第四章 神经网络

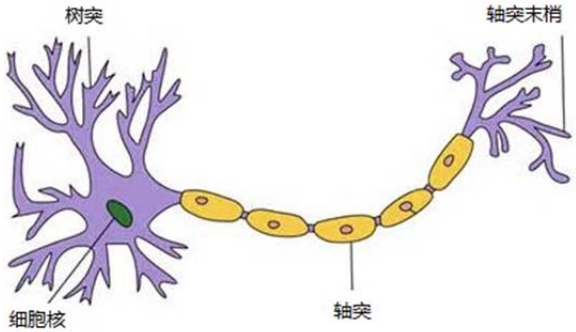
1 概述

1.1 大脑和神经元(Brain and Neurons)

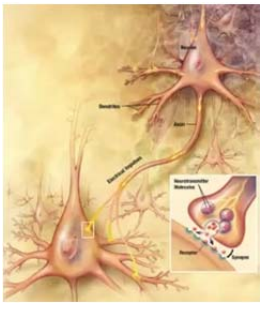
神经网络是最初产生的目的是制造能模拟大脑的机器。神经网络逐渐兴起于二十世纪八九十年代，当时应用得非常广泛，但由于各种原因，在 90 年代的后期应用减少了。但是最近，神经网络又东山再起了， 其中一个原因是：尽管神经网络是一个计算量有些偏大的算法，但近些年来计算机的运行速度变快，使得大规模的神经网络算法得以运行，如今的神经网络如**深度神经网络**对于许多应用来说是最先进的技术。

如果我们能找出大脑的学习算法 ,然后在计算机上执行大脑学习算法或与之相似的算法，也许这将是我们向人工智能迈进做出的最好的尝试。人工智能的梦想就是：有一天能制造出真正的智能机器。

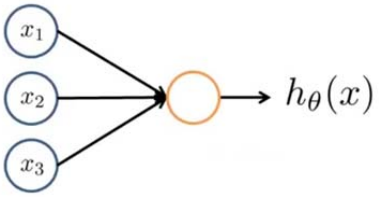
神经网络是在模拟大脑中的神经元时发明的。我们首先看看神经元在大脑中是什么样子。在我们的大脑中，充满了如下图所示的神经元，每个神经元都有一个细胞核、一定数量的输入神经（树突 Dendrite）、一个输出神经（轴突 Axon）。简单来讲，神经元是一个计算单元，它从输入神经接收一定数目的信息，并作出一些计算，将计算结果通过轴突传送到其他神经元。



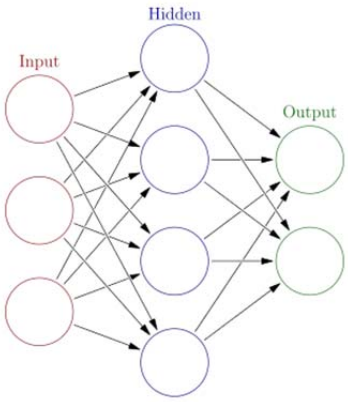
下图是一组神经元的示意图，神经元利用微弱的电流进行沟通，这些弱电流也称作**动作电位**。如果神经元想要传递一个消息 ,它就会通过它的轴突发送一段微弱电流给其他神经元，这个神经元接收到这条信息，作出一些计算，再将计算的信息传送给其他的神经元。其实，这就是人类思考的模型：神经元把自己的收到的消息进行计算，并向其他神经元传递消息。这也是我们的感觉和肌肉的运转原理。例如，如果你想活动一块肌肉，就会触发一个神经元给你的肌肉发送脉冲，并引起你的肌肉收缩，如果一些感官（例如眼睛）想要给大脑传送一个消息，那么它也会这样发送电脉冲给大脑。



在一个人工神经网络中 ,我们可以使用一个非常简单的模型来模拟神经元的工作。将神经元模拟为一个**逻辑单元**，如下图所示，中间的黄色圈圈可以看做类似于神经元的东西 ,然后通过它的树突（或者叫做它的输入神经 input wires）传递给该神经元一些消息，接下来该神经元做一些计算，并通过它的轴突（或叫做输出神经 output wires）输出计算结果。



该模型是一个简单的神经元模型，它的三个输入为 x_1, x_2, x_3 ，输出是类似于逻辑回归中的假设函数的形式 $h_\theta(x)$ ，即 $h_\theta(x) = \dfrac{1}{1+e^{-\theta^T x}}$ 。

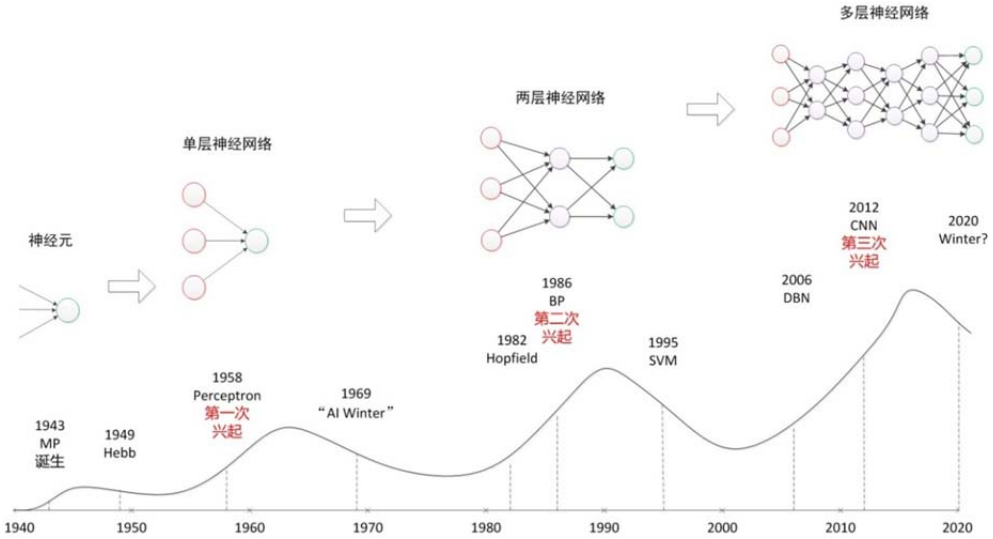


一个神经网络模型， 通常包含大量彼此连接的**节点** （神经元），通过某种特定的输出函数 （也叫**激活函数**），计算处理来自其它相邻神经元的加权输入值。神经元之间的信息传递的强度，用所谓**权重**来定义，算法会不断自我学习，调整这个**权重**。在此基础上，神经网络的计算模型，依靠大量的数据来训练，通过代价函数用来评估根据输入值计算出来的输出结果离真实值有多远。

随着神经网络研究的不断变迁，其计算特点和传统的生物神经元的连接模型渐渐脱钩。但是它保留的精髓是：非线性、分布式、并行计算、自适应、自组织。

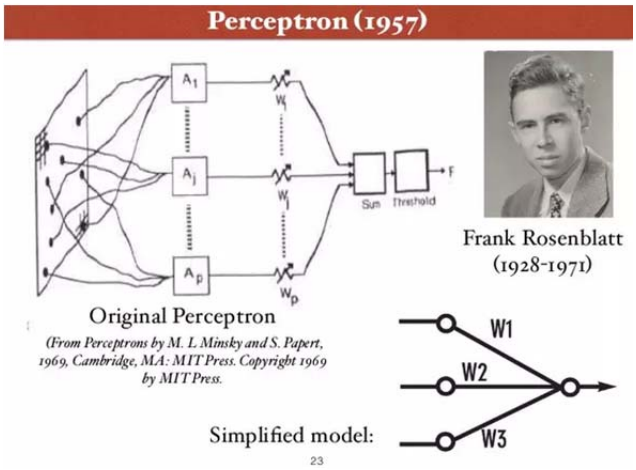
1.2 从神经网络到深度学习

从单层神经网络（感知器）开始，到包含一个隐层的两层神经网络，再到多层的深度神经网络，一共有三次兴起过程。详见下图。



神经网络作为一个计算模型的理论，1943 年 最初由科学家 Warren McCulloch 和 Walter Pitts 提出。

康内尔大学教授 Frank Rosenblatt 1957 年 提出的“感知器” （Perceptron），是第一个用算法来精确定义神经网络，第一个具有自组织自学习能力的数学模型，是日后许多新的神经网络模型的始祖。



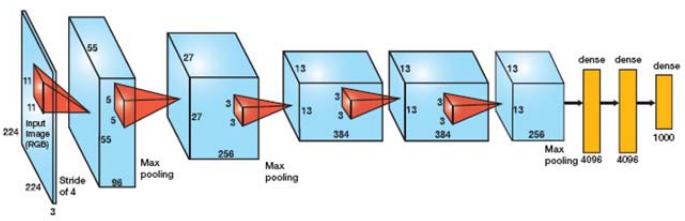
1969 年，Marvin Minsky 和 Seymour Papert 出版了新书：“感知器：计算几何简介”。书中论证了单层的神经网络无法解决不可线性分割的问题，典型例子如异或门。此后的十几年，以神经网络为基础的人工智能研究进入低潮。

1986 年 7 月，Hinton 和 David Rumelhart 合作在自然杂志上发表论文，“Learning Representations by Back-propagating errors”，第一次系统简洁地阐述反向传播算法在神经网络模型上的应用。神经网络的研究开始复苏。

九十年代左右，对于神经网络一个新的寒冬开始来临——社会对它们再次失去信心。一个新的方法，被称为**支持向量机**（Support Vector Machines），得到发展并且渐渐被发现是优于神经网络。

深度学习的概念由 **Hinton** 等人于 2006 年提出。深度学习的意思是建设巨大的神经网络。就是用很多层神经元构成的神经网络达到机器学习的功能。深度学习在图像识别、语音识别等方面取得了巨大的成功。神经网络再次成为研究热潮。

研究人员通过分析人脑的工作方式发现：通过感官信号从视网膜传递到前额大脑皮质再到运动神经的时间，推断出大脑皮质并未直接地对数据进行特征提取处理，而是使接收到的刺激信号通过一个复杂的层状网络模型，进而获取观测数据展现的规则。人类感知系统这种明确的层次结构极大地降低了视觉系统处理的数据量，并保留了物体有用的结构信息。深度学习正是希望通过模拟人脑多层次的分析方式来提高学习的准确性。

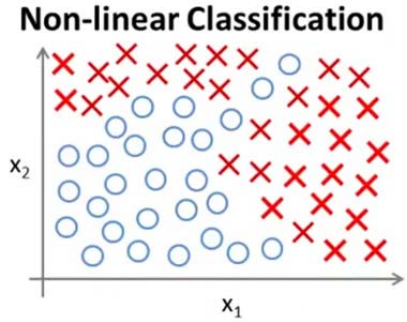


实际生活中，人们为了解决一个问题，如对象的分类（对象可是文档、图像等），首先必须做的事情是如何来表达一个对象，即必须抽取一些特征来表示一个对象，因此特征对结果的影响非常大。在传统的数据挖掘方法中，特征的选择一般都是通过手工完成的，通过手工选取的好处是可以借助人的经验或者专业知识选择出正确的特征；但缺点是效率低，而且在复杂的问题中，人工选择可能也会陷入困惑。深度学习能够通过多层次通过组合低层特征形成更抽象的高层特征，从而实现自动的学习特征，而不需要人参与特征的选取。

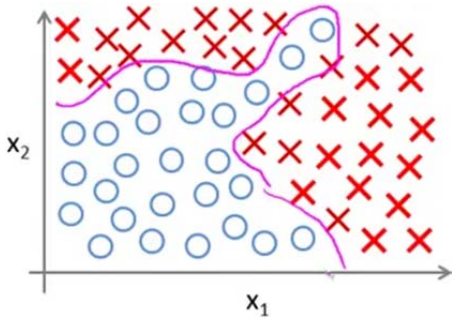
2012 年，斯坦福大学人工智能实验室主任 Andrew Ng（吴恩达）和谷歌合作建造了一个当时最大的深度学习神经网络。参数多达十七亿。后来 Ng 自己在斯坦福又搞了个更大的神经网络，参数更高达一百一十二亿。这个神经网络会用到大量的图形处理芯片 GPU，GPU 是模拟神经网络的完美硬件，因为每个 GPU 芯片内都有大量的小核心，这和神经网络的大规模并行性天然相似。

1.3 为什么需要神经网络

前面我们学习了数据分析中常用的方法：**线性回归**和**逻辑回归**。那为什么还需要神经网络这个东西呢？我们先来举一个分类问题的例子，假设有如下的训练集：



我们的目的是进行分类，假若如图所示的训练集一样，只有两个特征 x_1, x_2，通过逻辑回归很容易就能得到一个决策边界：



然而，当特征非常多的时候，我们必须用足够高次的项来保证假设函数的准确率，例如需要用 100 个特征来预测房价，且最高次项为 6 次，在逻辑回归中，假设函数为：

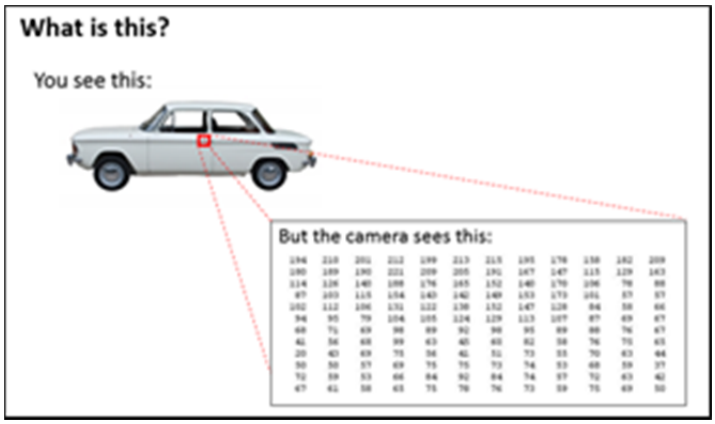
$$h_\theta(\boldsymbol{x}) = g(\boldsymbol{\theta}^T \boldsymbol{x})$$
$$= \frac{1}{1+e^{-\theta^T x}}$$

其中

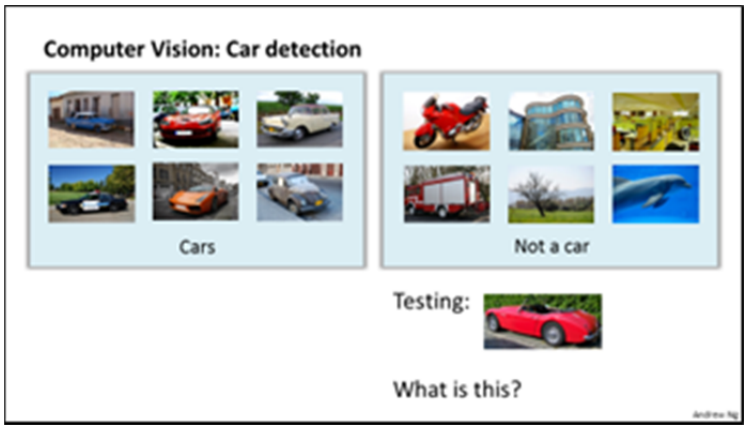
$$g(\boldsymbol{\theta}^T \boldsymbol{x}) = g(\theta_0 + \theta_1 x_1 + \theta_2 x_2 + \ldots + \theta_k x_1 x_2 + \theta_{k+1} x_1 x_3 + \ldots + \theta_n x_1^6 + \theta_{n+1} x_2^6 + \ldots)$$

从上面这个式子可以看出，如果要用**梯度下降法**或者**正规方程**方法来得到θ_j的值，需要巨大的计算量。因为就算只有 100 个特征，如果只用最高项为二次项的函数$g(\boldsymbol{\theta}^T \boldsymbol{x})$，$C_{100}^2 + 201$个$\theta$项。所以当遇到特征数很多的情况时，我们需要另外一种方法，也就是**神经网络(Neural Networks)**。下面的例子也进一步说明了神经网络的必要性。

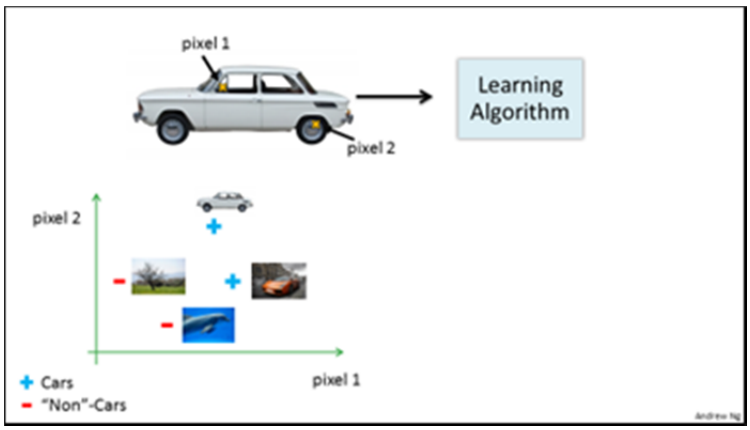
下图给出了一个计算机视觉中的问题：利用机器学习来训练一个分类器，它可以判定一副输入图像是否为一辆汽车。取出图像中的一个小矩形部分，将其放大，可以看到，人类眼中的汽车，计算机看到的却是一个数值矩阵（每个数值代表相应位置的灰度值），所以，这个识别问题对计算机而言，就变为了根据像素点亮度矩阵判断这个数值矩阵到底代表的是什么。



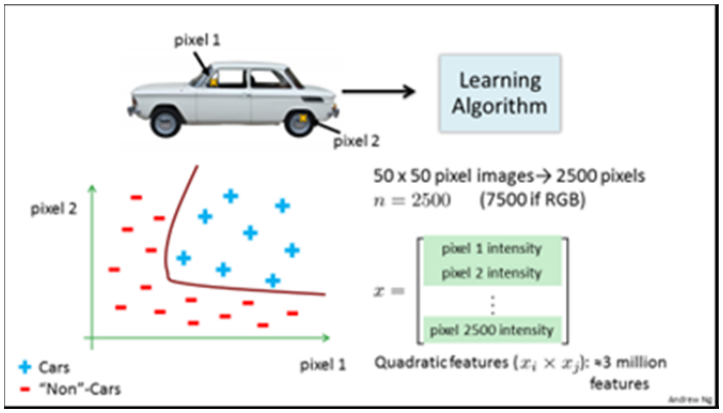
具体而言，用机器学习算法构造汽车识别器的基本过程是：将给定的一个带有标签的样本集（其中，一类是汽车样本，另一类是非汽车样本）输入到学习算法中，从而得到一个分类器； 对于给定的一个新的测试样本，该分类器就可以判断出"这是什么东西"。理想情况下，分类器可以识别出一个汽车。



为了描述引入非线性分类器的必要性，我们从训练样本集中挑出一些汽车图片和非汽车图片，从每组图像中挑出一组像素点（pixel1 和 pixel2 两个像素点），在坐标系中标出该汽车的位置（即为坐标系中的一个点，x 坐标为 pixel1 亮度值、y 坐标为 pixel2 亮度值），利用同样的方法，标出其它图片中的汽车的位置及非汽车物体的位置，绘制结果如下图所示。对于这种简答的情况，似乎结果还是可以线性分割的。



下面，继续绘制更多的样本点，绘制结果如下图所示。可以看到，汽车样本和非汽车样本分布在坐标系中的不同区域（一个非线性分类问题），所以，我们需要一个非线性分类器将这两类样本尽量区分开来（下图绘制出了一个非线性的决策边界）。



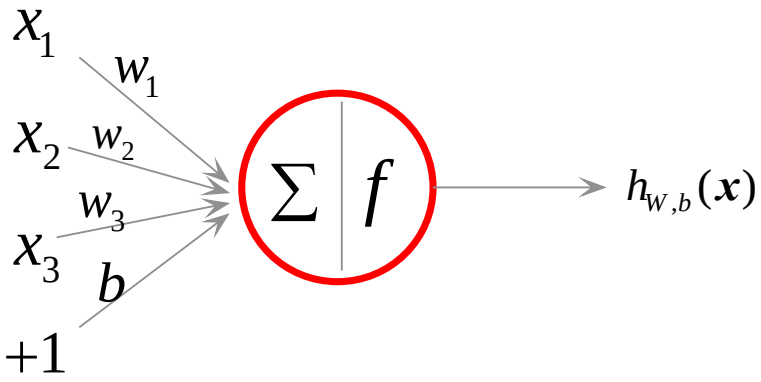
这个分类问题的样本特征个数大概有多少呢？假使我们采用的样本像素大小都是 50×50，并且将所有的像素点的灰度值视为特征，则会有 2500 个特征，如果再进一步将两两特征组合构成一个多项式模型，则会有约 $2500^2/2$ 个（接近 300 万个）特征。因此，只是简单地增加二次型、三次项等的逻辑回归算法，并不是一个解决复杂非线性问题的好办法，

因为当 n 值很大时，会产生非常多的特征项。所以，接下来，将要讲解神经网络算法，它被证实：在求解非线性分类问题中，即使在 n 值很大时，它是一种很好的算法。

2 神经网络模型的结构

以监督学习为例，假设我们有训练样本集 $(x^{(i)}, y^{(i)})_{i=1}^{m}$ ，那么神经网络算法能够提供一种复杂且非线性的假设模型 $h_{W,b}(x)$ ，它具有参数 W, b ，可以以此参数来拟合我们的数据。

为了描述神经网络，我们先从最简单的神经网络讲起，这个神经网络仅由一个“神经元”构成，以下即是这个“神经元”的图示：



这个“神经元”是一个以 x_1, x_2, x_3 及 $+1$ 为输入值的运算单元，其输出为

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

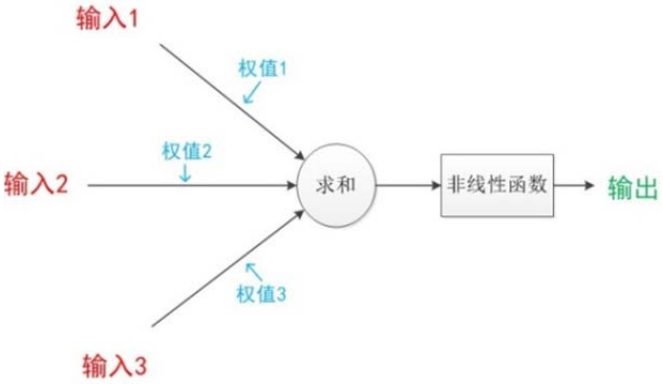
其中线上标注的 w_1, w_2, w_3 称为**权重**， b 称为**偏置**。函数 $f : \Re \mapsto \Re$ 被称为“**激活函数** activtion function”。我们选用非线性函数 sigmoid 函数作为激活函数

$$f(z) = \frac{1}{1 + \exp(-z)}.$$

如果把权重 W 和偏置 b 看作参数 θ ，神经元的输出可以写成

$$h_\theta(x) = \frac{1}{1+e^{-\theta^T x}}$$

和逻辑回归一样。逻辑回归模型也可以表示为如下的结构：



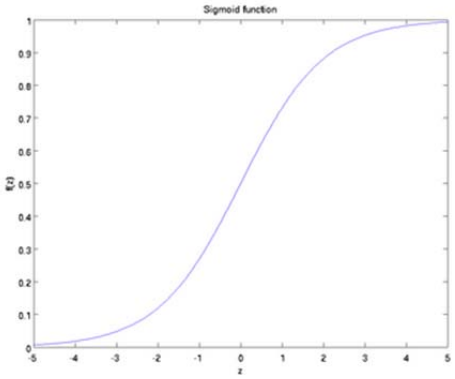
可以看出，单一神经元的输入 - 输出映射关系其实就是一个**逻辑回归**（logistic regression）。

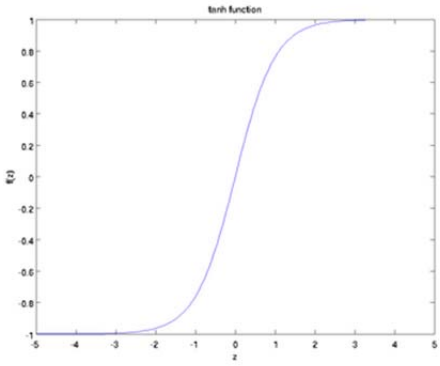
其实神经网络就像是逻辑回归，只不过把逻辑回归中的输入向量 $\boldsymbol{x}$ 变成了中间层的激活值 $\boldsymbol{a}$ 。从本质上讲，神经网络能够通过学习得出其自身的一系列特征。在普通的逻辑回归中，我们被限制为使用数据中的原始特征，我们虽然可以使用一些高次项来组合这些特征，但是我们仍然受到这些原始特征的限制。在神经网络中，原始特征只是输入层，在我们上面三层的神经网络例子中，第三层也就是输出层做出的预测利用的是第二层的特征，而非输入层中的原始特征，我们可以认为第二层中的特征是神经网络通过学习后自己得出的一系列用于预测输出变量的新特征。所以这些更高级的特征值远比仅仅将 x 的高次方的组合厉害，也能更好地预测新数据。这就是神经网络相比于逻辑回归和线性回归更具有优势。

除了采用 sigmoid 函数作为激活函数，也可以选择双曲正切函数（tanh）：

$$f(z) = \tanh(z) = \frac{e^z - e^{-z}}{e^z + e^{-z}},$$

以下分别是 sigmoid 及 tanh 的函数图像





$\tanh(z)$ 函数是 sigmoid 函数的一种变体，它的取值范围为 $[-1, 1]$，而不是 sigmoid 函数的 $[0, 1]$。后面讨论了更多的激活函数 。

需要说明的是，有一个等式我们以后会经常用到：如果选择

$$f(z) = 1/(1 + \exp(-z)),$$

也就是 sigmoid 函数，那么它的导数就是

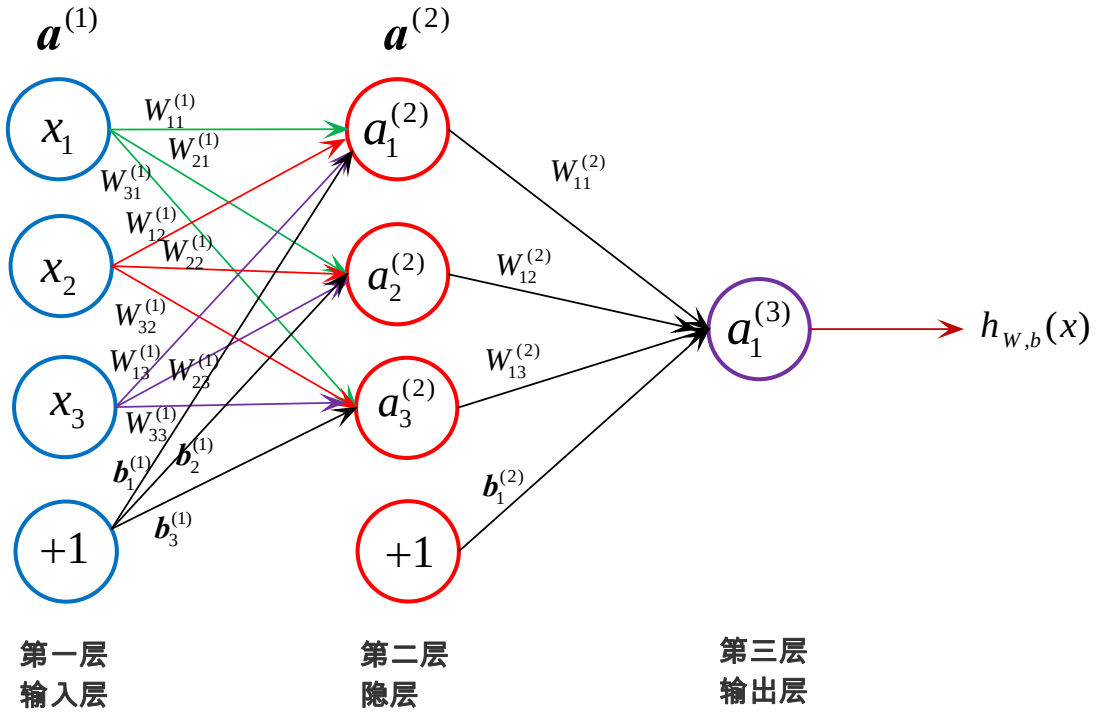
$$f'(z) = f(z)(1 - f(z))$$

如果选择 tanh 函数，那它的导数就是

$$f'(z) = 1 - (f(z))^2 ,$$

你可以根据 sigmoid（或 tanh）函数的定义自行推导这个等式。

所谓**神经网络就是将多个神经元联结在一起构成的网络。**这样，一个神经元的输出就可以是另一个神经元的输入。例如，下图就是一个简单的神经网络：



我们使用圆圈来表示神经网络的输入，标上“$+1$”的圆圈被称为**偏置单元**(bias unit)，也就是**截距项**。神经网络最左边的一层叫做**输入层**（Input Layer），最右的一层叫做**输出层**（Output Layer）（本例中，输出层只有一个节点）。中间所有节点组成的一层叫做**隐层**（Hidden Layers），在监督学习中，可以看到输入层和输入层，看不到中间层，所以才称中间层为隐层。同时可以看到，以上神经网络的例子中有 3 个**输入单元**（偏置单元不计在内），3 个**隐藏单元**及一个**输出单元**。

我们用 n_l 来表示网络的层数，本例中 $n_l = 3$，我们将第 l 层记为 L_l，于是 L_1 是输入层，输出层是 L_{n_l}。

本例神经网络有参数 $(W, b) = (W^{(1)}, b^{(1)}, W^{(2)}, b^{(2)})$，其中 $W_{ij}^{(l)}$ 是第 l 层第 j 单元与第 $l+1$ 层第 i 单元之间的联接参数（其实就是连接线上的权重，**注意标号顺序**），比如 $W_{23}^{(1)}$ 是第一层第 3 个单元与第二层第 2 个单元 的连接权重。把 $W_{ij}^{(l)}$ 组成矩阵 $W^{(l)}$，得

$$W^{(1)} = \begin{bmatrix} W_{11}^{(1)} & W_{12}^{(1)} & W_{13}^{(1)} \\ W_{21}^{(1)} & W_{22}^{(1)} & W_{23}^{(1)} \\ W_{31}^{(1)} & W_{32}^{(1)} & W_{33}^{(1)} \end{bmatrix}, \quad W^{(2)} = \begin{bmatrix} W_{11}^{(2)} & W_{12}^{(2)} & W_{13}^{(2)} \end{bmatrix}$$

$b_i^{(l)}$ 是第 $l+1$ 层第 i 单元的偏置项。向量化表示为：

$$\boldsymbol{b}^{(1)} = \begin{bmatrix} b_1^{(1)} \\ b_2^{(1)} \\ b_3^{(1)} \end{bmatrix}, \qquad \boldsymbol{b}^{(2)} = \begin{bmatrix} b_1^{(2)} \end{bmatrix}$$

同时，我们用 s_l 表示第 l 层的节点数（偏置单元不计在内）。

我们用 $a_i^{(l)}$ 表示第 l 层第 i 单元的**激活值**（输出值）。当 $l=1$ 时，$a_i^{(1)}=x_i$，也就是第 i 个输入值（输入值的第 i 个特征）。对于给定参数集合 W,b，我们的神经网络就可以按照函数 $h_{W,b}(x)$ 来计算输出结果。本例神经网络的计算步骤如下：

$$
\begin{aligned}
a_1^{(2)} &= f(W_{11}^{(1)}x_1 + W_{12}^{(1)}x_2 + W_{13}^{(1)}x_3 + b_1^{(1)}) \\
a_2^{(2)} &= f(W_{21}^{(1)}x_1 + W_{22}^{(1)}x_2 + W_{23}^{(1)}x_3 + b_2^{(1)}) \\
a_3^{(2)} &= f(W_{31}^{(1)}x_1 + W_{32}^{(1)}x_2 + W_{33}^{(1)}x_3 + b_3^{(1)}) \\
h_{W,b}(x) = a_1^{(3)} &= f(W_{11}^{(2)}a_1^{(2)} + W_{12}^{(2)}a_2^{(2)} + W_{13}^{(2)}a_3^{(2)} + b_1^{(2)})
\end{aligned}
$$

我们用 $z_i^{(l)}$ 表示第 l 层第 i 单元**输入加权和**（包括偏置单元），比如，

$$z_i^{(2)} = \sum_{j=1}^{n} W_{ij}^{(1)}x_j + b_i^{(1)}\ ,$$

则

$$a_i^{(l)} = f(z_i^{(l)})。$$

注意到

$z_i^{(l)}$ 是第 l 层第 i 单元的“**净输入**”，

$a_i^{(l)}$ 是第 l 层第 i 单元的输出，

$b_i^{(l)}$ 是第 $l+1$ 层第 i 单元的偏置项

这里我们将激活函数 $f(\cdot)$ 扩展为用向量（分量的形式）来表示，即

$$f([z_1,z_2,z_3]) = [f(z_1),f(z_2),f(z_3)]\ ,$$

并采用向量和矩阵，记

$$\boldsymbol{x} = \begin{bmatrix} x_1 \\ x_2 \\ x_3 \end{bmatrix}\ , \qquad \boldsymbol{z}^{(2)} = \begin{bmatrix} z_1^{(2)} \\ z_2^{(2)} \\ z_3^{(2)} \end{bmatrix}\ , \qquad \boldsymbol{a}^{(2)} = \begin{bmatrix} a_1^{(2)} \\ a_2^{(2)} \\ a_3^{(2)} \end{bmatrix}\ ,$$

那么，上面的等式可以更简洁地表示为：

$$
\begin{aligned}
z^{(2)} &= W^{(1)}x + b^{(1)} \\
a^{(2)} &= f(z^{(2)}) \\
z^{(3)} &= W^{(2)}a^{(2)} + b^{(2)} \\
h_{W,b}(x) &= a^{(3)} = f(z^{(3)})
\end{aligned}
$$

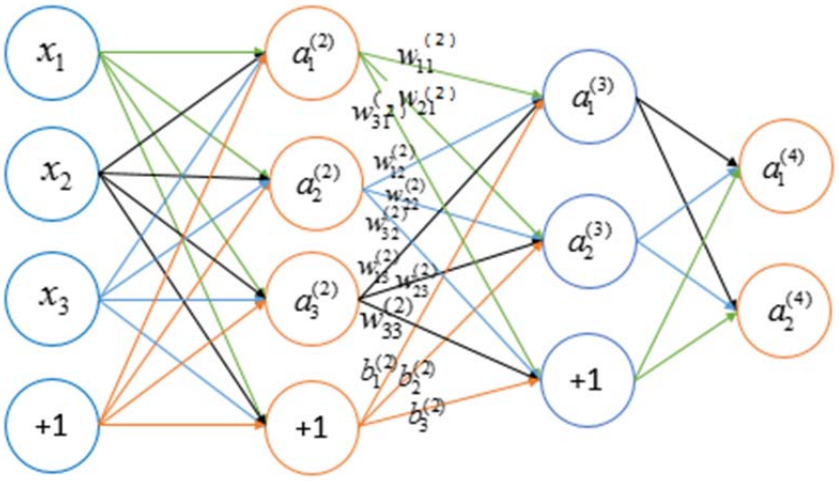
我们将上面的计算步骤叫作**正向传播**。回想一下，之前我们用 $a^{(1)}=x$ 表示输入层的激活值，那么给定第 l 层的激活值 $a^{(l)}$ 后，第 $l+1$ 层的激活值 $a^{(l+1)}$ 就可以按照下面步骤计算得到：

$$z^{(l+1)} = W^{(l)} a^{(l)} + b^{(l)}$$
$$a^{(l+1)} = f(z^{(l+1)})$$

将参数矩阵化，使用矩阵 - 向量运算方式，我们就可以利用线性代数的优势对神经网络进行快速求解。

目前为止，我们讨论了一种神经网络，我们也可以构建另一种**结构**的神经网络（这里结构指的是神经元之间的联接模式），也就是包含多个隐层的神经网络。最常见的一个例子是 n_l 层的神经网络，第 1 层是输入层，第 n_l 层是输出层，中间的每个层 l 与层 $l+1$ 紧密相联。这种模式下，要计算神经网络的输出结果，我们可以按照之前描述的等式，按部就班，进行正向传播，逐一计算第 L_2 层的所有激活值，然后是第 L_3 层的激活值，以此类推，直到第 L_{n_l} 层。这是一个**前馈神经网络**的例子，因为这种联接图没有闭环或回路。

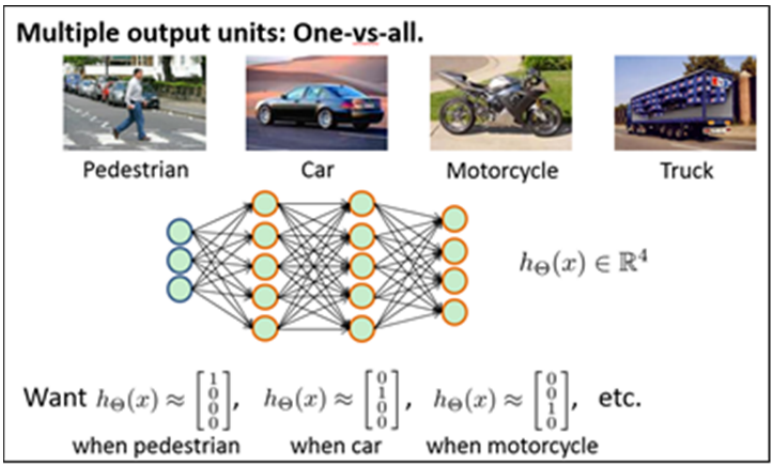
神经网络也可以有多个输出单元。比如，下面的神经网络有两层隐层： L_2 及 L_3 ，输出层 L_4 有两个输出单元。



第1层L1（输入层）　第2层L2（隐藏层）　第3层L3(隐藏层)　第4层L4,因为 n_l 表示神经网络的层数，所以这一层也即 L_{n_l} 层。(输出层)

要训练这样的神经网络，需要样本集 $(x^{(i)}, y^{(i)})$ ，其中 $y^{(i)} \in \Re^2$ 。如果你想预测的输出是多个的，那这种神经网络很适用。（比如，在医疗诊断应用中，患者的体征指标就可以作为向量的输入值，而不同的输出值 y_i 可以表示不同的疾病存在与否。）

当我们有不止两种分类时（也就是 y =1,2,3....），比如以下这种情况，该怎么办？如果我们要训练一个神经网络算法来识别路人、汽车、摩托车和货车，在输出层我们应该有 4 个值。例如，第一个值为 1 或 0 用于预测是否是行人，第二个值用于判断是否为汽车，第三个值用来判别是否为摩托车，第四个值用来判别是否为货车。输入向量 $\boldsymbol{x}$ 有三维，两个隐层， 输出层 4 个神经元分别用来表示 4 类。



3 神经网络的学习

神经网络的学习，也就是训练过程，指的是输入层神经元接收输入信息，传递给隐层神经元，最后传递到输出层神经元，由输出层输出结果的过程。在这个过程中，神经网络通过不断调整网络的权重和偏置，达到学习、训练的目的，当网络输出的误差减少到可以接受的程度，或者预先设定的学习次数后，学习就可以停止了。

如何调节网络的权重和偏置呢？我们可以使用**梯度下降法**等优化算法来训练神经网络。梯度下降法需要求出代价函数关于网络每个参数的偏导数。偏导数的计算需要用到**反向传播算法**。为了更深刻地理解反向传导算法，我们先介绍基于计算图模型的链式法则与反向求导方法。

3.1 基于计算图模型的链式法则与反向求导

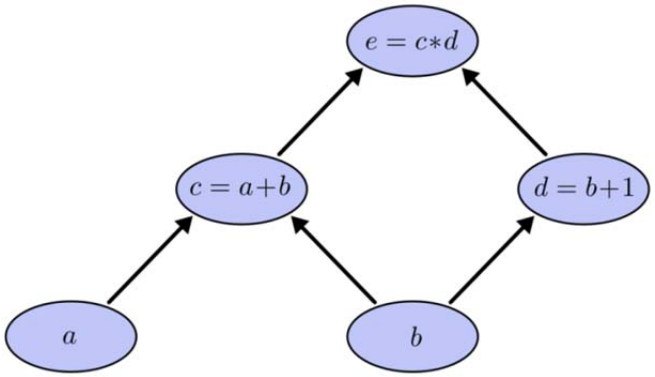
考虑函数

$$e=(a+b)*(b+1)$$

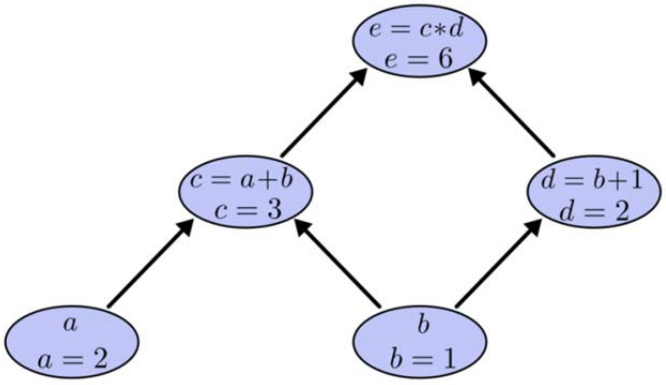
a, b 是输入变量，e 是输出变量。其中有 3 个运算，两个加法和一个乘法。引入两个中间变量 c 和 d ，有

$$c=a+b$$
$$d=b+1$$
$$e=c*d$$

把 3 个运算以及两个输入作为节点，得到如下的**计算图**（Computational Graphs）模型，其中箭头方向表示一个节点是另一个节点的输入：



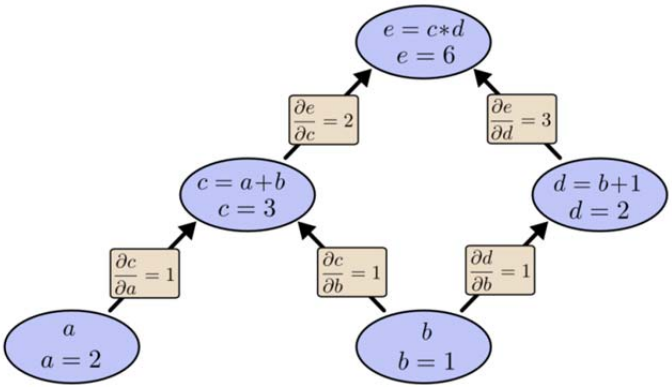
我们可以设置输入变量的值，通过以上的图计算输出变量的值，比如，设置 $a=2$ ， $b=1$。计算得到输出变量 $e=6$。



如果输入变量 a 发生改变，中间变量 c 也会发生，这样输出变量 e 就会发生改变。其变化率可以通过导数得出。我们把每个运算节点的输出对输入的导数标记在图的边上，如下图。其中计算偏导数用到了下面的和与积的导数法则。

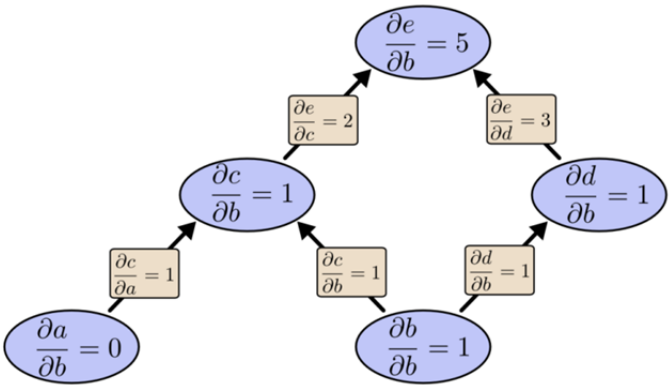
$$\frac{\partial}{\partial a}(a+b)=\frac{\partial a}{\partial a}+\frac{\partial b}{\partial a}=1$$

$$\frac{\partial}{\partial u}(uv)=u\frac{\partial v}{\partial u}+v\frac{\partial u}{\partial u}=v$$



图中输出节点 e 和输入节点 a 没有直接相连。输入 a 的变化是如何影响输出 e 的呢。如果 a 的变化量 Δa 为 h ，那么 c 的变化量 Δc 也是 h（因为 $c=a+b$ ， b 保持不变）。 c 的变化 $(\Delta c=h)$ 导致 e 的变化，变化量 Δe 是 $2h$。所以 e 关于 a 的变化率 $\frac{\Delta e}{\Delta a}=\frac{2h}{h}=2=\frac{\Delta c}{\Delta a}\frac{\Delta e}{\Delta c}$，令 $h\to 0$ ，$\frac{\partial e}{\partial a}=\frac{\partial c}{\partial a}\frac{\partial e}{\partial c}=1*2=2$。即求输出 e 对输入 a 的导数只需要把输入 a 到输出 e 的路径上的各边的导数相乘即可。

输入 b 对输出 e 的影响有两个路径， $b\to c\to e$ 以及 $b\to d\to e$ ，每条路径只是部分影响，计算输出 e 对输入 b 的导数需要把所有路径的影响相加。即 $\frac{\partial e}{\partial b}=\frac{\partial c}{\partial b}\frac{\partial e}{\partial c}+\frac{\partial d}{\partial b}\frac{\partial e}{\partial d}=1*2+1*3=5$。如下图所示：

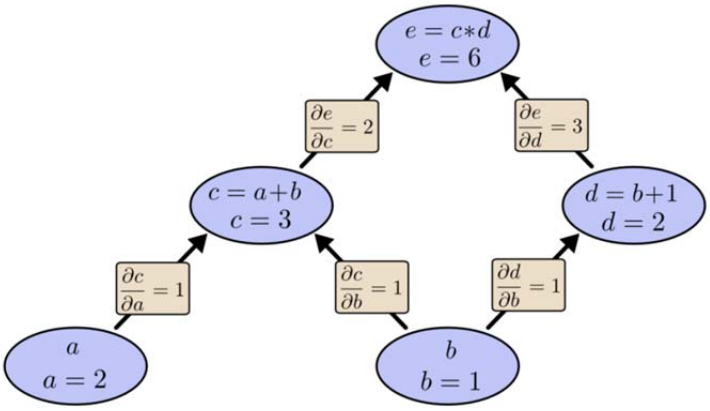


一般来说，对于上层节点 p 和下层节点 q ，要求得 $\frac{\partial p}{\partial q}$ ，需要找到从 q 节点到 p 节点的所有路径 ,并且对每条路径 ,求得该路径上的所有偏导数之乘积 ,然后将所有路径的 “乘积” 累加起来得到 $\frac{\partial p}{\partial q}$ 的值。

上面讲的就是多元函数求导的**链式法则**。这里通过图形解释了链式法则。

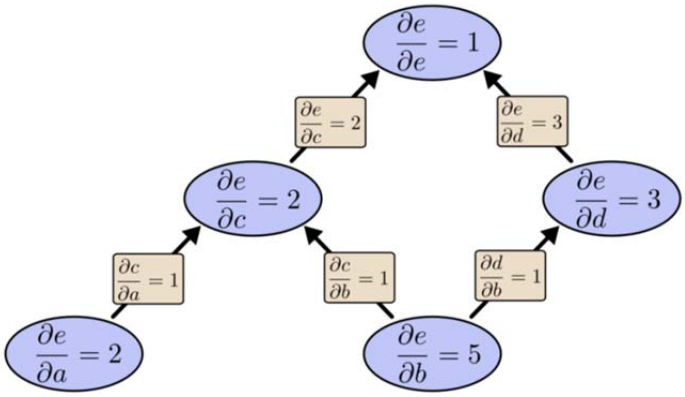
以上按照计算图模型的箭头方向求出输出变量对输入变量的梯度的过程称为**“正向求导”。**通过计算图模型的正向求导方法与微积分课程中的求导方法是相似的。但这不是求梯度的高效率的方法。因为这样做是十分冗余的 ,因为很多**路径被重复访问了。**例如在上图中，求 $\frac{\partial e}{\partial a}, \frac{\partial e}{\partial b}$ 时都通过了路径 $c \to e$ 。如果输入变量的数量巨大，这样的重复就会导致计算效率低下。下面介绍的**“反向求导”**方法，**同样是利用链式法则，但机智地避开了这种冗余，它对于每一个路径只访问一次就能求顶点对所有下层节点的偏导值。**

如果我们从 e 开始向下计算导数（自顶向下），计算 e 关于每个节点的导数。首先按箭头方向计算各个节点的值和各个边上的偏导值（这点和正向求导类似），如下图：



接着从顶点 e 开始（称为输出层），计算输出 e 对节点 e 的导数，即 $\frac{\partial e}{\partial e}=1$。节点 e 下面的两个节点 c 和 d 称为中间层。把节点 e 的导数值分别乘以 e 到某个节点路径上的偏导值，并将结果 “**堆放**” 在该节点中。具体地，节点 c 接受 e 发送的 1*2 并堆放起来，节点 d 接受 e 发送的 1*3 并堆放起来，至此中间层完毕。求出各节点总堆放量并继续向下一层（输入层）发送。节点 c 向 a 发送 2*1 并对堆放起来，节点 c 向 b 发送 2*1 并堆放起来，节点 d 向 b 发

送 3*1 并堆放起来，至此输入层完毕。节点 a 堆放起来的量为 2，即顶点 e 对 a 的偏导数为 2。节点 b 堆放起来的量为 2*1+3*1=5，即顶点 e 对 b 的偏导数为 5。

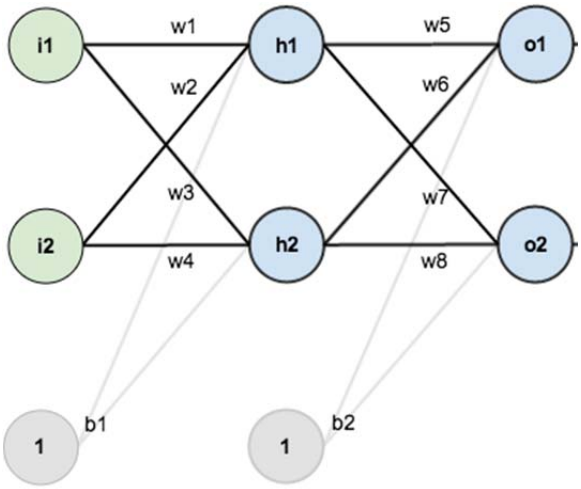


可以看出，反向求导方法求出了 e 对所有节点的导数，包括所有的输入节点。这样就一次性地求出了输出变量对**所有**输入变量的导数，即梯度。而正向求导方法每次只求出了输出变量对**一个**输入变量的导数。所以反向求导比正向求导计算效率更高，特别是输入变量个数非常大的情形。

3.2 信号的正向传播，误差的反向传播及权值更新---一个算例

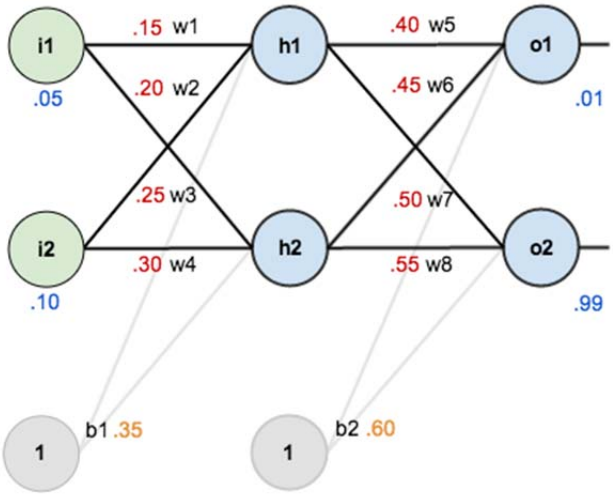
为了避免陷入符号恐惧而影响理解，下面采用简单的网络和简单符号描述网络，并以数据计算说明**输入信号的正向传播、误差的反向传播及权值更新是如何进行的**。

假设，有一个神经网络如下图所示：



第一层是输入层，包含两个神经元 i_1 ， i_2 ，和偏置 b_1 ；第二层是隐层，包含两个神经元 h_1, h_2 和偏置 b_2 ，第三层是输出 o_1, o_2 ，每条线上标的 w_i 是层与层之间连接的权重，激活函数默认为逻辑函数(sigmoid 函数)。

我们对该网络设置一些初始的权重、偏置以及输入和输出：，如下图：



其中，输入数据 i_1 =0.05，i_2 =0.10;

输出数据 o_1 =0.01, o_2 =0.99;

初始权重 w_1 =0.15, w_2 =0.20, w_3 =0.25, w_4 =0.30;

w_5 =0.40, w_6 =0.45, w_7 =0.50, w_8 =0.55

反向传播的目标是调节权重，使得神经网络能够学习到从任意的输入到输出的准确映射。我们将使用单个训练集 :给定输入(0.05 ,0.10),我们希望网络的输出为(0.01 ,0.99)。

前向传播

首先，让我们看看在给定上面的权重和偏置初值以及输入（0.05，0.10），神经网络的输出是什么，即需要求出网络输出 o_1 和 o_2 。 为此，我们将通过网络向前传递这些输入。我们先计算从全部网络的输入到隐层的每一个神经元，激活函数 g 采用 sigmoid 函数，对于从隐层到输出层，我们重复这一过程。

这里我们用符号 net 表示神经元的加权输入（和前面的符号 z 意义一样），用 out 表示神经元的输出（和前面的符号 a 意义一样），即 $out = f(net)$ ，

1. 输入层---->隐层：

计算神经元 h1 的输入加权和：

$$net_{h1} = w_1 * i_1 + w_2 * i_2 + b_1 * 1$$

$$net_{h1} = 0.15 * 0.05 + 0.2 * 0.1 + 0.35 * 1 = 0.3775$$

然后我们将其输入到激活函数中，得到神经元 h_1 的输出(激活函数为逻辑函数)：

$$out_{h1} = \frac{1}{1+e^{-net_{h1}}} = \frac{1}{1+e^{-0.3775}} = 0.593269992$$

对于神经元 h_2 通过上面相同的过程，我们可以得到

$$out_{h2} = 0.596884378$$

2. 隐层---->输出层：

对于输出层神经元，将隐层的输出加权求和后作为输入：

$$net_{o1} = w_5 * out_{h1} + w_6 * out_{h2} + b_2 * 1$$
$$net_{o1} = 0.4 * 0.593269992 + 0.45 * 0.596884378 + 0.6 * 1 = 1.105905967$$

然后我们将其输入到激活函数中，得到神经元o_1的输出：

$$out_{o1} = \frac{1}{1+e^{-net_{o1}}} = \frac{1}{1+e^{-1.105905967}} = 0.75136507$$

同样的，重复上面相同的过程，可以得到

$$out_{o2} = 0.772928465$$

这样前向传播的过程就结束了，我们得到输出值为（0.75136079，0.772928465），与实际值（0.01，0.99）相差还很远，现在我们对误差进行反向传播，更新权值，重新计算输出。

反向传播

反向传播的目标是：通过更新网络中的每一个权重，使得最终的输出接近于实际值，这样就得到整个网络的误差作为一个整体进行了最小化。

- **计算总误差**

现在对于输出的每一个神经元，使用平方误差函数求和来计算总的误差：

$$E_{total} = \sum \frac{1}{2}(target - output)^2$$

但是有两个输出，所以分别计算$o1$和$o2$的误差，总误差为两者之和：

$$E_{o1} = \frac{1}{2}(target - output)^2 = \frac{1}{2}(0.01 - 0.75136507)^2 = 0.274811083$$

重复上面过程，可以得到第二个神经元的输出$o2$为：

$$E_{o2} = 0.023560026$$

所以整个神经网络的误差求和为：

$$E_{total} = E_{o1} + E_{o2} = 0.274811083 + 0.023560026 = 0.298371109$$

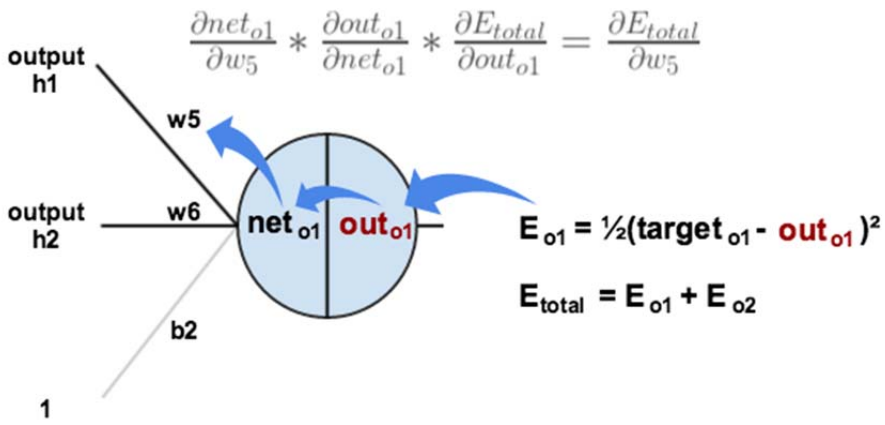
- **隐层---->输出层的权值更新：**

先来考察w_5，我们想知道对于w_5的改变可以多大程度上影响总误差，也就是$\frac{\partial E_{total}}{\partial w_5}$

通过使用链式法则，可以得到：

$$\frac{\partial E_{total}}{\partial w_5} = \frac{\partial E_{total}}{\partial out_{o1}} * \frac{\partial out_{o1}}{\partial net_{o1}} * \frac{\partial net_{o1}}{\partial w_5}$$

为了更直观的表述上面链式法则的过程，对其进行可视化：



我们对上面使用链式法则得到的每一项分别进行计算。

1. $\frac{\partial E_{total}}{\partial w_5} = \frac{\partial E_{total}}{\partial out_{o1}} * \frac{\partial out_{o1}}{\partial net_{o1}} * \frac{\partial net_{o1}}{\partial w_5}$

   $\rightarrow \frac{\partial E_{total}}{\partial out_{o1}}$ ：输出的改变对总误差的影响

   $$E_{total} = E_{o1} + E_{o2} = \frac{1}{2}(target_{o1} - output_{o1})^2 + \frac{1}{2}(target_{o2} - output_{o2})^2$$

   $$\frac{\partial E_{total}}{\partial out_{o1}} = \frac{\partial E_{o1}}{\partial out_{o1}} + \frac{\partial E_{o2}}{\partial out_{o1}}$$

   $$= -(target_{o1} - output_{o1}) = -(0.01 - 0.75136507) = 0.74136507$$

   其中

   $$\frac{\partial E_{o1}}{\partial out_{o1}} = 0.74136507$$ ，**这个值要保留，待会还要用**

   $\frac{\partial E_{o2}}{\partial out_{o1}} = 0$ ，即 $\frac{\partial E_{total}}{\partial out_{o1}} = \frac{\partial E_{o1}}{\partial out_{o1}}$

2. $\frac{\partial E_{total}}{\partial w_5} = \frac{\partial E_{total}}{\partial out_{o1}} * \frac{\partial out_{o1}}{\partial net_{o1}} * \frac{\partial net_{o1}}{\partial w_5}$

   $\rightarrow \frac{\partial out_{o1}}{\partial net_{o1}}$ ：逻辑函数的偏导数为输出乘以 1 与输出的差

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

   $$\frac{\partial out_{o1}}{\partial net_{o1}} = out_{o1}(1 - out_{o1}) = 0.75136507(1 - 0.75136507) = 0.186815602$$

   即

   $$\frac{\partial out_{o1}}{\partial net_{o1}} = 0.186815602$$ ， **这个值要保留，待会还要用**

3. $\frac{\partial E_{total}}{\partial w_5} = \frac{\partial E_{total}}{\partial out_{o1}} * \frac{\partial out_{o1}}{\partial net_{o1}} * \frac{\partial net_{o1}}{\partial w_5}$

   $\rightarrow \frac{\partial net_{o1}}{\partial w_5}$ ： $w5$的改变对$o1$的输入 net 的影响

$$net_{o1} = w_5 * out_{h1} + w_6 * out_{h2} + b_2 * 1$$

$$\frac{\partial net_{o1}}{\partial w_5} = out_{h1} = 0.593269992$$

最后，三者相乘，

$$\frac{\partial E_{total}}{\partial w_5} = \frac{\partial E_{total}}{\partial out_{o1}} * \frac{\partial out_{o1}}{\partial net_{o1}} * \frac{\partial net_{o1}}{\partial w_5}$$

$$= 0.74136507 * 0.186815602 * 0.593269992 = 0.082167041$$

这样我们就计算出总误差 E_{total} 对 w_5 的偏导值。

回过头来再看看上面的公式，我们发现：

$$\frac{\partial E_{total}}{\partial w_5} = \frac{\partial E_{total}}{\partial out_{o1}} * \frac{\partial out_{o1}}{\partial net_{o1}} * \frac{\partial net_{o1}}{\partial w_5}$$

$$= -(target_{o1} - out_{o1}) * out_{o1}(1 - out_{o1}) * out_{h1}$$

令

$$\delta_{o1} = \frac{\partial E_{o1}}{\partial net_{o1}} = \frac{\partial E_{o1}}{\partial out_{o1}} * \frac{\partial out_{o1}}{\partial net_{o1}}$$

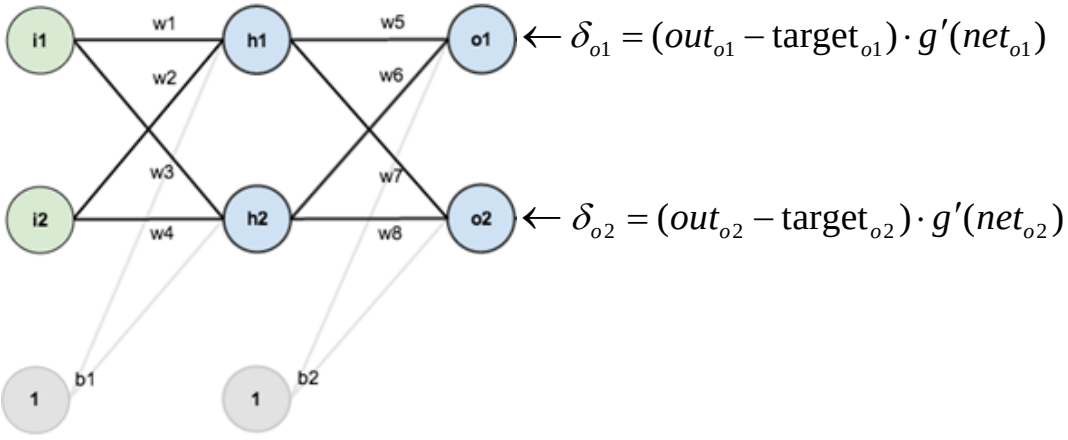
$$= -(target_{o1} - out_{o1}) * g'(net_{o1})$$

$$= -(target_{o1} - out_{o1}) * out_{o1}(1 - out_{o1})$$

注意到 δ_{o1} 的值只与神经元 $o1$ 的输出值 out_{o1} 与目标值 $target_{o1}$ 有关。同理可以定义

$$\delta_{o2} = \frac{\partial E_{o2}}{\partial net_{o2}} = (out_{o2} - target_{o2}) * g'(net_{o2})$$

δ 有时称为“**残差**”。残差一般指实预测值与真实值之间的差。这里残差 δ 的定义稍稍有点不同，**输出层的残差等于预测误差（输出值与目标值之差）与激活函数的导数的乘积**。



利用记号 δ，w_5 的改变对总误差的影响，也就是 $\frac{\partial E_{total}}{\partial w_5}$ 可以简单表示为（注意到 $\frac{\partial E_{total}}{\partial out_{o1}} = \frac{\partial E_{o1}}{\partial out_{o1}}$）：

$$\frac{\partial E_{total}}{\partial w_5} = \delta_{o1} out_{h1}$$

为了减小误差，我们将 w_5 原来的值减去总误差对目前权重的偏导(通常会乘上一个学习率 η ，这里我们将其设置为 0.5)，得到更新后的权重 w_5'，即采用**梯度下降法**更新权重：

$$w_5' = w_5 - \eta * \frac{\partial E_{total}}{\partial w_5}$$

重复上面的过程，我们可以得到更新后的 w_6' ， w_7' ， w_8' ：

$$w_6' = 0.408666186$$
$$w_7' = 0.511301270$$
$$w_8' = 0.561370121$$

- **隐层---->隐层的权值更新：**

我们继续推进反向传播来计算 w_1、 w_2、 w_3、 w_4 更新后的权重：

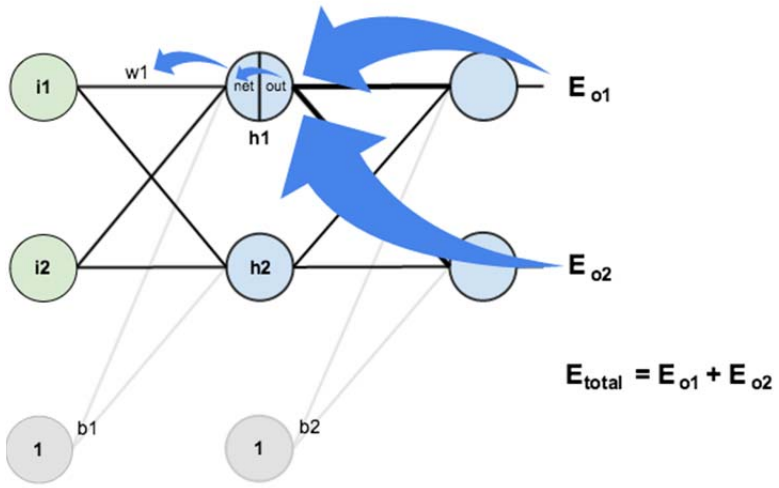
同样使用链式法则，我们可以得到：

$$\frac{\partial E_{total}}{\partial w_1} = \frac{\partial E_{total}}{\partial out_{h_1}} * \frac{\partial out_{h_1}}{\partial net_{h_1}} * \frac{\partial net_{h_1}}{\partial w_1}$$

可视化上面的链式法则如下图：

$$\frac{\partial E_{total}}{\partial w_1} = \frac{\partial E_{total}}{\partial out_{h1}} * \frac{\partial out_{h1}}{\partial net_{h1}} * \frac{\partial net_{h1}}{\partial w_1}$$

$$\frac{\partial E_{total}}{\partial out_{h1}} = \frac{\partial E_{o1}}{\partial out_{h1}} + \frac{\partial E_{o2}}{\partial out_{h1}}$$



1. $\frac{\partial E_{total}}{\partial w_1} = \frac{\partial E_{total}}{\partial out_{h_1}} * \frac{\partial out_{h_1}}{\partial net_{h1}} * \frac{\partial net_{h1}}{\partial w_1}$

$\rightarrow \frac{\partial E_{total}}{\partial out_{h1}}$ ：

对于这一层(隐层)的更新我们采用上面输出层相似的处理方式，不过会稍有不同，这种不同主要是因为每一个隐层的神经元的输出对于最终的输出都是有贡献的。我们知道out_{h1}既影响out_{o1}也影响out_{o2}，因此$\frac{\partial E_{total}}{\partial out_{h1}}$需要同时考虑到这两个输出神经元影响：

$$\frac{\partial E_{total}}{\partial out_{h1}} = \frac{\partial E_{o1}}{\partial out_{h_1}} + \frac{\partial E_{o2}}{\partial out_{h1}}$$

又由于：

$$\frac{\partial E_{o1}}{\partial out_{h1}} = \frac{\partial E_{o1}}{\partial net_{o1}} * \frac{\partial net_{o1}}{\partial out_{h1}}$$

我们可以用前面计算的值来计算$\frac{\partial E_{o1}}{\partial net_{o1}}$：

$$\frac{\partial E_{o1}}{\partial net_{o1}} = \frac{\partial E_{o1}}{\partial out_{o1}} * \frac{\partial out_{o1}}{\partial net_{o1}} = 0.74136507 * 0.186815602 = 0.138498562$$

又因为

$$net_{o1} = w_5 * out_{h1} + w_6 * out_{h2} + b_2 * 1$$

$$\frac{\partial net_{o1}}{\partial out_{h1}} = w_5 = 0.40$$

将上面每步分开算的结果合起来得：

$$\frac{\partial E_{o1}}{\partial out_{h1}} = \frac{\partial E_{o1}}{\partial net_{o1}} * \frac{\partial net_{o1}}{\partial out_{h1}} = 0.138498562 * 0.40 = 0.055399425$$

与上面的步骤一样，我们可以得到：

$$\frac{\partial E_{o2}}{\partial out_{h1}} = -0.019049119$$

因此：

$$\frac{\partial E_{total}}{\partial out_{h1}} = \frac{\partial E_{o1}}{\partial out_{h1}} + \frac{\partial E_{o2}}{\partial out_{h1}} = 0.055399425 + (-0.019049119) = 0.036350306$$

2. $\frac{\partial E_{total}}{\partial w_1} = \frac{\partial E_{total}}{\partial out_{h_1}} * \frac{\partial out_{h_1}}{\partial net_{h1}} * \frac{\partial net_{h1}}{\partial w_1}$

$\rightarrow \frac{\partial out_{h_1}}{\partial net_{h1}}$：

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

$$\frac{\partial out_{h1}}{\partial net_{h1}} = out_{h1}(1 - out_{h1}) = 0.59326999(1 - 0.59326999) = 0.241300709$$

3. $\frac{\partial E_{total}}{\partial w_1}=\frac{\partial E_{total}}{\partial out_{h_1}}*\frac{\partial out_{h_1}}{\partial net_{h1}}*\frac{\partial net_{h1}}{\partial w_1}$

$\rightarrow \frac{\partial net_{h1}}{\partial w_1}$ ：

$$net_{h1}=w_1*i_1+w_2*i_2+b_1*1$$

$$\frac{\partial net_{h1}}{\partial w_1}=i_1=0.05$$

将上面计算的三个部分相乘：

$$\frac{\partial E_{total}}{\partial w_1}=\frac{\partial E_{total}}{\partial out_{h1}}*\frac{\partial out_{h1}}{\partial net_{h1}}*\frac{\partial net_{h1}}{\partial w_1}$$

$$=0.036350306*0.241300709*0.05=0.000438568$$

下面我们以残差后向传播的观点来看上面的计算过程。

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

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

注意到 $\frac{\partial E_{o1}}{\partial out_{o1}}*\frac{\partial out_{o1}}{\partial net_{o1}}=\delta_{o1}$ ，$\frac{\partial E_{o2}}{\partial out_{o2}}*\frac{\partial out_{o2}}{\partial net_{o2}}=\delta_{o2}$ ，$\frac{\partial net_{o1}}{\partial out_{h1}}=w_5$ ，$\frac{\partial net_{o2}}{\partial out_{h1}}=w_7$

所以

$$\frac{\partial E_{total}}{\partial w_1}=(\delta_{o1}*w_5+\delta_{o2}*w_7)*out_{h1}(1-out_{h1})*i_1$$

$$=(\delta_{o1}*w_5+\delta_{o2}*w_7)*g'(net_{h1})*i_1$$

令隐层神经元 h_1 的残差 δ_{h1} 为：

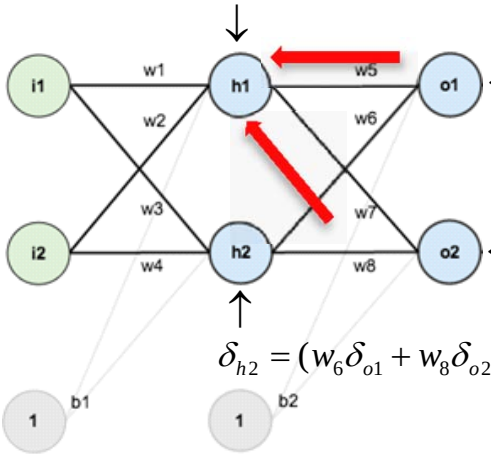
$$\delta_{h1}=(\delta_{o1}*w_5+\delta_{o2}*w_7)*g'(net_{h1})$$

则 $$\frac{\partial E_{total}}{\partial w_1}=\delta_{h1}*i_1$$

类似有 $$\delta_{h2}=(\delta_{o1}*w_6+\delta_{o2}*w_8)*g'(net_{h2})$$

注意，**输入层神经元没有残差**。残差后向传播的过程如下图：

$$\delta_{h1}=(w_5\delta_{o1}+w_7\delta_{o1})\cdot g'(net_{h1})$$



$$\leftarrow \delta_{o1}=(out_{o1}-\text{target}_{o1})\cdot g'(net_{o1})$$

$$\leftarrow \delta_{o2}=(out_{o2}-\text{target}_{o2})\cdot g'(net_{o2})$$

$$\delta_{h2}=(w_6\delta_{o1}+w_8\delta_{o2})\cdot g'(net_{h2})$$

现在我们可以更新 w_1：

$$w_1' = w_1 - \eta * \frac{\partial E_{total}}{\partial w_1} = 0.15 - 0.5 * 0.000438568 = 0.149780716$$

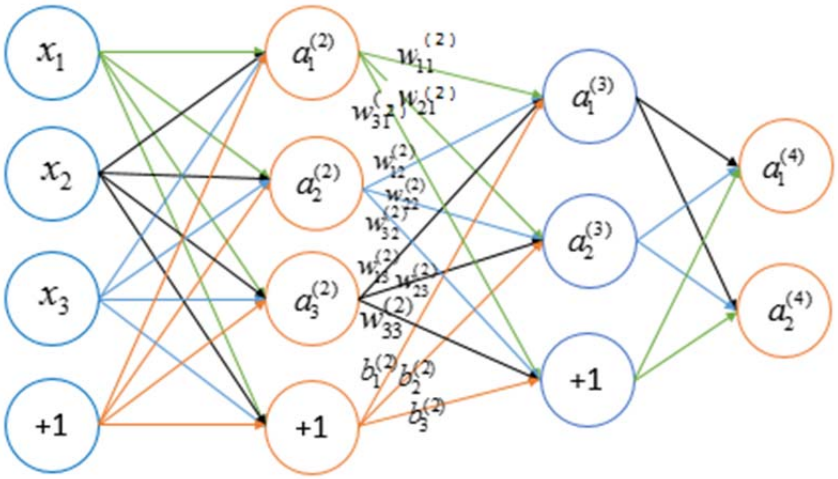
根据上面相同的计算过程，我们可以得到 w_2、 w_3、 w_4 的更新：

$$w_2' = 0.19956143$$
$$w_3' = 0.24975114$$
$$w_4' = 0.29950229$$

现在，我们已经更新了所有的权重，偏置的更新可以类似更新。在最初，在我们的输入为 0.05 和 0.1 的时候，网络的误差为 0.298371109， 经过第一次方向传播后，网络的误差降低到了 0.291027924。虽然看起来下降得不是很多，但是在重复这个过程 10000 次以后，网络的误差就下降到了 0.000035085。这个时候，当我们把 0.05 和 0.1 再输入进去，两个神经元的输出为 0.015912196(vs 0.01)和 0.984065734(vs 0.99)。

***3.3 反向传播的数学原理**

我们考虑一般的神经网络模型。网络共有 n_l 层，第一层是输入层，第 n_l 层表示输出层，其余的为隐层。设第 l 层的神经元个数为 s_l，第 $l+1$ 层的神经元个数为 s_{l+1}，输出层的神经元个数为 s_{n_l}。



假设我们有一个样本集 $\{(x^{(1)}, y^{(1)}), \ldots, (x^{(m)}, y^{(m)})\}$，它包含 m 个样例。我们可以用**批量梯度下降法**来训练神经网络。具体来讲，对于**单个样例** (x, y)，其**代价函数**为：

$$J(W, b; x, y) = \frac{1}{2} \left\| h_{W,b}(x) - y \right\|^2 . \qquad (1.3)$$

这是一个最小二乘代价函数。给定一个包含 m 个样例的数据集，我们可以定义整体代价函数为：

$$\begin{aligned} J(W,b) &= \left[\frac{1}{m}\sum_{i=1}^{m} J(W,b;x^{(i)},y^{(i)})\right] + \frac{\lambda}{2}\sum_{l=1}^{n_l-1}\sum_{i=1}^{s_l}\sum_{j=1}^{s_{l+1}}\left(W_{ji}^{(l)}\right)^2 \\ &= \left[\frac{1}{m}\sum_{i=1}^{m}\left(\frac{1}{2}\left\|h_{W,b}(x^{(i)})-y^{(i)}\right\|^2\right)\right] + \frac{\lambda}{2}\sum_{l=1}^{n_l-1}\sum_{i=1}^{s_l}\sum_{j=1}^{s_{l+1}}\left(W_{ji}^{(l)}\right)^2 \end{aligned} \quad (1.4)$$

以上公式中的第一项 $J(W,b)$ 是一个均方差项。第二项是一个**正则化项**（也叫**权重衰减项**），其目的是减小权重的幅度，防止过度拟合。

注：通常权重衰减的计算并不使用偏置项 $b_i^{(l)}$，比如我们在 $J(W,b)$ 的定义中就没有使用。一般来说，将偏置项包含在权重衰减项中只会对最终的神经网络产生很小的影响。

权重衰减参数 λ 用于控制公式中两项的相对重要性。在此重申一下这两个复杂函数的含义：$J(W,b;x,y)$ 是针对**单个样例**计算得到的方差代价函数；$J(W,b)$ 是**整体样本**代价函数，它包含权重衰减项。

以上的代价函数经常被用于**分类和回归**问题。在分类问题中，我们用 $y=0$ 或 1，来代表两种类型的标签（回想一下，这是因为 sigmoid 激活函数的值域为 $[0,1]$；如果我们使用双曲正切型激活函数，那么应该选用 -1 和 $+1$ 作为标签）。对于回归问题，我们首先要变换输出值域（也就是 y），以保证其范围为 $[0,1]$（同样地，如果我们使用双曲正切型激活函数，要使输出值域为 $[-1,1]$）。

我们的**目标**是针对参数 W 和 b 来求其函数 $J(W,b)$ 的最小值。为了求解神经网络，我们需要将每一个参数 $W_{ij}^{(l)}$ 和 $b_i^{(l)}$ 初始化为一个很小的、接近零的随机值（比如说，使用正态分布 $Normal(0,\epsilon^2)$ 生成的随机值，其中 ϵ 设置为 0.01），之后对目标函数使用诸如批量梯度下降法的最优化算法。因为 $J(W,b)$ 是一个非凸函数，梯度下降法很可能会收敛到局部最优解，但是在实际应用中，梯度下降法通常能得到令人满意的结果。最后，需要再次强调的是，要将参数进行随机初始化，而不是全部置为 0。如果所有参数都用相同的值作为初始值，那么所有隐层单元最终会得到与输入值有关的、相同的函数（也就是说，对于所有 i，$W_{ij}^{(1)}$ 都会取相同的值，那么对于任何输入 x 都会有：$a_1^{(2)}=a_2^{(2)}=a_3^{(2)}=\ldots$）。随机初始化的目的是使**对称失效**。

批量梯度下降法中每一次迭代都按照如下公式对参数 W 和 b 进行更新：

（批量指的是每一时候参数的更新使用到了所有的训练样本。）

$$\begin{aligned} W_{ij}^{(l)} &= W_{ij}^{(l)} - \alpha\frac{\partial}{\partial W_{ij}^{(l)}}J(W,b) \\ b_i^{(l)} &= b_i^{(l)} - \alpha\frac{\partial}{\partial b_i^{(l)}}J(W,b) \end{aligned} \quad (1.5)$$

其中 α 是学习速率。其中关键步骤是**计算偏导数**。我们现在来讲一下**反向传播**算法，它是计算偏导数的一种有效方法。

我们首先来讲一下如何使用反向传播算法来计算

$\frac{\partial}{\partial W_{ij}^{(l)}} J(W,b;x,y)$ 和 $\frac{\partial}{\partial b_i^{(l)}} J(W,b;x,y)$，

这两项是单个样例 (x,y) 的代价函数 $J(W,b;x,y)$ 的偏导数。一旦我们求出该偏导数，就可以推导出整体代价函数 $J(W,b)$ 的偏导数：

$$\begin{aligned}\frac{\partial}{\partial W_{ij}^{(l)}} J(W,b) &= \left[\frac{1}{m}\sum_{i=1}^{m}\frac{\partial}{\partial W_{ij}^{(l)}} J(W,b;x^{(i)},y^{(i)})\right] + \lambda W_{ij}^{(l)} \\ \frac{\partial}{\partial b_i^{(l)}} J(W,b) &= \frac{1}{m}\sum_{i=1}^{m}\frac{\partial}{\partial b_i^{(l)}} J(W,b;x^{(i)},y^{(i)})\end{aligned} \tag{1.6}$$

矩阵形式：

$$\begin{aligned}\frac{\partial J(W,b)}{\partial W^{(l)}} &= \frac{1}{m}\sum_{i=1}^{m}\left(\frac{\partial J(W,b;x^{(i)},y^{(i)})}{\partial W^{(l)}}\right) + \lambda W^{(l)} \\ \frac{\partial J(W,b)}{\partial b^{(l)}} &= \frac{1}{m}\sum_{i=1}^{m}\left(\frac{\partial J(W,b;x^{(i)},y^{(i)})}{\partial b^{(l)}}\right)\end{aligned}$$

以上两行公式稍有不同，第一行比第二行多出一项，是因为权重衰减是作用于 W 而不是 b。

反向传播算法的思路如下：给定一个样例 (x,y)，我们首先进行“**正向传导**”运算，计算出网络中所有的激活值，包括 $h_{W,b}(x)$ 的输出值。之后，针对第 l 层的每一个节点 i，我们计算出其**残差** $\delta_i^{(l)}$，该残差表明了该节点对最终输出值的残差产生了多少影响。对于最终的输出节点，我们可以直接算出网络产生的激活值与真实值之间的差距，我们将这个差距定义为 $\delta_i^{(n_l)}$（第 n_l 层表示输出层）。对于隐藏单元我们如何处理呢？我们将基于节点（第 $l+1$ 层节点）残差的加权平均值计算 $\delta_i^{(l)}$，这些节点以 $a_i^{(l)}$ 作为输入。下面将给出反向传导算法的细节：

1. 进行正向传导计算，利用正向传导公式，得到 $L_2, L_3, \ldots$ 直到输出层 L_{n_l} 的激活值。
2. 对于第 n_l 层（**输出层**）的每个输出单元 i，我们根据以下公式计算**残差**：

$$\delta_i^{(n_l)} = \frac{\partial}{\partial z_i^{(n_l)}} \frac{1}{2}\left\|y - h_{W,b}(x)\right\|^2 = -(y_i - a_i^{(n_l)}) \cdot f'(z_i^{(n_l)})$$

其推导过程为

$$\begin{aligned}\delta_i^{(n_l)} &= \frac{\partial}{\partial z_i^{n_l}} J(W,b;x,y) = \frac{\partial}{\partial z_i^{n_l}} \frac{1}{2} \left\| y - h_{W,b}(x) \right\|^2 \\ &= \frac{\partial}{\partial z_i^{n_l}} \frac{1}{2} \sum_{j=1}^{S_{n_l}} (y_j - a_j^{(n_l)})^2 = \frac{\partial}{\partial z_i^{n_l}} \frac{1}{2} \sum_{j=1}^{S_{n_l}} (y_j - f(z_j^{(n_l)}))^2 \\ &= -(y_i - f(z_j^{(n_l)})) \cdot f'(z_i^{(n_l)}) = -(y_i - a_i^{(n_l)}) \cdot f'(z_i^{(n_l)})\end{aligned}$$

3. 对 $l = n_l - 1, n_l - 2, n_l - 3, \ldots, 2$ 的各个层，第 l 层的第 i 个节点的残差计算方法如下：

$$\delta_i^{(l)} = \left(\sum_{j=1}^{s_{l+1}} W_{ji}^{(l)} \delta_j^{(l+1)} \right) f'(z_i^{(l)})$$

其推导过程为：

$$\begin{aligned}\delta_i^{(n_l-1)} &= \frac{\partial}{\partial z_i^{n_l-1}} J(W,b;x,y) = \frac{\partial}{\partial z_i^{n_l-1}} \frac{1}{2} \left\| y - h_{W,b}(x) \right\|^2 = \frac{\partial}{\partial z_i^{n_l-1}} \frac{1}{2} \sum_{j=1}^{S_{n_l}} (y_j - a_j^{(n_l)})^2 \\ &= \frac{1}{2} \sum_{j=1}^{S_{n_l}} \frac{\partial}{\partial z_i^{n_l-1}} (y_i - a_j^{(n_l)})^2 = \frac{1}{2} \sum_{j=1}^{S_{n_l}} \frac{\partial}{\partial z_i^{n_l-1}} (y_i - f(z_j^{(n_l)})^2 \\ &= \sum_{j=1}^{S_{n_l}} (y_j - f(z_j^{(n_l)}) \cdot \frac{\partial}{\partial z_i^{(n_l-1)}} f(z_j^{(n_l)}) = \sum_{j=1}^{S_{n_l}} (y_j - f(z_j^{(n_l)}) \cdot f'(z_j^{(n_l)})) \cdot \frac{\partial z_j^{(n_l)}}{\partial z_i^{(n_l-1)}} \\ &= \sum_{j=1}^{S_{n_l}} \delta_j^{(n_l)} \cdot \frac{\partial z_j^{(n_l)}}{\partial z_i^{n_l-1}} = \sum_{j=1}^{S_{n_l}} \left(\delta_j^{(n_l)} \cdot \frac{\partial}{\partial z_i^{n_l-1}} \sum_{k=1}^{S_{n_l-1}} f(z_k^{n_l-1}) \cdot W_{jk}^{n_l-1} \right) \\ &= \sum_{j=1}^{S_{n_l}} \delta_j^{(n_l)} \cdot W_{ji}^{n_l-1} \cdot f'(z_i^{n_l-1}) = \left(\sum_{j=1}^{S_{n_l}} W_{ji}^{n_l-1} \delta_j^{(n_l)} \right) f'(z_i^{n_l-1})\end{aligned}$$

将上式中的$n_l - 1$与n_l 的关系替换为l与$l+1$的关系，就可以得到：

$$\delta_i^{(l)} = \left(\sum_{j=1}^{s_{l+1}} W_{ji}^{(l)} \delta_j^{(l+1)} \right) f'(z_i^{(l)})$$

4. 计算偏导数，方法如下：

$$\begin{aligned}\frac{\partial}{\partial W_{ij}^{(l)}} J(W,b;x,y) &= a_j^{(l)} \delta_i^{(l+1)} \\ \frac{\partial}{\partial b_i^{(l)}} J(W,b;x,y) &= \delta_i^{(l+1)}.\end{aligned} \qquad (1.9)$$

最后，我们用矩阵-向量表示法重写以上算法。我们使用“$\bullet$” 表示向量乘积运算符（在 MATLAB 里用“.*”表示，也称作阿达马乘积）。若 $a = b \bullet c$，则 $a_i = b_i c_i$。在上一个教程中我们扩展了 $f(\cdot)$ 的定义，使其包含向量运算，这里我们也对偏导数 $f'(\cdot)$ 也做了同样的处理，于是有

$$f'([z_1, z_2, z_3]) = [f'(z_1), f'(z_2), f'(z_3)]$$

那么，反向传播算法可表示为以下几个步骤：

第一步：进行前馈传导计算，利用正向传导公式，得到 $L_2, L_3, \ldots$ 直到输出层 L_{n_l} 的激活值。

第二步：对输出层（第 n_l 层），计算：

$$\delta^{(n_l)} = -(y - a^{(n_l)}) \bullet f'(z^{(n_l)}) \qquad (1.10)$$

第三步：对于 $l = n_l - 1, n_l - 2, n_l - 3, \ldots, 2$ 的各层，计算：

$$\delta^{(l)} = \left((W^{(l)})^T \delta^{(l+1)}\right) \bullet f'(z^{(l)}) \qquad (1.11)$$

第四步：计算最终需要的偏导数值：

$$\begin{aligned} \nabla_{W^{(l)}} J(W, b; x, y) &= \delta^{(l+1)} (a^{(l)})^T, \\ \nabla_{b^{(l)}} J(W, b; x, y) &= \delta^{(l+1)}. \end{aligned} \qquad (1.12)$$

实现中应注意：在以上的第 2 步和第 3 步中，我们需要为每一个 i 值计算其 $f'(z_i^{(l)})$。假设 $f(z)$ 是 sigmoid 函数，并且我们已经在正向传导运算中得到了 $a_i^{(l)}$。那么，使用我们早先推导出的 $f'(z)$ 表达式，就可以计算得到

$$f'(z_i^{(l)}) = a_i^{(l)}(1 - a_i^{(l)}). \qquad (1.13)$$

最后，我们将对梯度下降算法做个全面总结。在下面的伪代码中，$\Delta W^{(l)}$ 是一个与矩阵 $W^{(l)}$ 维度相同的矩阵，$\Delta b^{(l)}$ 是一个与 $b^{(l)}$ 维度相同的向量。注意这里“$\Delta W^{(l)}$”是一个矩阵，而不是“Δ 与 $W^{(l)}$ 相乘”。下面，我们实现批量梯度下降法中的一次迭代：

1. 对于所有 l，令 $\Delta W^{(l)} := 0$，$\Delta b^{(l)} := 0$（设置为全零矩阵或全零向量）

2. 对于 $i = 1$ 到 m，

a. 使用反向传播算法计算 $\nabla_{W^{(l)}} J(W, b; x, y)$ 和 $\nabla_{b^{(l)}} J(W, b; x, y)$。

b. 计算 $\Delta W^{(l)} := \Delta W^{(l)} + \nabla_{W^{(l)}} J(W, b; x, y)$。

c. 计算 $\Delta b^{(l)} := \Delta b^{(l)} + \nabla_{b^{(l)}} J(W, b; x, y)$。

3. 更新权重参数：

$$\begin{aligned} W^{(l)} &= W^{(l)} - \alpha \left[\left(\frac{1}{m} \Delta W^{(l)} \right) + \lambda W^{(l)} \right] \\ b^{(l)} &= b^{(l)} - \alpha \left[\frac{1}{m} \Delta b^{(l)} \right] \end{aligned}$$

现在，我们可以重复梯度下降法的迭代步骤来减小代价函数 $J(W, b)$ 的值，进而训练我们的神经网络。

4 *神经网络的常用激活函数

在神经网络中，神经元的激活函数多用逻辑函数。但实际上，另外一个激活函数：tanh 函数的效果要比逻辑函数好些，因为 tanh 函数并不会将神经元的输出局限于大于 0，tanh 的函数输出值区间为[-1,1]。而在深度神经网络中，会用到效果更好的激活函数。

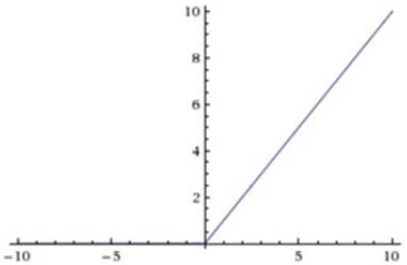
下面是在实践中可能遇到的几种激活函数：

- **Sigmoid。**sigmoid 非线性函数的数学公式是$\sigma(x) = 1/(1+e^{-x})$，它输入实数值并将其“挤压”到 0 到 1 范围内。更具体地说，很大的负数变成 0，很大的正数变成 1。在历史上，sigmoid 函数很常用，这是因为它对于神经元的激活频率有良好的解释：从完全不激活（0）到在求和后的最大频率处的完全激活（1）。

  现在深度神经网络中，sigmoid 函数已经不太受欢迎，实际很少使用了，这是因为它有个主要缺点： **Sigmoid 函数饱和使梯度消失。**sigmoid 神经元有一个不好的特性，就是当神经元的激活在接近 0 或 1 处时会饱和（指变化很小）：在这些区域，梯度几乎为 0。回忆一下，在反向传播的时候，这个（局部）梯度将会与整个代价函数关于该单元输出的梯度相乘。因此，如果局部梯度非常小，那么相乘的结果也会接近零，这会有效地“杀死”梯度，几乎就有没有信号通过神经元传到权重再到数据了。还有，为了防止饱和，必须对于权重矩阵初始化特别留意。比如，如果初始化权重过大，那么大多数神经元将会饱和，导致网络就几乎不学习了。

- **Tanh。**tanh 将实数值压缩到[-1,1]之间。和 sigmoid 神经元一样，它也存在饱和问题，但是和 sigmoid 神经元不同的是，它的输出是零中心的。因此，在实际操作中，**tanh 非线性函数比 sigmoid 非线性函数更受欢迎。**注意 tanh 神经元是一个简单放大的 sigmoid 神经元，具体说来就是：$tanh(x) = 2\sigma(2x) - 1$。

- **ReLU。**在近些年 ReLU 变得非常流行。它的函数公式是$f(x) = max(0, x)$。换句话说，这个激活函数就是一个关于 0 的阈值。



使用 ReLU 有以下一些优缺点：

- 优点：相较于 sigmoid 和 tanh 函数，ReLU 对于随机梯度下降的收敛有巨大的加速作用（ Krizhevsky 等的论文指出有 6 倍之多）。据称这是由它的线性，非饱和的公式导致的。sigmoid 和 tanh 神经元含有指数运算等耗费计算资源的操作，而 ReLU 可以简单地通过对一个矩阵进行阈值计算得到。

•缺点：在训练的时候，ReLU 单元比较脆弱并且可能“死掉”。举例来说，当一个很大的梯度流过 ReLU 的神经元的时候，可能会导致梯度更新到一种特别的状态，在这种状态下神经元将无法被其他任何数据点再次激活。如果这种情况发生，那么从此所以流过这个神经元的梯度将都变成 0。也就是说，这个 ReLU 单元在训练中将不可逆转的死亡，因为这导致了数据多样化的丢失。例如，如果学习率设置得太高，可能会发现网络中 40%的神经元都会死掉（在整个训练集中这些神经元都不会被激活）。通过合理设置学习率，这种情况的发生概率会降低。

- **Leaky ReLU。**Leaky ReLU 是为解决“ReLU 死亡”问题的尝试。ReLU 中当 x<0 时，函数值为 0。而 Leaky ReLU 则是给出一个很小的负数梯度值，比如 0.01。所以其函数公式为$f(x) = 1(x < 0)(\alpha x) + 1(x >= 0)(x)$其中$\alpha$是一个小的常量。有些研究者的论文指出这个激活函数表现很不错，但是其效果并不是很稳定。
- **Maxout。**一些其他类型的单元被提了出来，它们对于权重和数据的内积结果不再使用$f(w^T x + b)$函数形式。一个相关的流行选择是 Maxout（最近由 Goodfellow 等发布）神经元。Maxout 是对 ReLU 和 leaky ReLU 的一般化归纳，它的函数是：$max(w_1^T x + b_1, w_2^T x + b_2)$。ReLU 和 Leaky ReLU 都是这个公式的特殊情况（比如 ReLU 就是当$w_1, b_1 = 0$的时候）。这样 Maxout 神经元就拥有 ReLU 单元的所有优点（线性操作和不饱和），而没有它的缺点（死亡的 ReLU 单元）。然而和 ReLU 对比，它每个神经元的参数数量增加了一倍，这就导致整体参数的数量激增。

5 *常见神经网络模型

这一节介绍常见的神经网络模型。

- **Boltzmann 机和受限 Boltzmann 机**

神经网络中有一类模型是为网络状态定义一个“能量”，能量最小化时网络达到理想状态，而网络的训练就是在最小化这个能量函数。Boltzmann（玻尔兹曼）机就是基于能量的模型，其神经元分为两层：显层和隐层。显层用于表示数据的输入和输出，隐层则被理解为数据的内在表达。Boltzmann 机的神经元都是布尔型的，即只能取 0、1 值。标准的 Boltzmann 机是全连接的，也就是说各层内的神经元都是相互连接的，因此计算复杂度很高，而且难以用来解决实际问题。因此，我们经常使用一种特殊的 Boltzmann 机——受限玻尔兹曼机（Restricted Boltzmann Mechine，简称 RBM），它层内无连接，层间有连接，可以看做是一个二部图。：

- **RBF 网络**

RBF（Radial Basis Function）径向基函数网络是一种单隐层前馈神经网络，它使用径向基函数作为隐层神经元激活函数，而输出层则是对隐层神经元输出的线性组合。

- **ART 网络**

ART（Adaptive Resonance Theory）自适应谐振理论网络是竞争型学习的重要代表，该网络由比较层、识别层、识别层阈值和重置模块构成。ART 比较好的缓解了竞争型学习

中的“可塑性-稳定性窘境”（stability-plasticity dilemma），可塑性是指神经网络要有学习新知识的能力，而稳定性则指的是神经网络在学习新知识时要保持对旧知识的记忆。这就使得 ART 网络具有一个很重要的优点：可进行增量学习或在线学习。

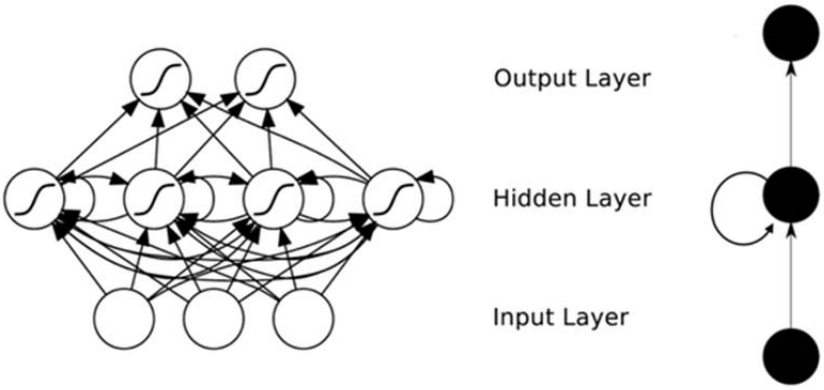
- **SOM 网络**

SOM（Self-Organizing Map，自组织映射）网络是一种竞争学习型的无监督神经网络，它能将高维输入数据映射到低维空间（通常为二维），同事保持输入数据在高维空间的拓扑结构，即将高维空间中相似的样本点映射到网络输出层中的临近神经元。

- **递归神经网络以及 Elman 网络**

与前馈神经网络不同，递归神经网络（Recurrent Neural Networks，简称 RNN）允许网络中出现环形结构，从而可以让一些神经元的输出反馈回来作为输入信号，这样的结构与信息反馈过程，使得网络在 t 时刻的输出状态不仅与 t 时刻的输入有关，还与 t−1 时刻的网络状态有关，从而能处理与时间有关的动态变化。

Elman 网络是最常用的递归神经网络之一，其结构如下图所示：



为了解决时间轴上的梯度发散，长短时记忆单元（Long-Short Term Memory，简称 LSTM），通过门的开关实现时间上的记忆功能，并防止梯度发散。其实除了学习历史信息，RNN 和 LSTM 还可以被设计成为双向结构，即双向 RNN、双向 LSTM，同时利用历史和未来的信息。

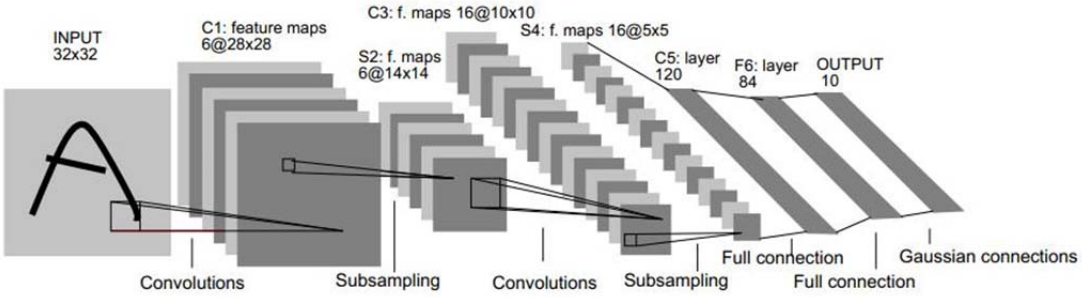
- **深度学习**

深度学习指的是深度神经网络模型，一般指网络层数在三层或者三层以上的神经网络结构。

理论上而言，参数越多的模型复杂度越高，“容量”也就越大，也就意味着它能完成更复杂的学习任务。就像前面多层感知机带给我们的启示一样，神经网络的层数直接决定了它对现实的刻画能力。但是在一般情况下，复杂模型的训练效率低，易陷入过拟合，因此难以受到人们的青睐。具体来讲就是，随着神经网络层数的加深，优化函数越来越容易陷入局部最优解。同时，不可忽略的一个问题是随着网络层数增加，“**梯度消失**”现象更加严重。我们经常使用 sigmoid 函数作为隐层的功能神经元，对于幅度为 1 的信号，在 BP 反向传播梯度时，每传递一层，梯度衰减为原来的 0.25。层数一多，梯度指数衰减后低层基本接收不到有效的训练信号。

为了解决深层神经网络的训练问题，一种有效的手段是采取无监督逐层训练（unsupervised layer-wise training），其基本思想是每次训练一层隐节点，训练时将上一层隐节点的输出作为输入，而本层隐节点的输出作为下一层隐节点的输入，这被称之为“预训练”（pre-training）；在预训练完成后，再对整个网络进行“微调”（fine-tunning）训练。比如 Hinton 在深度信念网络（Deep Belief Networks，简称 DBN）中，每层都是一个 RBM，即整个网络可以被视为是若干个 RBM 堆叠而成。在使用无监督训练时，首先训练第一层，这是关于训练样本的 RBM 模型，可按标准的 RBM 进行训练；然后，将第一层预训练号的隐节点视为第二层的输入节点，对第二层进行预训练；... 各层预训练完成后，再利用 BP 算法对整个网络进行训练。

另一种节省训练开销的做法是进行“权共享”（weight sharing），即让一组神经元使用相同的连接权，这个策略在卷积神经网络（Convolutional Neural Networks，简称 CNN）中发挥了重要作用。下图为一个 CNN 网络示意图：



CNN 可以用 BP 算法进行训练，但是在训练中，无论是卷积层还是采样层，其每组神经元（即上图中的每一个“平面”）都是用相同的连接权，从而大幅减少了需要训练的参数数目。