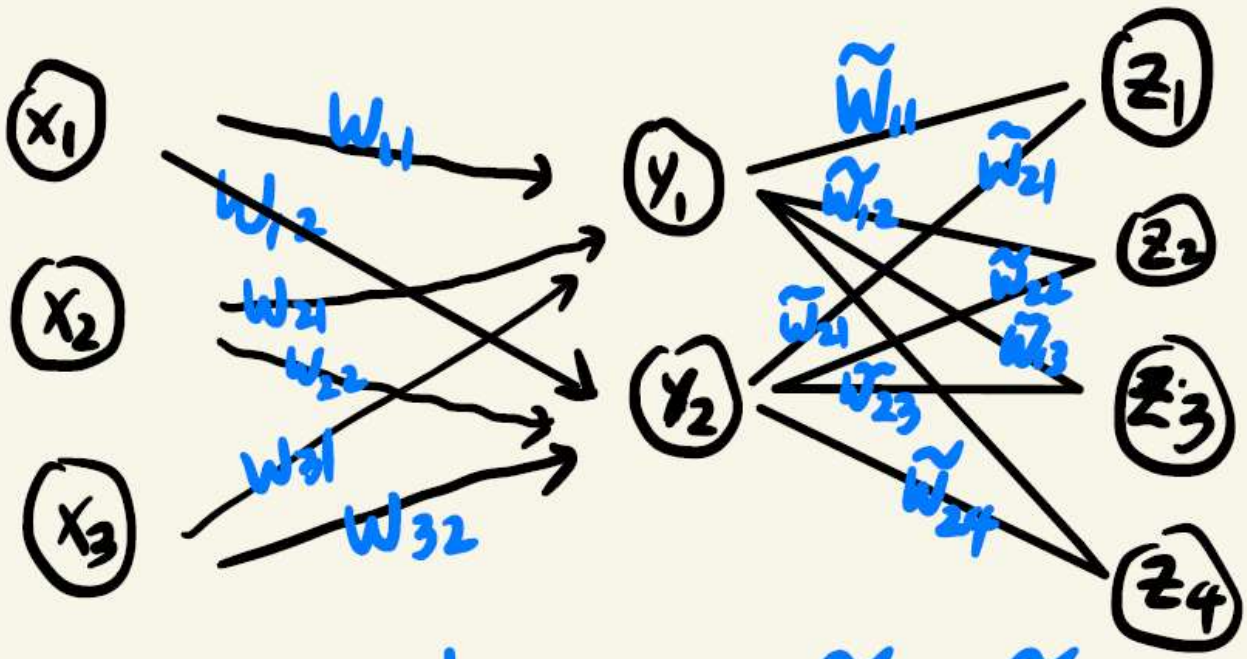


$$y_1 = \phi\left(\sum_{i=1}^{n} w_{i1} x_i + b_1\right)$$

$$y_2 = \phi\left(\sum_{i=1}^{n} w_{i2} x_i + b_2\right)$$

$$y = \phi(W^T x + b)$$

$$= N(x; \theta) \quad , \quad \theta = \{W, b\}$$

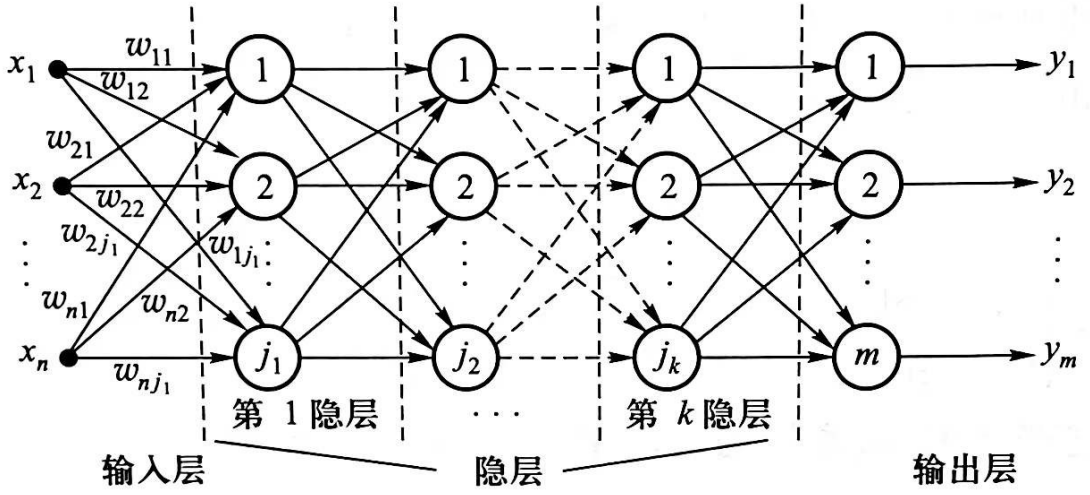


w, b $\quad \tilde{w} \quad \tilde{b}$

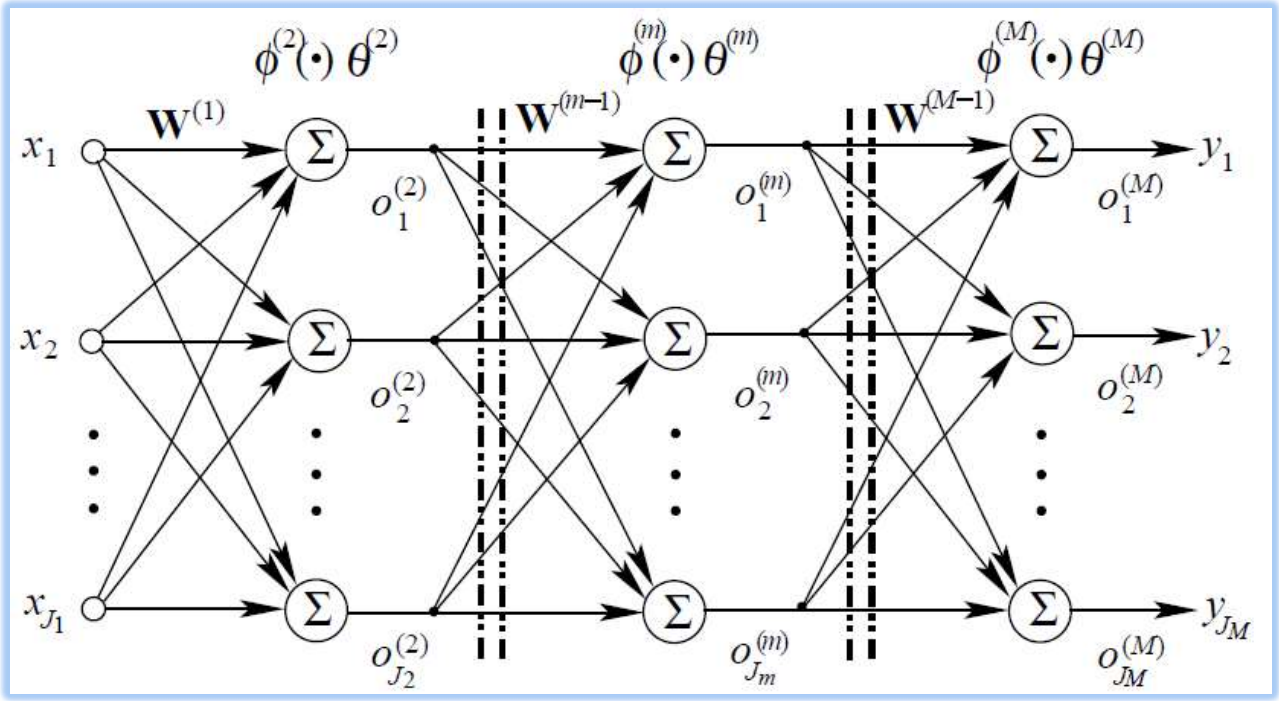
$$y = \phi(W^T x + b), \quad z = \tilde{\phi}(\tilde{W}^T y + \tilde{b})$$

$$z = \tilde{\phi}(\tilde{W}^T \phi(W^T x + b) + \tilde{b})$$

$$= N(x; \theta), \quad \theta = \{W, b, \tilde{W}, \tilde{b}\}.$$

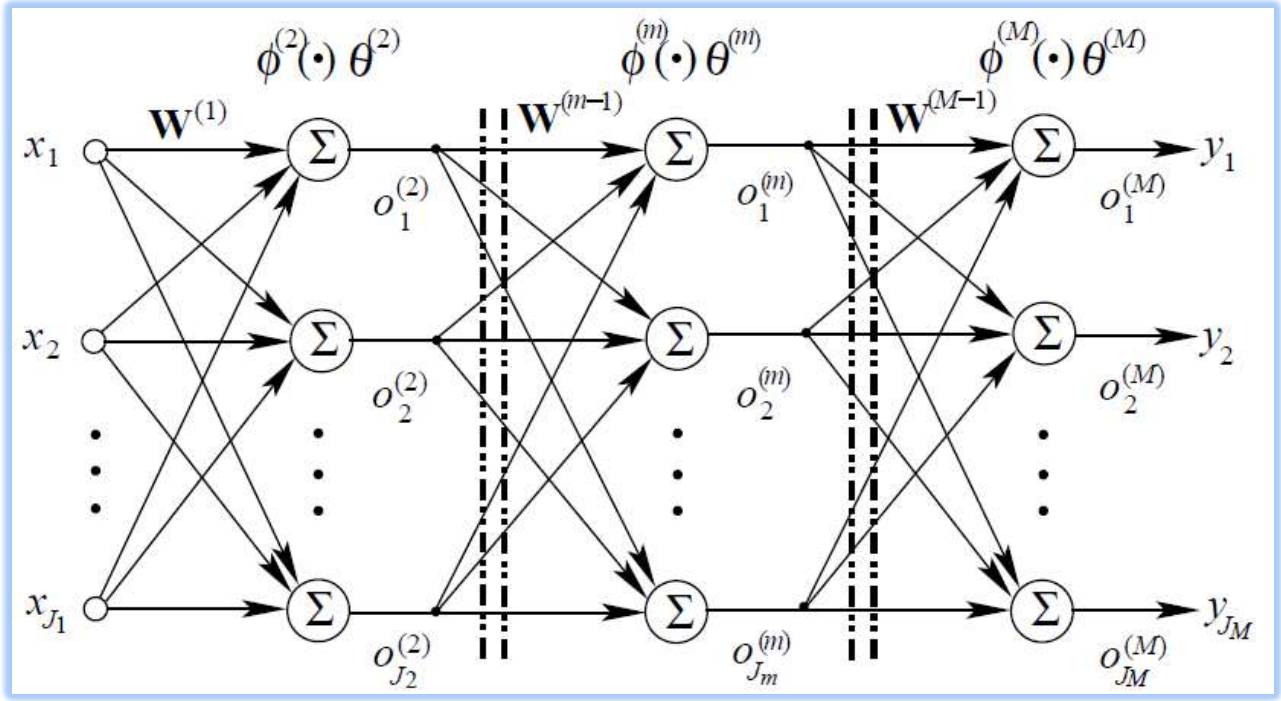


人工神经网络(artificial neural network,简称ANN)是由若干个人工神经元相互连接组成的广泛并行互联的网络.因连接方式的不同，有“前馈神经网络”和“反馈神经网络”.前馈网络因为其权值的训练中采用误差反向传播计算方式，所以亦称为反向传播(back propagation)神经网络，简称BP网.



patterns. The architecture of MLP is illustrated in Fig. 5.1. Assume that there are M layers, each having $J_m, m = 1, \ldots, M$, nodes. The weights from the $(m-1)$th layer to the mth layer are denoted by $\mathbf{W}^{(m-1)}$; the bias, output, and activation function of the ith neuron in the mth layer are, respectively, denoted as $\theta_i^{(m)}$, $o_i^{(m)}$, and $\phi_i^{(m)}(\cdot)$. An MLP trained with the BP algorithm is also called a *BP network*. MLP can be used for classification of linearly inseparable patterns and for function approximation.

- MLP： Multilayer Perceptrons（多层感知器）



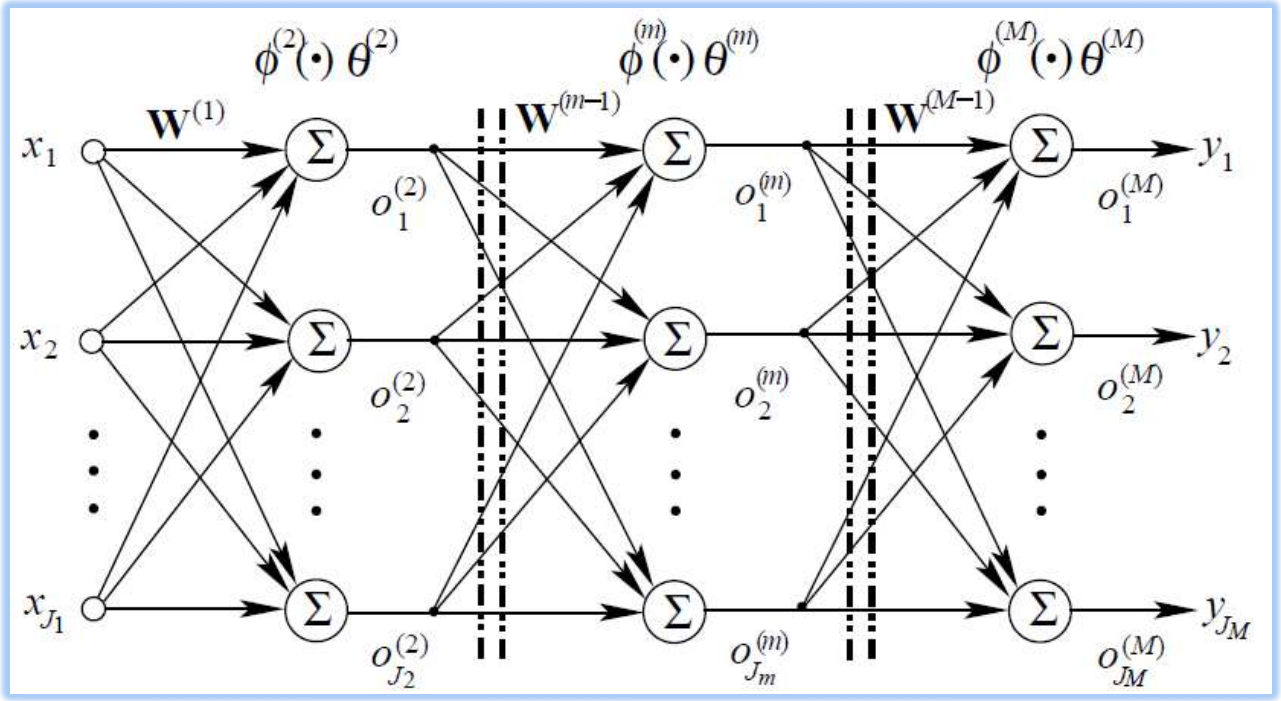
the bias vector for easy presentation. For $m = 2, \ldots, M$ and the pth example:

$$\hat{\boldsymbol{y}}_p = \boldsymbol{o}_p^{(M)}, \quad \boldsymbol{o}_p^{(1)} = \boldsymbol{x}_p, \tag{5.1}$$

$$\boldsymbol{net}_p^{(m)} = \left[\mathbf{W}^{(m-1)}\right]^T \boldsymbol{o}_p^{(m-1)} + \boldsymbol{\theta}^{(m)}, \tag{5.2}$$

$$\boldsymbol{o}_p^{(m)} = \boldsymbol{\phi}^{(m)}\left(\boldsymbol{net}_p^{(m)}\right), \tag{5.3}$$

where $\boldsymbol{net}_p^{(m)} = \left(net_{p,1}^{(m)}, \ldots, net_{p,J_m}^{(m)}\right)^T$, $\mathbf{W}^{(m-1)}$ is a J_{m-1}-by-J_m matrix, $\boldsymbol{o}_p^{(m-1)} = \left(o_{p,1}^{(m-1)}, \ldots, o_{p,J_{m-1}}^{(m-1)}\right)^T$, $\boldsymbol{\theta}^{(m)} = \left(\theta_1^{(m)}, \ldots, \theta_{J_m}^{(m)}\right)^T$ is the bias vector, and $\boldsymbol{\phi}^{(m)}(\cdot)$ applies $\phi_i^{(m)}(\cdot)$ to the ith component of the vector within.



$$\hat{y}_4 = \phi^{(4)}\left(\left[W^{(3)}\right]^T \phi^{(3)}\left(\left[W^{(2)}\right]^T \phi^{(2)}\left(\left[W^{(1)}\right]^T x + \theta^{(2)}\right) + \theta^{(3)}\right) + \theta^{(4)}\right)$$

$$= N(x;\theta), \quad \theta = \left\{W^{(1)}, W^{(2)}, W^{(3)}, \theta^{(2)}, \theta^{(3)}, \theta^{(4)}\right\}$$

Back Propagation

Backpropagation Learning Algorithm

- BP algorithm propagates backward the error between the desired signal and the network output through the network. After providing an input pattern, the output of the network is then compared with a given target pattern and the error of each output unit calculated. This error signal is propagated backward, and a closed-loop control system is thus established. The weights can be adjusted by a gradient-descent based algorithm.

The objective function for optimization is defined as the MSE between the actual network output $\hat{\boldsymbol{y}}_p$ and the desired output $\boldsymbol{y}_p$ for all the training pattern pairs $(\boldsymbol{x}_p, \boldsymbol{y}_p) \in \mathcal{S}$

$$E = \frac{1}{N} \sum_{p \in \mathcal{S}} E_p = \frac{1}{2N} \sum_{p \in \mathcal{S}} \left\| \hat{\boldsymbol{y}}_p - \boldsymbol{y}_p \right\|^2, \tag{5.4}$$

where N is the size of the sample set, and

$$E_p = \frac{1}{2} \left\| \hat{\boldsymbol{y}}_p - \boldsymbol{y}_p \right\|^2 = \frac{1}{2} \boldsymbol{e}_p^T \boldsymbol{e}_p, \tag{5.5}$$

$$\boldsymbol{e}_p = \hat{\boldsymbol{y}}_p - \boldsymbol{y}_p, \tag{5.6}$$

where the ith element of $\boldsymbol{e}_p$ is $e_{p,i} = \hat{y}_{p,i} - y_{p,i}$. Notice that a factor $\frac{1}{2}$ is used in E_p for the convenience of derivation.

All the network parameters $\mathbf{W}^{(m-1)}$ and $\boldsymbol{\theta}^{(m)}, m = 2, \ldots, M$, can be combined and represented by the matrix $\mathbf{W} = \left[w_{ij} \right]$. The error function E or E_p can be minimized by applying the gradient-descent procedure. When minimizing E_p, we have

$$\Delta_p \mathbf{W} = -\eta \frac{\partial E_p}{\partial \mathbf{W}}, \tag{5.7}$$

where η is the learning rate or step size, provided that it is a sufficiently small positive number. Note that the gradient term $\frac{\partial E_p}{\partial \mathbf{W}}$ is a matrix whose (i, j)th entry is $\frac{\partial E_p}{\partial w_{ij}}$.

$$\hat{\boldsymbol{y}}_p = \boldsymbol{o}_p^{(M)}, \quad \boldsymbol{o}_p^{(1)} = \boldsymbol{x}_p,$$

$$\boldsymbol{net}_p^{(m)} = \left[\mathbf{W}^{(m-1)}\right]^T \boldsymbol{o}_p^{(m-1)} + \boldsymbol{\theta}^{(m)},$$

$$\boldsymbol{o}_p^{(m)} = \boldsymbol{\phi}^{(m)}\left(\boldsymbol{net}_p^{(m)}\right),$$

Applying the chain rule, the derivative in (5.7) can be expressed as

$$\frac{\partial E_p}{\partial w_{uv}^{(m)}} = \frac{\partial E_p}{\partial net_{p,v}^{(m+1)}} \frac{\partial net_{p,v}^{(m+1)}}{\partial w_{uv}^{(m)}}. \tag{5.8}$$

The second factor of (5.8) is derived from (5.2)

$$\frac{\partial net_{p,v}^{(m+1)}}{\partial w_{uv}^{(m)}} = \frac{\partial}{\partial w_{uv}^{(m)}} \left(\sum_{\omega=1}^{J_m} w_{\omega v}^{(m)} o_{p,\omega}^{(m)} + \theta_v^{(m+1)} \right) = o_{p,u}^{(m)}. \tag{5.9}$$

The first factor of (5.8) can again be derived using the chain rule

$$\frac{\partial E_p}{\partial net_{p,v}^{(m+1)}} = \frac{\partial E_p}{\partial o_{p,v}^{(m+1)}} \frac{\partial o_{p,v}^{(m+1)}}{\partial net_{p,v}^{(m+1)}} = \frac{\partial E_p}{\partial o_{p,v}^{(m+1)}} \dot{\phi}_v^{(m+1)}\left(net_{p,v}^{(m+1)}\right), \tag{5.10}$$

where (5.3) is used. To solve the first factor of (5.10), we need to consider two situations for the output units ($m = M - 1$) and for the hidden units ($m = 1, \ldots, M - 2$):

$$\frac{\partial E_p}{\partial o_{p,v}^{(m+1)}} = e_{p,v}, \quad m = M - 1, \tag{5.11}$$

$$\begin{aligned}
\frac{\partial E_p}{\partial o_{p,v}^{(m+1)}} &= \sum_{\omega=1}^{J_{m+2}} \left(\frac{\partial E_p}{\partial net_{p,\omega}^{(m+2)}} \frac{\partial net_{p,\omega}^{(m+2)}}{\partial o_{p,v}^{(m+1)}} \right) \\
&= \sum_{\omega=1}^{J_{m+2}} \left[\frac{\partial E_p}{\partial net_{p,\omega}^{(m+2)}} \frac{\partial}{\partial o_{p,v}^{(m+1)}} \left(\sum_{u=1}^{J_{m+1}} w_{u\omega}^{(m+1)} o_{p,u}^{(m+1)} + \theta_{\omega}^{(m+2)} \right) \right] \\
&= \sum_{\omega=1}^{J_{m+2}} \frac{\partial E_p}{\partial net_{p,\omega}^{(m+2)}} w_{v\omega}^{(m+1)}, \quad m = 1, \ldots, M - 2.
\end{aligned} \tag{5.12}$$

Define the delta function by

$$\delta_{p,v}^{(m)} = -\frac{\partial E_p}{\partial net_{p,v}^{(m)}}, \quad m = 2, \ldots, M. \tag{5.13}$$

By substituting (5.8), (5.12), and (5.13) into (5.10), we finally obtain for the output units ($m = M - 1$) and for the hidden units ($m = 1, \ldots, M - 2$):

$$\delta_{p,v}^{(M)} = -e_{p,v}\dot{\phi}_v^{(M)}\left(net_{p,v}^{(M)}\right), \quad m = M - 1, \tag{5.14}$$

$$\delta_{p,v}^{(m+1)} = \dot{\phi}_v^{(m+1)}\left(net_{p,v}^{(m+1)}\right)\sum_{\omega=1}^{J_{m+2}}\delta_{p,\omega}^{(m+2)}w_{v\omega}^{(m+1)}, \quad m = 1, \ldots, M - 2. \tag{5.15}$$

Equations (5.14) and (5.15) provide a recursive method to solve $\delta_{p,v}^{(m+1)}$ for the whole network. Thus, $\mathbf{W}$ can be adjusted by

$$\frac{\partial E_p}{\partial w_{uv}^{(m)}} = -\delta_{p,v}^{(m+1)}o_{p,u}^{(m)}. \tag{5.16}$$

For the activation functions, we have the following relations:

$$\dot{\phi}(net) = \beta\phi(net)\left[1 - \phi(net)\right], \quad \text{for logistic function,} \tag{5.17}$$

$$\dot{\phi}(net) = \beta\left[1 - \phi^2(net)\right], \quad \text{for tanh function.} \tag{5.18}$$

The update for the biases can be in two ways. The biases in the $(m+1)$th layer $\boldsymbol{\theta}^{(m+1)}$ can be expressed as the expansion of the weight $\mathbf{W}^{(m)}$, that is, $\boldsymbol{\theta}^{(m+1)} = \left(w_{0,1}^{(m)}, \ldots, w_{0,J_{m+1}}^{(m)}\right)^T$. Accordingly, the output $\boldsymbol{o}^{(m)}$ is expanded into $\boldsymbol{o}^{(m)} = \left(1, o_1^{(m)}, \ldots, o_{J_m}^{(m)}\right)^T$. Another way is to use a gradient-descent method with regard to $\boldsymbol{\theta}^{(m)}$, by following the above procedure. Since the biases can be treated as special weights, these are usually omitted in practical applications.

The BP algorithm is defined by (5.7), and is rewritten here as follows:

$$\Delta_p \mathbf{W}(t) = -\eta \frac{\partial E_p}{\partial \mathbf{W}}. \tag{5.19}$$

The algorithm is convergent in the mean if $0 < \eta < \frac{2}{\lambda_{\max}}$, where $\lambda_{\max}$ is the largest eigenvalue of the autocorrelation of the vector $\boldsymbol{x}$, denoted by $\mathbf{R}$ [144]. When η is too small, the possibility of getting stuck at a local minimum of the error function is increased. In contrast, the possibility of falling into oscillatory traps is high when η is too large. By statistically preprocessing the input patterns, namely, decorrelating

Algorithm 5.1 (BP for a three-layer MLP).

All units have the same activation function $\phi(\cdot)$, and all biases are absorbed into weight matrices.

1. *Initialize* $\mathbf{W}^{(1)}$ *and* $\mathbf{W}^{(2)}$.
2. *Calculate* E *using (5.4).*
3. ***for*** *each epoch:*
   - *Calculate* E *using (5.4).*
   - ***if*** E *is less than a threshold* ϵ, ***return***.
   - ***for*** *each* $\boldsymbol{x}_p$, $p = 1, \ldots, N$:
     a. *Forward pass*
        i. *Compute* $\boldsymbol{net}_p^{(2)}$ *by (5.2) and* $\boldsymbol{o}_p^{(2)}$ *by (5.3).*
        ii. *Compute* $\boldsymbol{net}_p^{(3)}$ *by (5.2) and* $\hat{\boldsymbol{y}}_p = \boldsymbol{o}_p^{(3)}$ *by (5.3).*
        iii *Compute* $\boldsymbol{e}_p$ *by (5.6).*
     b. *Backward pass,* ***for*** *all neurons*
        i. *Compute* $\delta_{p,v}^{(3)} = -e_{p,v}\dot{\phi}\left(net_{p,v}^{(3)}\right)$
        ii. *Update* $\mathbf{W}^{(2)}$ *by* $\Delta w_{uv}^{(2)} = \eta\delta_{p,v}^{(3)} o_{p,u}^{(2)}$
        iii *Compute* $\delta_{p,v}^{(2)} = \left(\sum_{\omega=1}^{J_3} \delta_{p,\omega}^{(3)} w_{v\omega}^{(2)}\right) \dot{\phi}\left(net_{p,v}^{(2)}\right)$
        iv. *Update* $\mathbf{W}^{(1)}$ *by* $\Delta w_{uv}^{(1)} = \eta\delta_{p,v}^{(2)} o_{p,u}^{(1)}$
4. ***end***

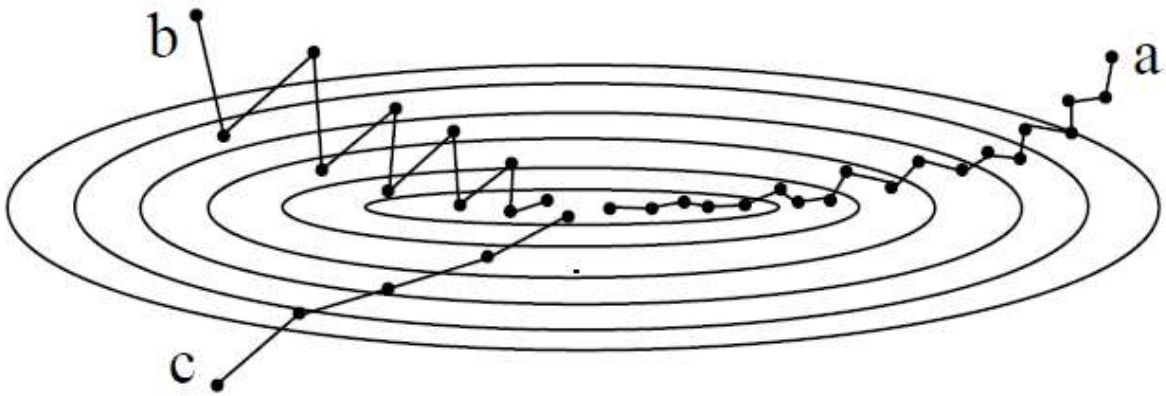


Fig. 5.2 Descent in weight space. **a** For small learning rate; **b** for large learning rate; **c** for large learning rate with momentum term added

The BP algorithm can be improved by adding a momentum term [108]

$$\Delta_p \mathbf{W}(t) = -\eta \frac{\partial E_p(t)}{\partial \mathbf{W}(t)} + \alpha \Delta \mathbf{W}(t-1), \tag{5.20}$$

where α is the momentum factor, usually $0 < \alpha \leq 1$. The typical value for α is 0.9. This method is usually called the *BP with momentum*. The momentum term can effectively magnify the descent in almost-flat steady downhill regions of the error surface by $\frac{1}{1-\alpha}$. In regions with high fluctuations (due to high learning rates), the momentum has a stabilizing effect. The momentum term actually inserts second-order information in the training process that performs like the CG method. The momentum term effectively smoothens the oscillations and accelerates the convergence. The role of the momentum term is shown in Fig. 5.2. BP with momentum is analyzed and the

Brief Summary of Deep Learning

DNN vs. Polynomial Approximation

$x \xrightarrow{f(x)} y$, $f(x) \in L^2(\Omega)$ is unknown

- Choose a finite dimensional subspace $V \subset L^2(\Omega)$.
- Slect a set of basis functions (e.g. polynomials)

$$\{\phi_i(x),\ i=1,2,\cdots,N\}$$

- Let $f(x) = \sum_{i=1}^{n} \theta_i \phi_i(x) \approx f(x)$. Find θ_i such that

$$\sum_{i=1}^{n} \theta_i \phi_i(x^{(j)}) \approx f(x^{(j)}) = y^{(j)},\quad 1 \le j \le J$$

DNN vs. Polynomial Approximation

$$\min_{\theta} \frac{1}{2} \sum_{j=1}^{J} \left| \sum_{i=1}^{n} \theta_i \phi_i(x^{(j)}) - y^{(j)} \right|^2$$

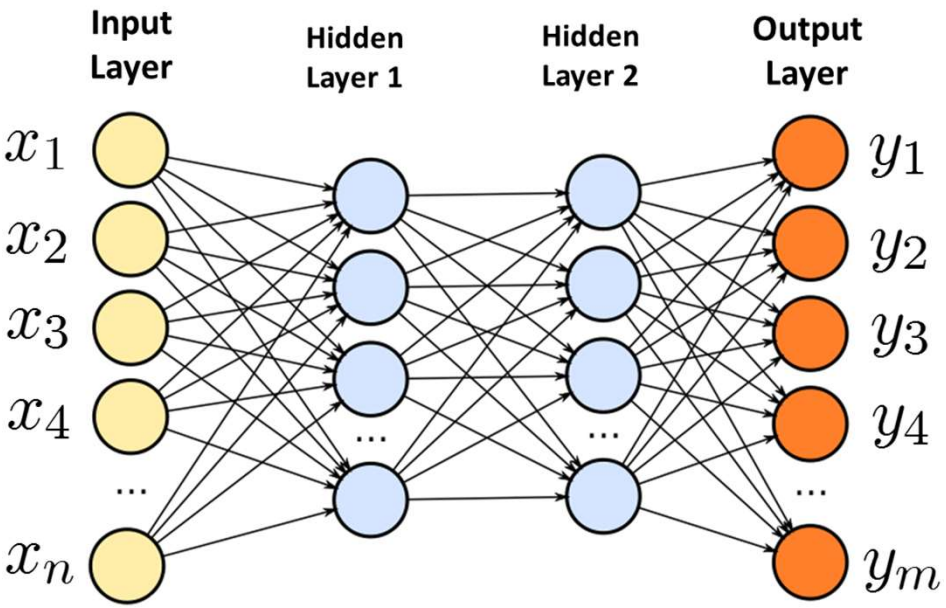
令 $A = (a_{ji}) = (\phi_i(x^{(j)}))_{\substack{1 \le i \le n \\ 1 \le j \le J}}$ $\quad Y = (y^{(j)})_{1 \le j \le J}$

$$\Longleftrightarrow \quad \min_{\theta} \frac{1}{2} \| A\theta - Y \|^2 ,$$

$J = n$ → 插值

$J > n$ → Least square

Deep Neural Network (DNN)



$$\mathbf{y}^{out} = \mathbf{N}(\mathbf{x}^{int}; \Theta)$$

- **Given a dataset of input and output pairs**

$$\left(\mathbf{x}^{(j)}, \mathbf{y}^{(j)}\right), \quad j = 1, 2, \ldots, J.$$

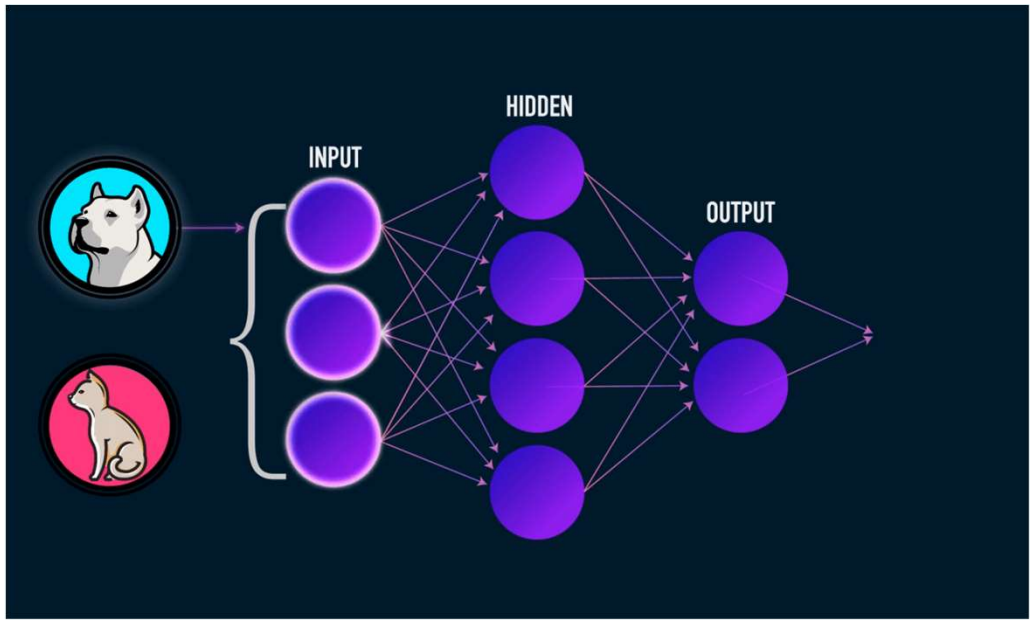
- **Deep learning tries to approximate the association between inputs and outputs**

$$\mathbf{N}(\mathbf{x}^{(j)}; \Theta_*) \approx \mathbf{y}^{(j)},$$

by minimizing

$$L(\Theta) := \frac{1}{J} \sum_{j=1}^{J} \left\| \mathbf{N}(\mathbf{x}^{(j)}; \Theta) - \mathbf{y}^{(j)} \right\|_2^2.$$

Deep Neural Network (DNN)



- **Given a dataset of input and output pairs**

$$\left(\mathbf{x}^{(j)}, \mathbf{y}^{(j)}\right), \quad j = 1, 2, \ldots, J.$$

- **Deep learning tries to approximate the association between inputs and outputs**

$$\mathbf{N}(\mathbf{x}^{(j)}; \Theta_*) \approx \mathbf{y}^{(j)},$$

by minimizing

$$L(\Theta) := \frac{1}{J} \sum_{j=1}^{J} \left\| \mathbf{N}(\mathbf{x}^{(j)}; \Theta) - \mathbf{y}^{(j)} \right\|_2^2.$$

$$\mathbf{y}^{out} = \mathbf{N}(\mathbf{x}^{int}; \Theta)$$

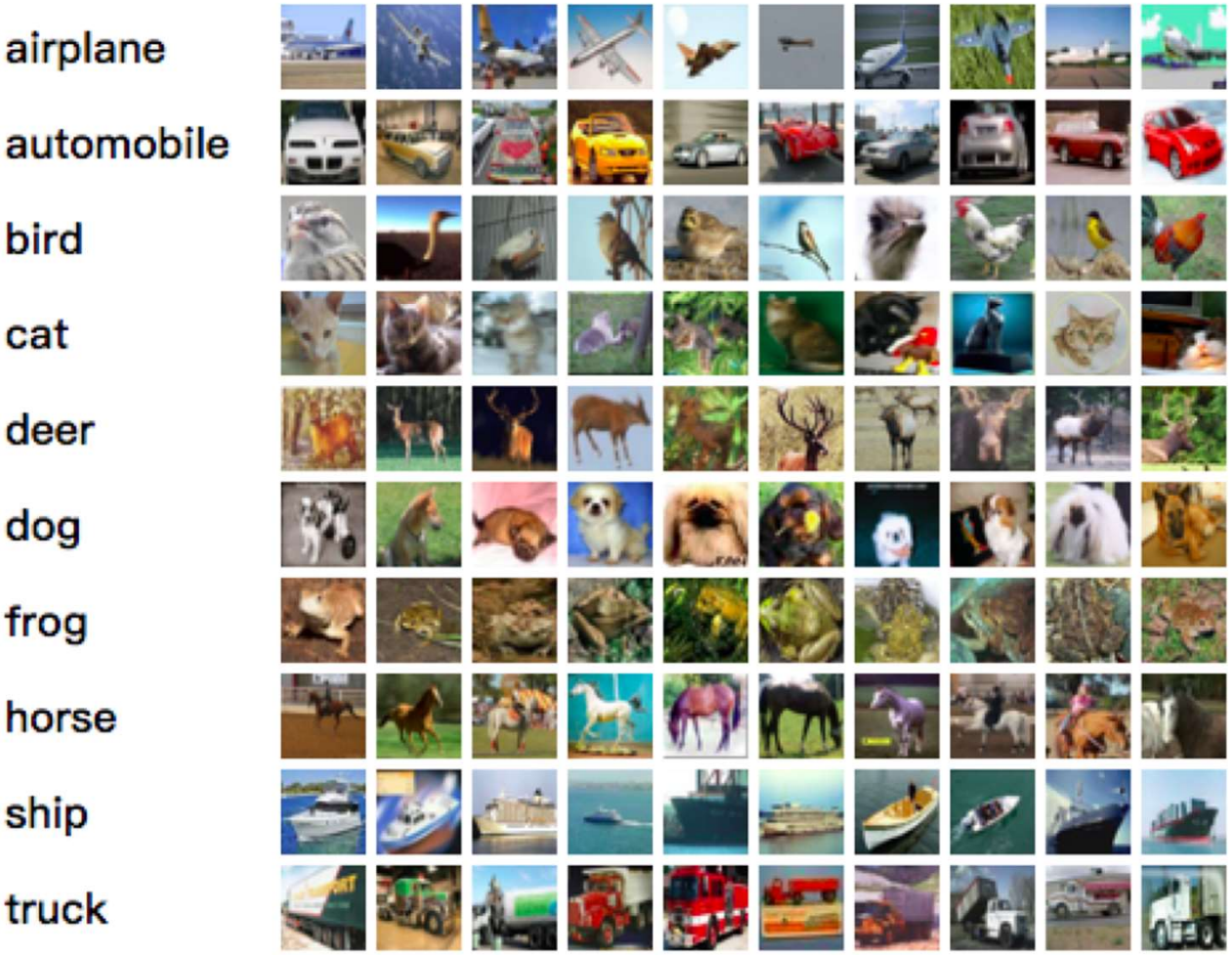
Dog =1, Cat =0

Images (tensor)

ML can do wonders: Approximating high dimensional functions

Given $S = \{(x_j, y_j = f^*(x_j)), j \in [n]\}$, learn (i.e. approximate) f^*.

Example: Cifar 10 dataset (f^* is a discrete function defined on the space of images)



- Input: each image $\in [0,1]^d$, $d = 32 \times 32 \times 3 = 3072$.
- Output: $f^* \in \{$airplane, ... , truck$\}$.
- $f^* : [0,1]^{3072} \to \{$airplane, ... , truck$\}$.
  $f^*($each image$)$ = category

Solving high dimensional Bellman equations

The optimal strategy obeys some Bellman-like equation.

All these are made possible by our ability to accurately approximate high dimensional functions using finite pieces of data.

This opens up new possibilities for attacking problems that suffer from the “curse of dimensionality" (CoD):

As dimensionality grows, computational cost grows exponentially fast.

DNN as A Function Approximator

- **Unknown target function**

$$\mathbf{y} = \mathbf{f}(\mathbf{x})$$

- **Data --- the samples of f(x)**

$$\mathbf{y}^{(j)} = \mathbf{f}(\mathbf{x}^{(j)}) + \epsilon_j, \quad j = 1, 2, \ldots, J.$$

- **The trained DNN is an approximation to f(x):**

$$\mathbf{N}(\bullet; \Theta_*) \approx \mathbf{f}$$

- **Given a dataset of input and output pairs**

$$\left(\mathbf{x}^{(j)}, \mathbf{y}^{(j)}\right), \quad j = 1, 2, \ldots, J.$$

- **Deep learning tries to approximate the association between inputs and outputs**

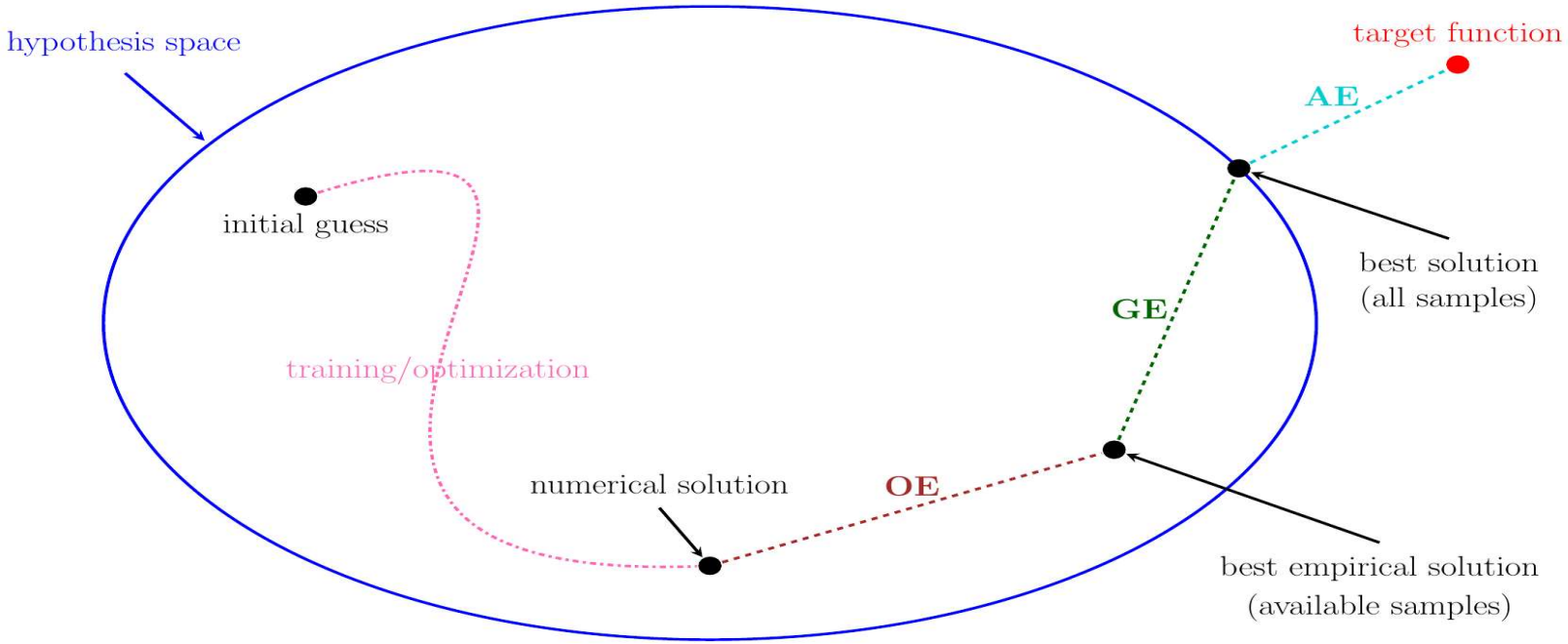
$$\mathbf{N}(\mathbf{x}^{(j)}; \Theta_*) \approx \mathbf{y}^{(j)},$$

by minimizing

$$L(\Theta) := \frac{1}{J} \sum_{j=1}^{J} \left\| \mathbf{N}(\mathbf{x}^{(j)}; \Theta) - \mathbf{y}^{(j)} \right\|_2^2.$$

Deep Learning's Ability

Network structure（表征误差） + optimization algorithm（优化误差） + data accuracy & distribution（泛化误差）



https://shijunzhang.top/research/

Deep Learning's Ability

Network structure（表征误差） + optimization algorithm（优化误差） + data accuracy & distribution（泛化误差）

$$\mathcal{E} := \|\tilde{u}_{\mathcal{T}} - u\| \leq \underbrace{\|\tilde{u}_{\mathcal{T}} - u_{\mathcal{T}}\|}_{\mathcal{E}_{\text{opt}}} + \underbrace{\|u_{\mathcal{T}} - u_{\mathcal{F}}\|}_{\mathcal{E}_{\text{gen}}} + \underbrace{\|u_{\mathcal{F}} - u\|}_{\mathcal{E}_{\text{app}}}.$$

The approximation error $\mathcal{E}_{\text{app}}$ measures how closely $u_{\mathcal{F}}$ can approximate u. The generalization error $\mathcal{E}_{\text{gen}}$ is determined by the number/locations of residual points in $\mathcal{T}$ and the capacity of the family $\mathcal{F}$. Neural networks of larger size have smaller approximation errors but could lead to higher generalization errors, which is called bias-variance tradeoff. Overfitting occurs when the generalization error dominates. In addition, the optimization error $\mathcal{E}_{\text{opt}}$ stems from the loss function complexity and the optimization setup, such as learning rate and number of iterations. However,

Deep Learning's Ability

Network structure（表征误差） + optimization algorithm（优化误差） + data accuracy & distribution（泛化误差）

Many empirical successes. Theoretical understanding is still quite limited.

- **Universal Approximation Theorem:**

  Sufficiently large neural networks (with non-polynomial activation)

  can represent a large class of functions, if given suitable parameters.

- **However, it does not provide any knowledge for understanding how to find those parameters.**

Deep Learning's Ability

Network structure（表征误差） + optimization algorithm（优化误差） + data accuracy & distribution（泛化误差）

Many empirical successes.

DNNs have been found to be very efficient for function approximation in high-dimensional spaces.

- **It is believed that the approximate ability of DNN is independent of the input dimensionality, while conventional linear regression methods suffer from the curse of dimensionality, which results in a decrease of the convergence rate with an increase of the input dimensionality.**

Training of DNN

- **Training** --- the process of numerically minimizing

$$L(\Theta) := \frac{1}{J}\sum_{j=1}^{J}\left\|\mathbf{N}(\mathbf{x}^{(j)};\Theta)-\mathbf{y}^{(j)}\right\|_2^2 := \frac{1}{J}\sum_{j=1}^{J} L_i(\Theta).$$

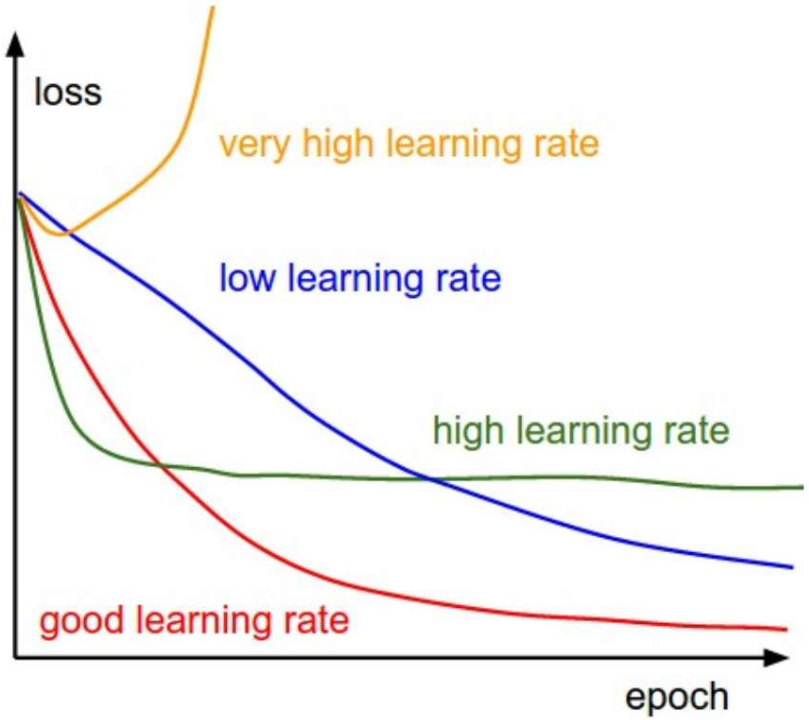
- **Gradient Descent (GD):**

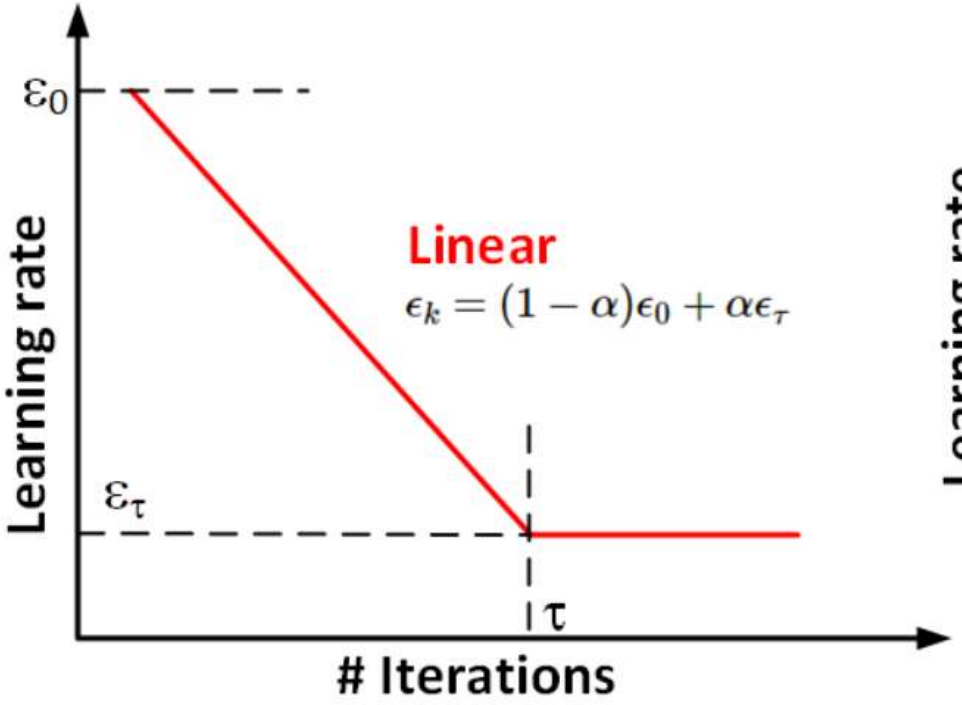
$$\Theta^{\text{new}} \longleftarrow \Theta - \eta\nabla L(\Theta)$$

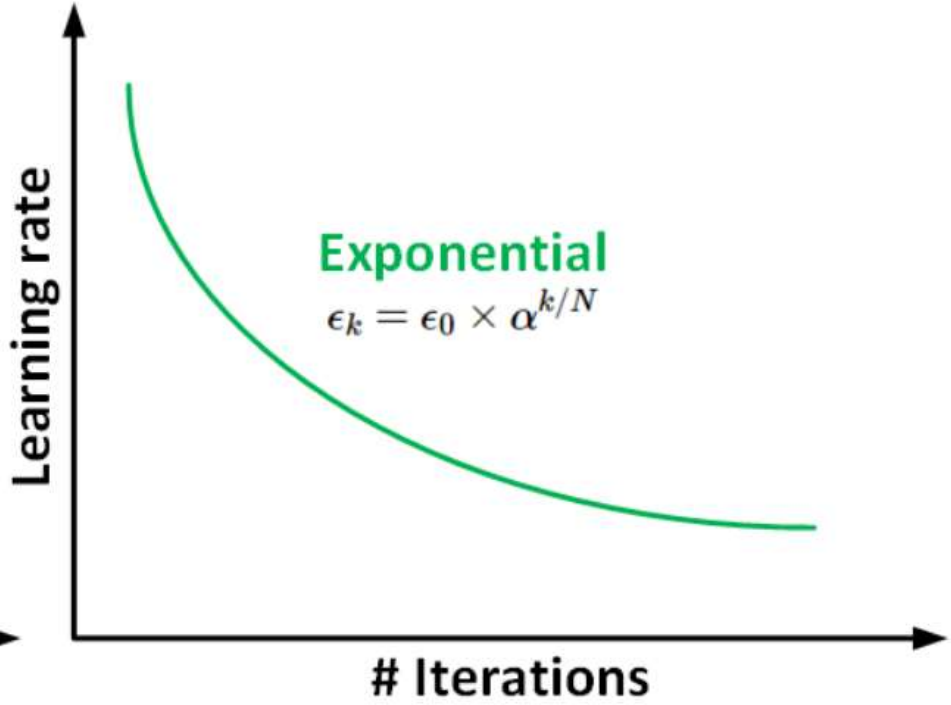
- **Stochastic GD (SGD):**
  - Randomly shuffle the data pairs in the training set.
  - Go though each data pair loss: $\text{for} \quad i \text{ from } 1 \text{ to } J$

$$\Theta^{\text{new}} \longleftarrow \Theta - \eta\nabla L_i(\Theta)$$

- A variant of **SGD** --- **Adam** (adaptive learning rate)

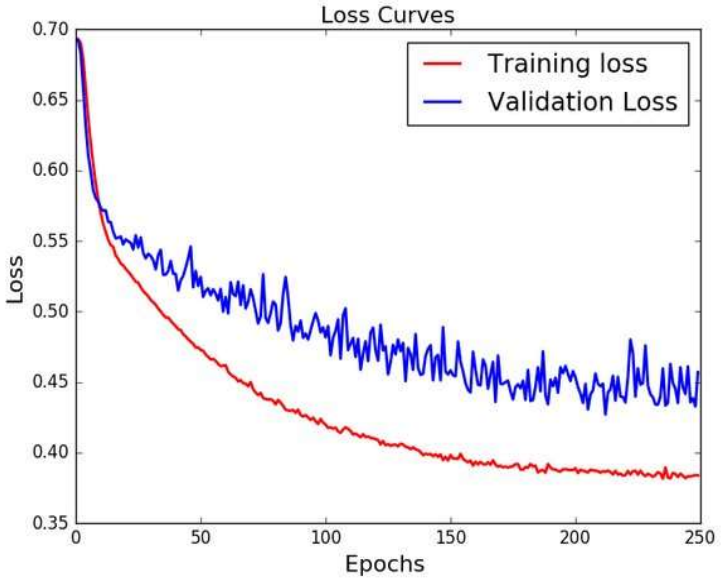

loss
very high learning rate
low learning rate
high learning rate
good learning rate
epoch

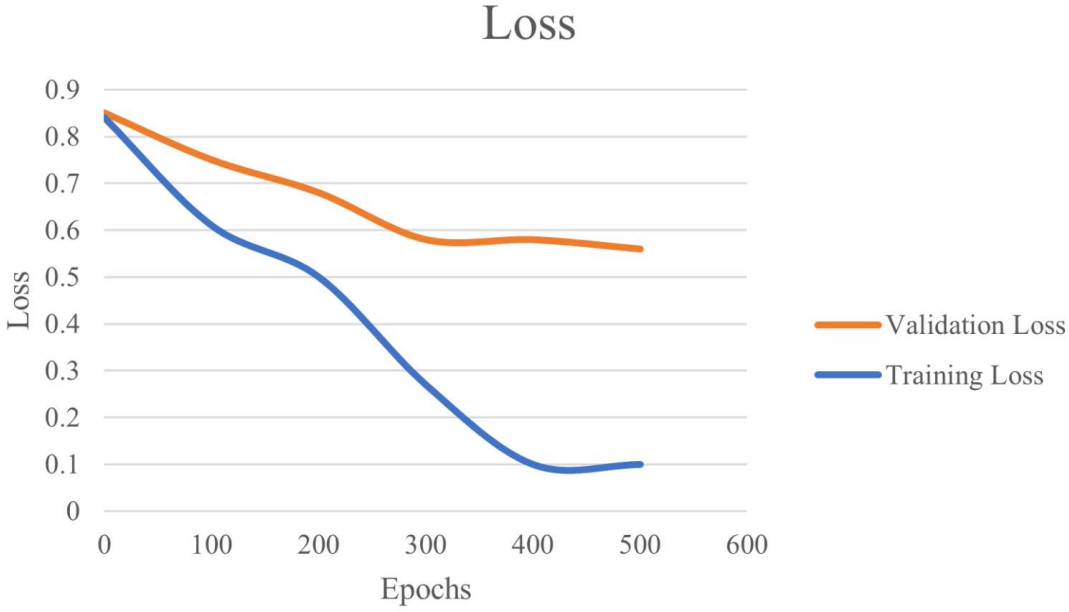

ε_0
Learning rate
Linear
$\epsilon_k = (1-\alpha)\epsilon_0 + \alpha\epsilon_\tau$
ε_τ
τ
Iterations


Learning rate
Exponential
$\epsilon_k = \epsilon_0 \times \alpha^{k/N}$
Iterations

Other Related Concepts

- **Generalization (泛化)** refers to your model's ability to adapt properly to new, previously unseen data, drawn from the same distribution as the one used to create the model.
- **Overfitting (过拟合)** is a concept in data science, which occurs when a statistical model fits exactly against its training data. When this happens, the algorithm





Code

Suggested:

- Python
  - Tensorflow / Keras
  - PyTorch
- Jupyter Notebook
- Matlab

Deep Learning for Science

- Approximate Functions/Operators
- Solve Governing Equations
- Learn/Discover Equations (Data-driven modeling)
- Learn to discrete (Learning numerical schemes)
- Many others…

Sampling unknown high dimensional distributions

The probability distribution of all real and fake human faces is an unknown distribution in high dimension.