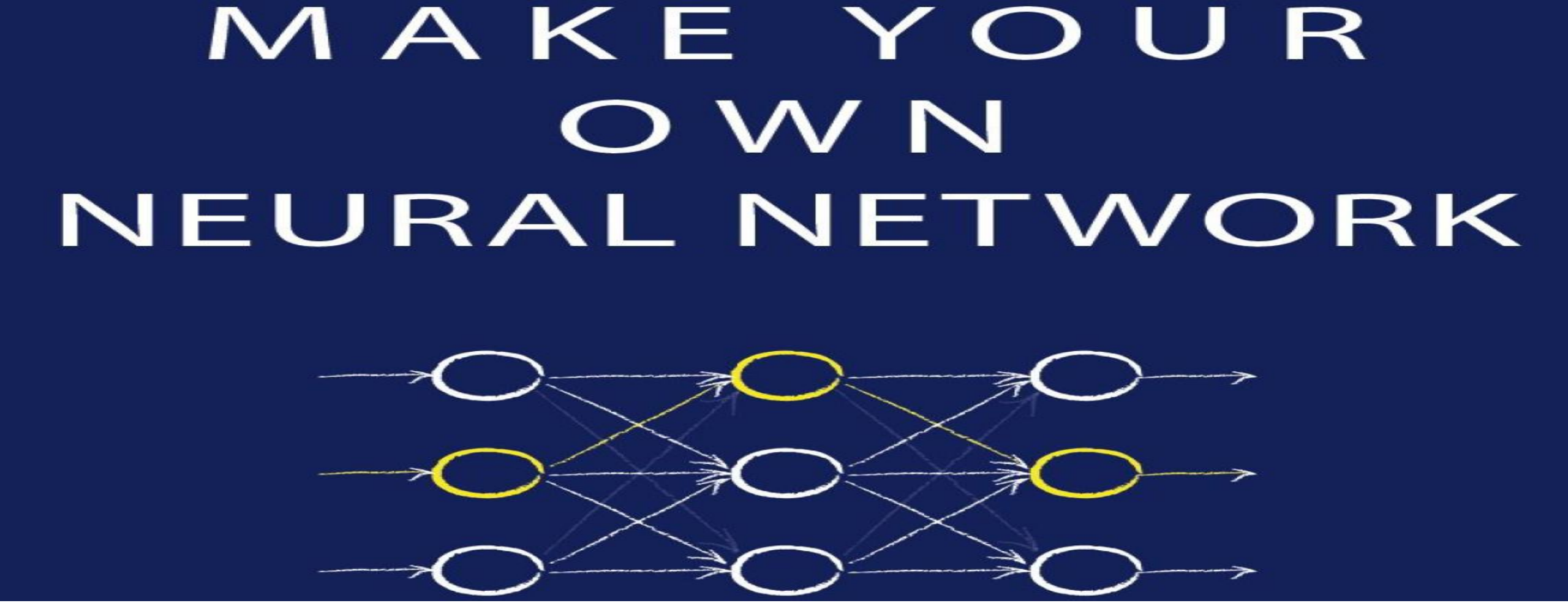


联结主义（神经网络）

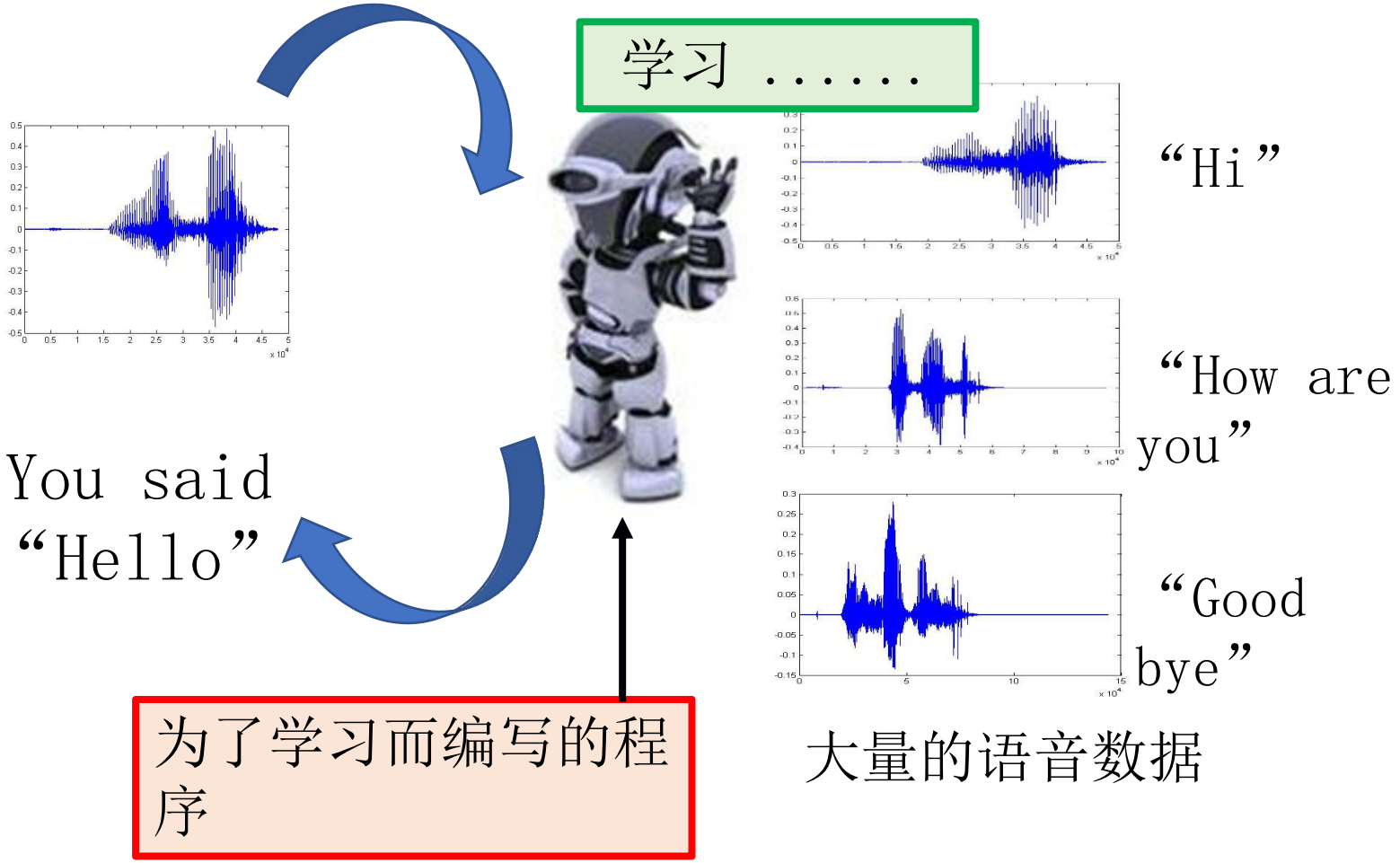
0. 机器学习先导

- 机器学习有下面几种定义：
  - “机器学习是一门人工智能的科学，该领域的主要研究对象是人工智能，特别是如何在经验学习中改善具体算法的性能”。
  - “机器学习是对能通过经验自动改进的计算机算法的研究”。
  - “机器学习是用数据或以往的经验，以此优化计算机程序的性能标准。”
  - 英文定义：A computer program is said to learn from experience E with respect to some class of tasks T and performance measure P if its performance at tasks in T, as measured by P, improves with experience E. （Tom Michael Mitchell, 1997）

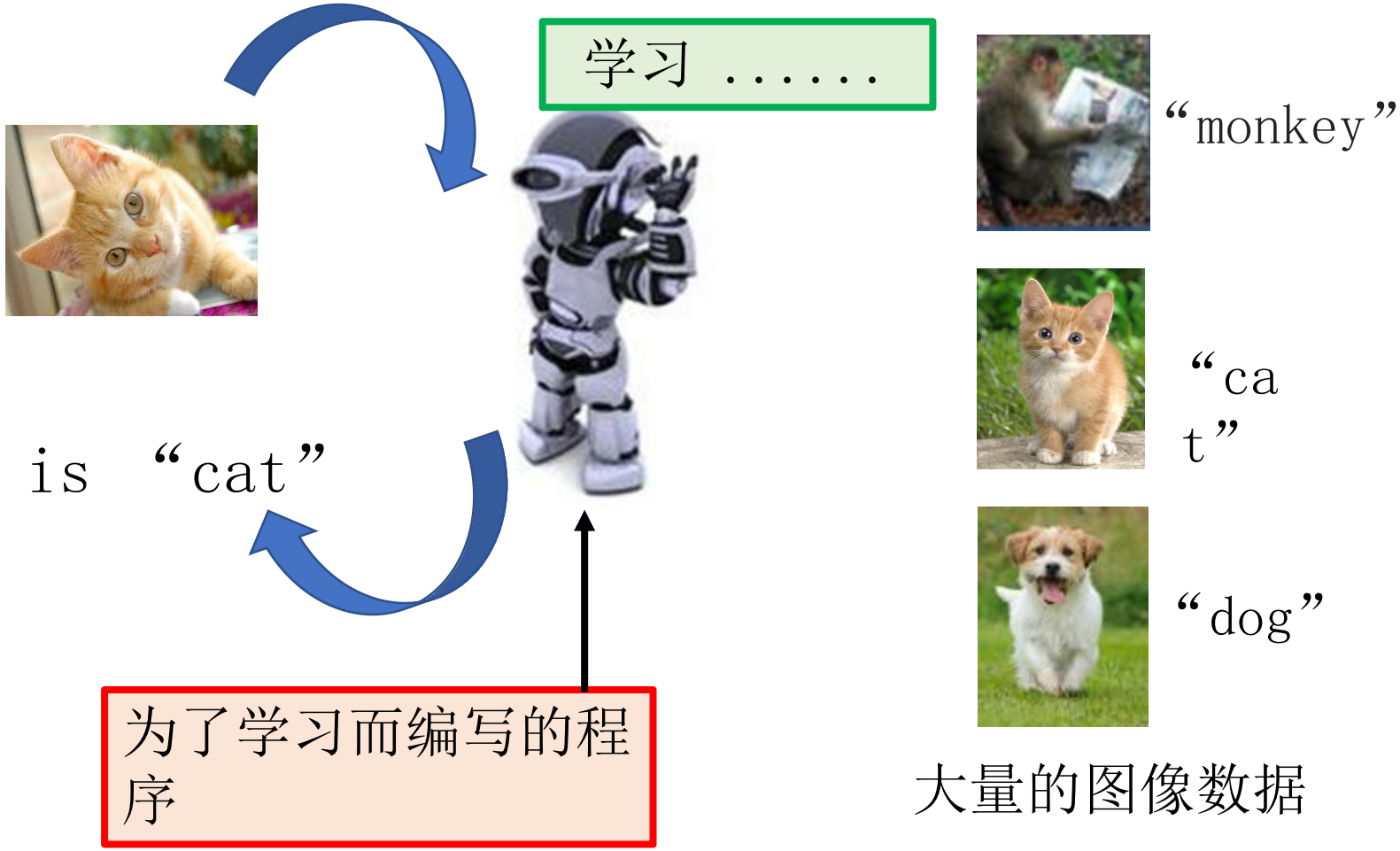
    对于某给定的任务T，在合理的性能度量方案P的前提下，某计算机程序可以自主学习任务T的经验E；随着提供合适、优质、大量的经验E,该程存对于任务T的性能逐步提高。

What Is Machine Learning?

机器学习登场

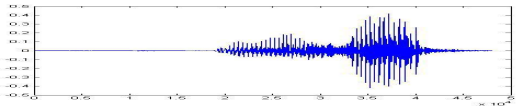


机器学习登场



机器学习≈找一个函数

- 语音识别



“How are you”

- 图像识别

“Cat”

- 下围棋

“5-5”
(next move)

- 对话系统

“How are you?”
(what the user said)

“I am fine.”
(system response)

Framework

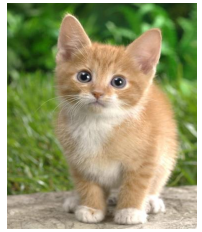
Image Recognition:

“cat”

一个函数集合

Model

“cat”



“monkey”

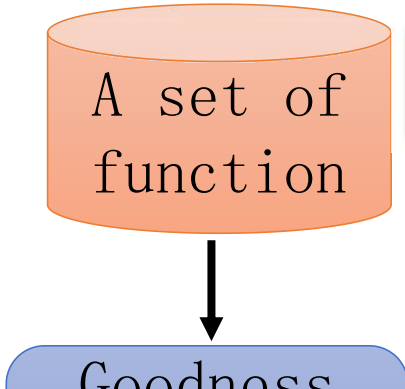
“dog”

“snake”

Framework

Image Recognition:

“cat”



Model

f_1()= “cat”

f_2()= “monkey”

Better!

f_1()= “dog”

f_2()= “snake”

Supervised Learning（监督学习）

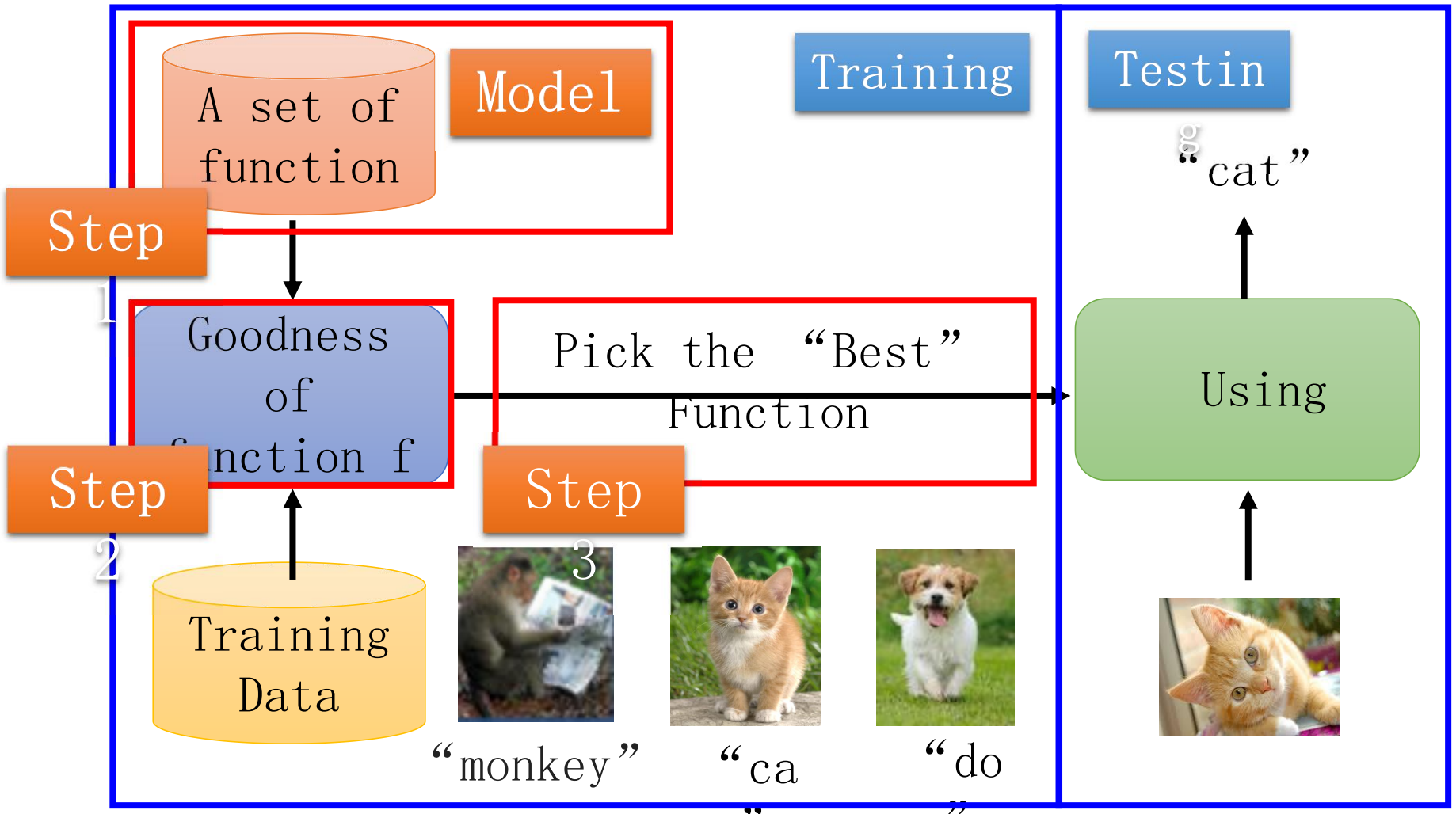
function input:

function output: “monkey” “cat” “dog”

Framework

Image Recognition:

“cat”



机器学习好简单……

Different Tasks（任务）

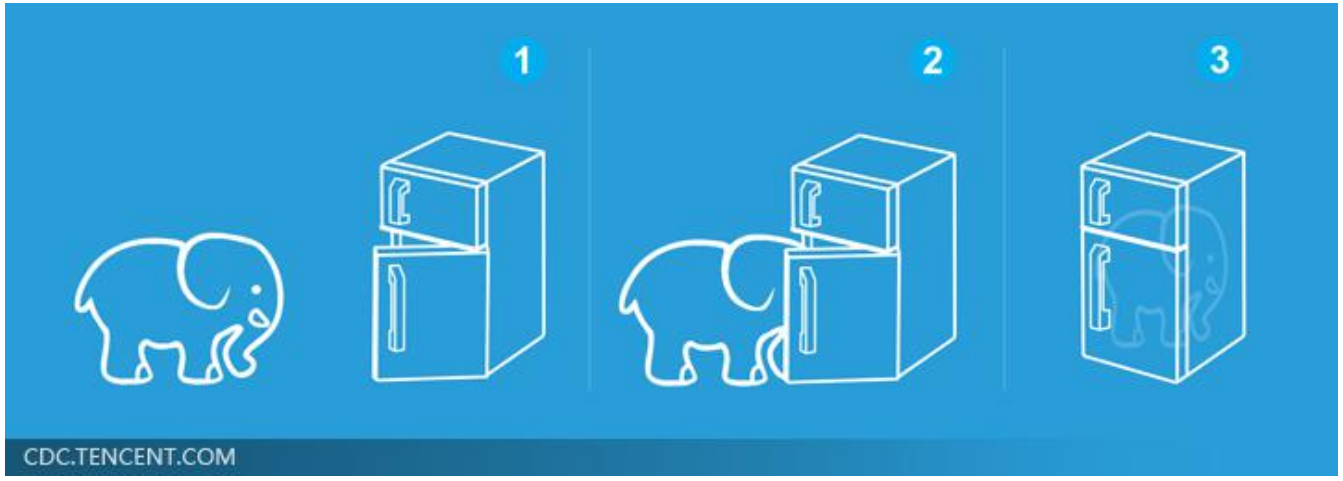
Step 0: What kind of function do you want to find?

Step 1: define a set of function

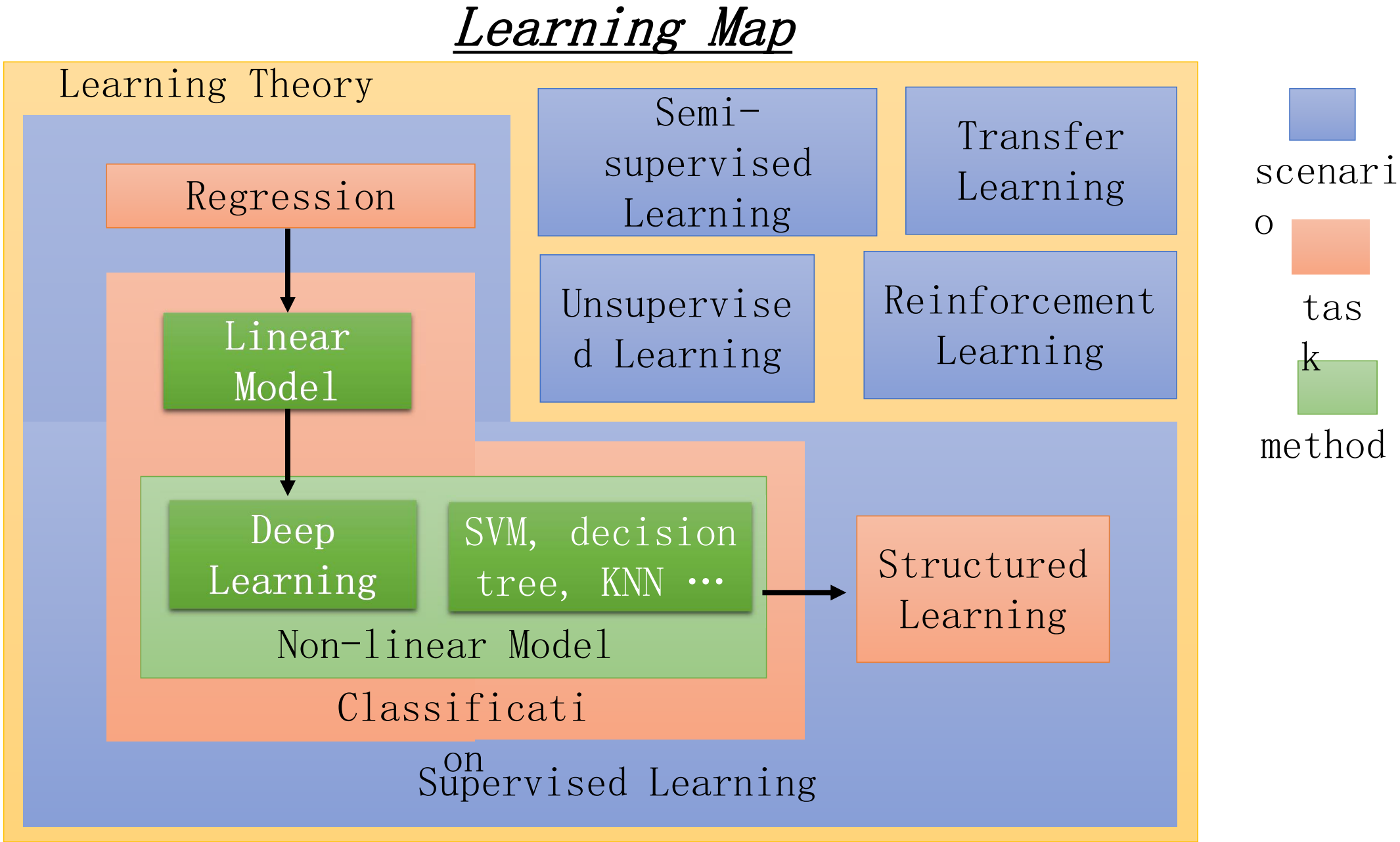
Step 2: goodness of function

Step 3: pick the best function

就好像把大象放进冰箱 .....



机器学习的类型

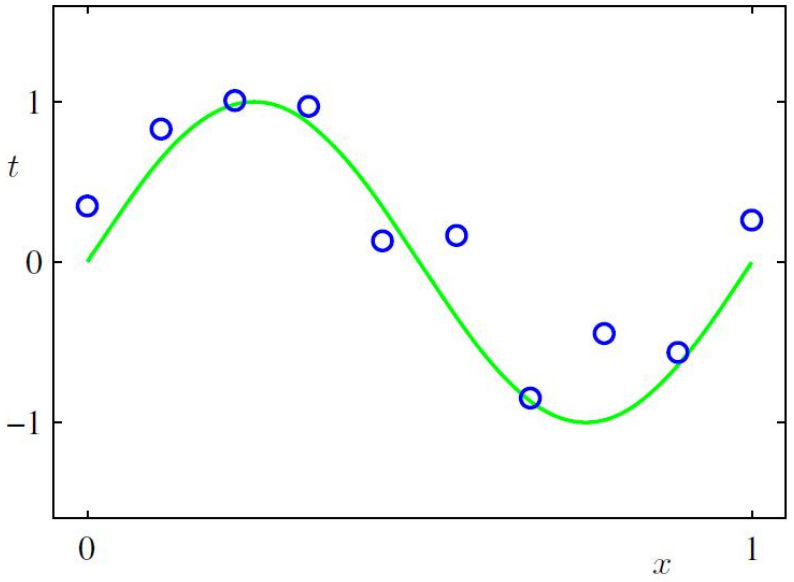

Learning Map
Learning Theory
Semi-supervised Learning
Transfer Learning
Regression
Unsupervised Learning
Reinforcement Learning
Linear Model
Deep Learning
SVM, decision tree, KNN …
Non-linear Model
Structured Learning
Classification
Supervised Learning
scenario
task
method

例子:多项式曲线拟合

目的：拟合sin(2πx)

假设：$\mathbf{x} \equiv (x_1, \ldots, x_N)^{\mathrm{T}}$

$\mathbf{t} \equiv (t_1, \ldots, t_N)^{\mathrm{T}}$

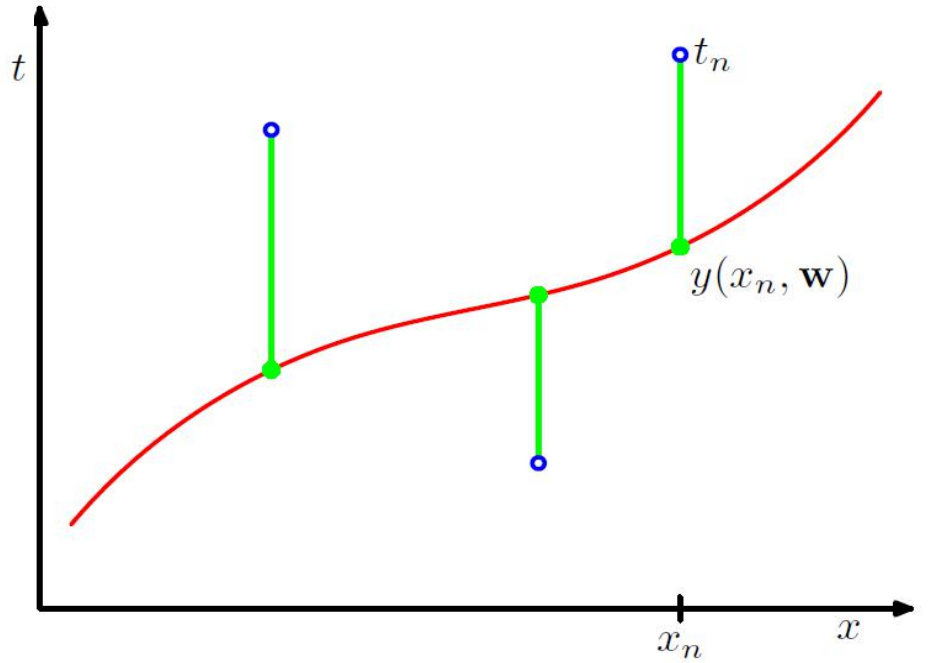


多项式拟合：

$$y(x, \mathbf{w}) = w_0 + w_1 x + w_2 x^2 + \ldots + w_M x^M = \sum_{j=0}^{M} w_j x^j$$

通过最小化一个误差函数！

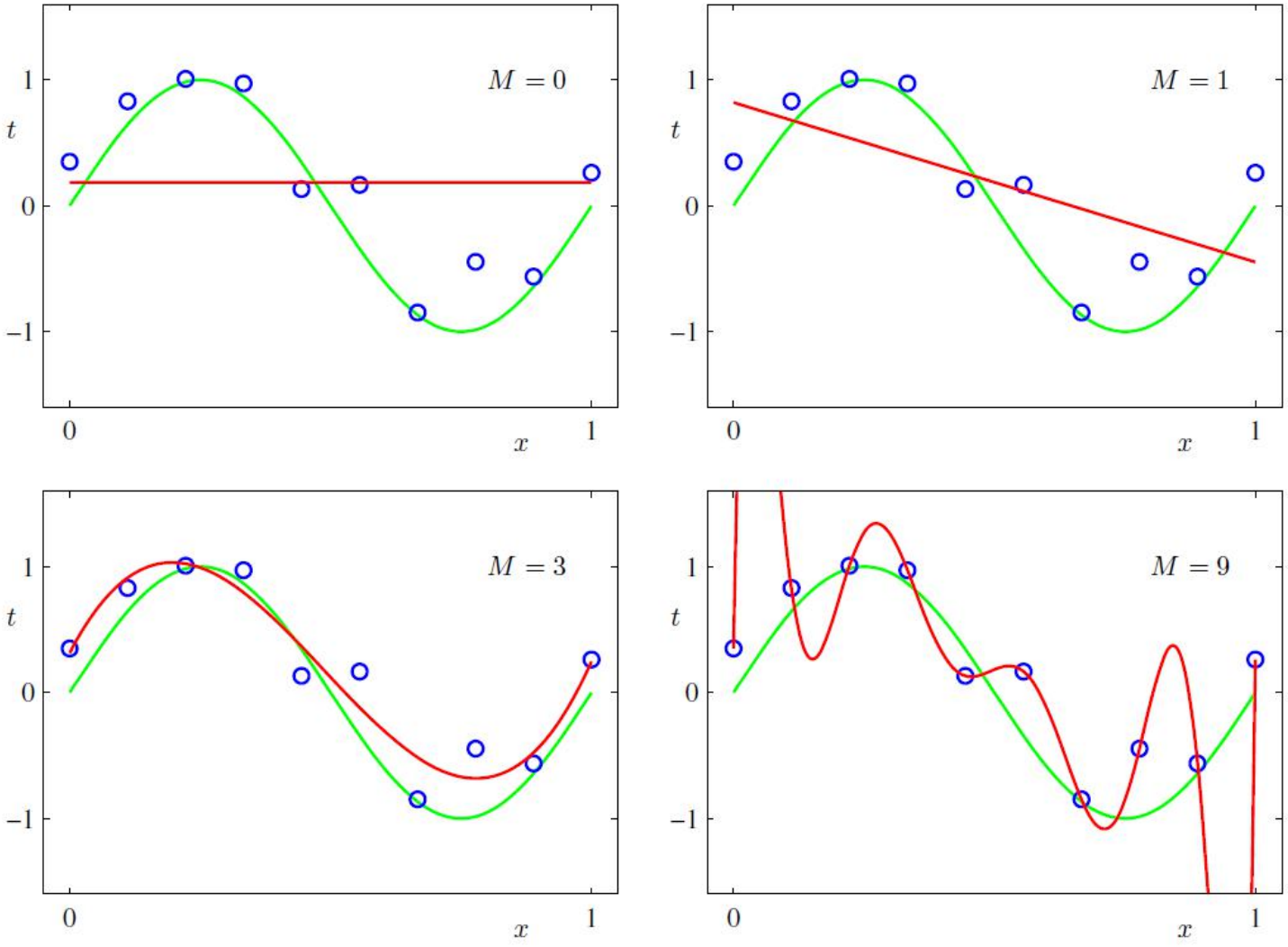
$$E(\mathbf{w}) = \frac{1}{2} \sum_{n=1}^{N} \{y(x_n, \mathbf{w}) - t_n\}^2$$



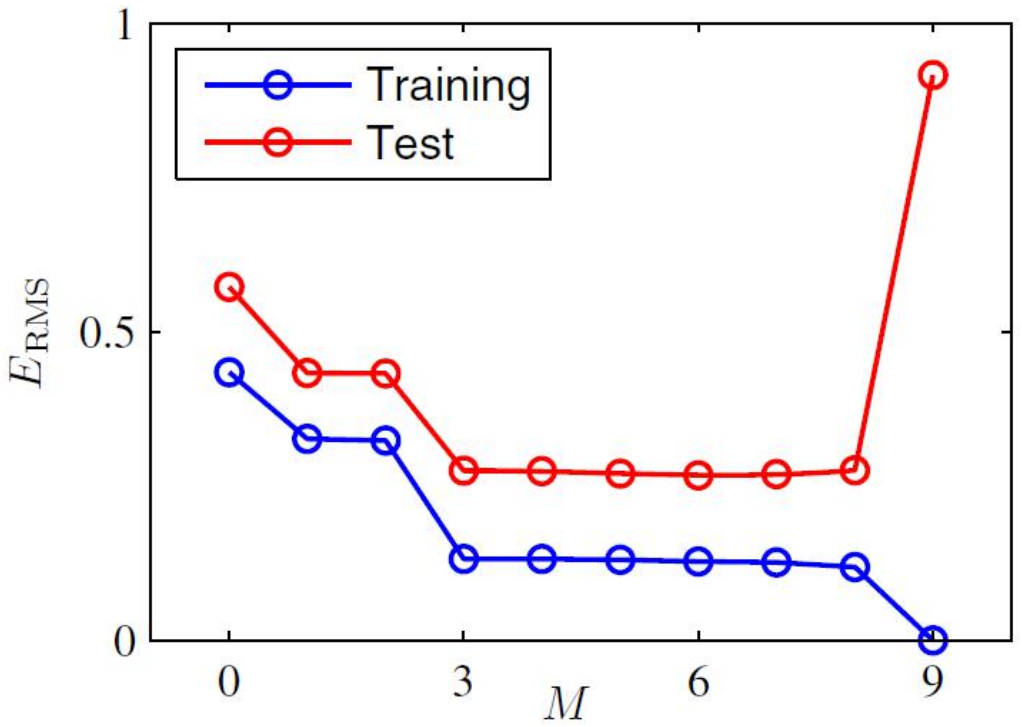
得到的多项式由函数 $y(x, \mathbf{w}^\star)$ 给出

模型选择

root-mean-square（均方根，RMS）error



$$E_{\mathrm{RMS}} = \sqrt{2E(\mathbf{w}^\star)/N}$$

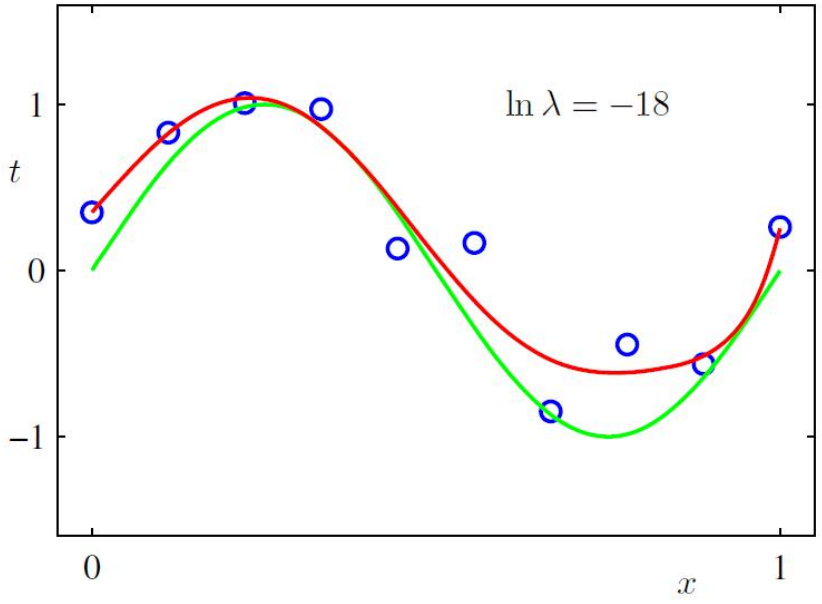


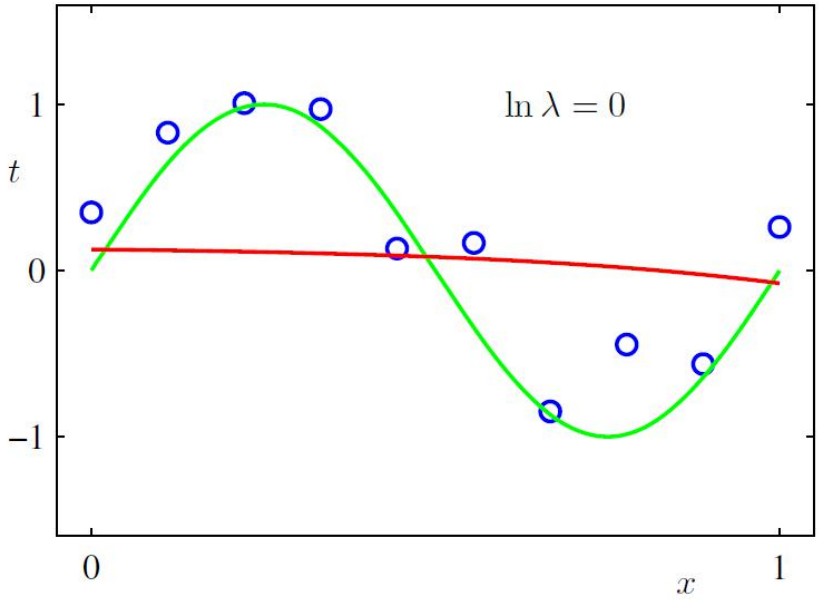
正则化---最简单的惩罚项采用所有系数的平方和的形式。

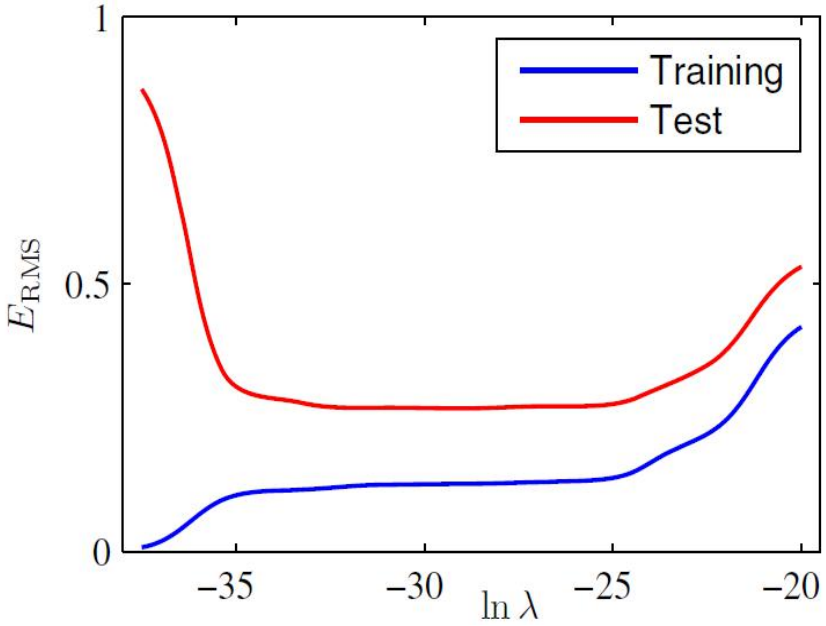
$$\widetilde{E}(\mathbf{w}) = \frac{1}{2}\sum_{n=1}^{N}\{y(x_n,\mathbf{w}) - t_n\}^2 + \frac{\lambda}{2}\|\mathbf{w}\|^2$$

$$\|\mathbf{w}\|^2 \equiv \mathbf{w}^{\mathrm{T}}\mathbf{w} = w_0^2 + w_1^2 + \ldots + w_M^2$$

与平方和误差项相比，系数 λ 决定了正则化项的相对重要性。





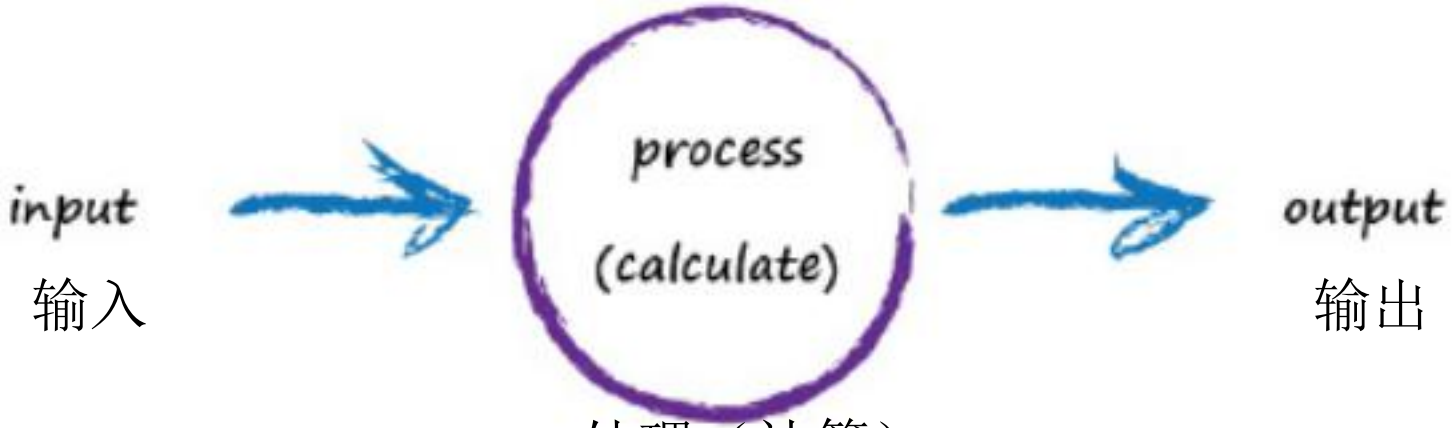


1. 神经元，自然的计算机制

我们通过我们的眼睛输入信息，用我们的大脑来分析场景，然后得出关于场景中有哪些物体的结论。想象一个基本的机器，它接受一个问题，进行一些“思考”并给出一个答案。

1. 神经元，自然的计算机制

计算机不会真正思考，它们只是美化的计算器，所以让我们用更恰当的词来描述正在发生的事情：计算机接受一些输入，进行一些计算并弹出输出。
如：处理“3 x 4”的输入，可能是通过将乘法转换为一组更简单的加法，然后弹出输出答案“12”。



输入　　处理（计算）　　输出

1. 神经元，自然的计算机制

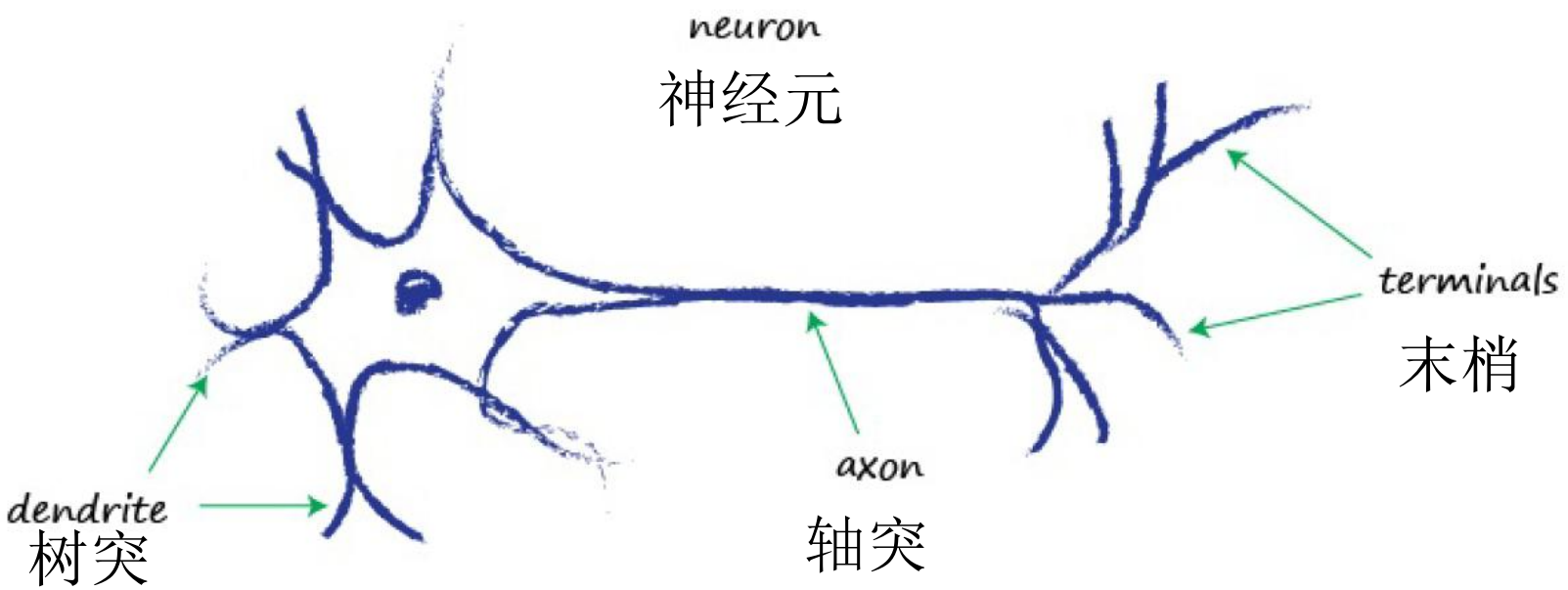
动物的大脑使科学家感到困惑，因为即使是很小的动物，具有比数字计算机要强大得多的电子计算元素数量，巨大的存储空间以及更快的运行频率。

传统电脑以非常精确的具体术语顺序处理数据，那里他们的冷硬计算没有模糊性。动物的大脑尽管节奏明显慢一些，能并行处理信号，而模糊性也是其计算的功能。

让我们看一下生物大脑的基本单位-神经元：

1. 神经元，自然的计算机制

神经元虽然有各种形式，但都从树突沿轴突到末端传输电信号。然后这些信号从一个神经元传递到另一个。这就是您的身体感应光，声音，触摸压力，热量等的方式。



1. 神经元，自然的计算机制

需要多少个神经元来执行有趣的、更复杂的任务？好吧，能力强大的人脑大约有1000亿个神经元！

一只果蝇大约有10万个神经元，能够飞行，觅食，躲避危险，
寻找食物以及许多其他相当复杂的任务。

线虫蠕虫只有302个神经元，与当今的数字计算机资源相比，这绝对是微不足道的！
但是，该蠕虫能够执行一些相当有用的任务，而规模更大的传统计算机程序
难以完成这些任务。

1. 神经元，自然的计算机制

那么秘密是什么？为什么生物大脑如此强大，尽管它们速度较慢并且由相对较少的计算元素组成？

大脑的完整功能（例如意识）仍然是一个谜，而且关于神经元的知识还不足以暗示不同的计算方式，即解决问题的不同方式。

因此，让我们看一下神经元是如何工作的。

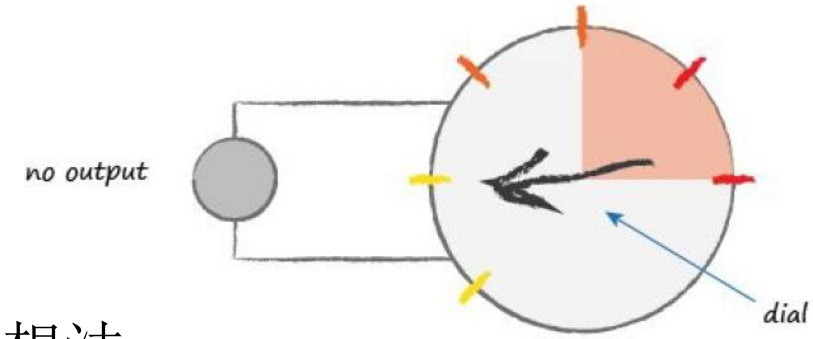
2.Activation Function（激活函数）

它需要一个电输入，然后弹出另一个电信号。这看起来就像机器学习中的分类或预测，它们接受了输入，进行了一些处理，然后弹出了输出。那么我们能像以前一样将神经元表示为线性函数吗？

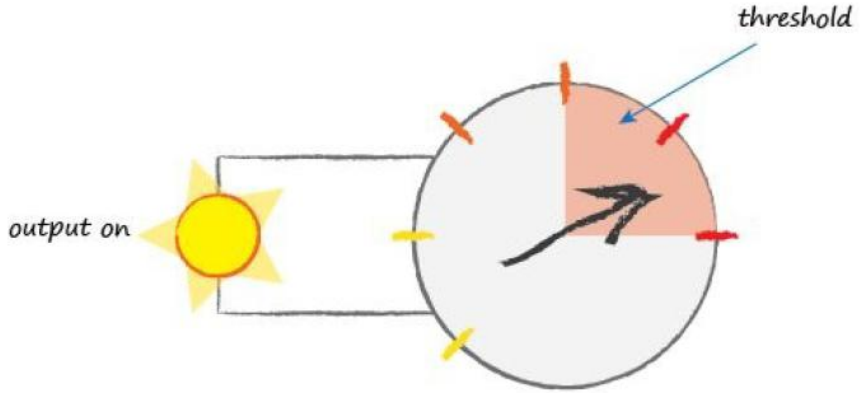
好主意，但是没有。生物神经元不会产生简单的输出输入的简单线性函数。
也就是说，其输出不采用以下形式：
输出=（常数*输入）+（也许是另一个常数）。

2. Activation Function（激活函数）

观察表明，神经元不容易做出反应，而是抑制输入，直到输入变得很大以至于触发输出。可以认为这是在产生任何输出之前必须达到的阈值。就像杯子里的水，直到杯子里的水第一次装满，水才溢出。



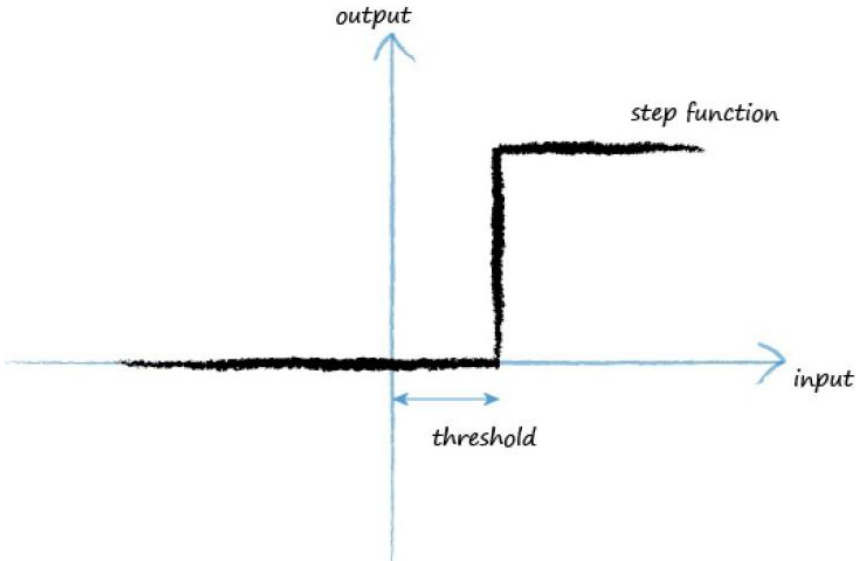
从直觉上讲，这是有道理的，神经元不想
传递微小的噪声信号，而只是传递强烈的故意信号。
仅在输入被充分拨动以超过阈值时才产生输出信号的想法。
接收输入信号并生成输出信号但考虑某种阈值的函数称为激活函数。

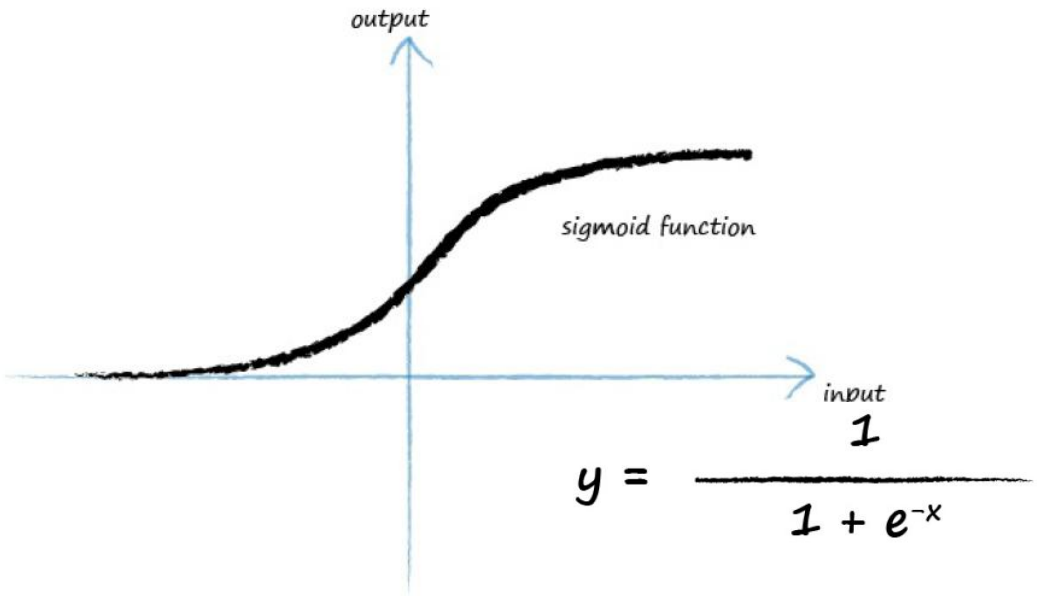


2. Activation Function（激活函数）

从数学上讲，有许多这样的激活功能可以实现此效果。
一个简单的阶跃函数可以做到这一点：对于低输入值，可以看到输出为零。但是，一旦达到阈值输入，输出就会跳升。像这样，人工神经元就像一个真正的生物神经元。当输入达到阈值时，神经元就会触发。

我们可以改善阶跃功能，如以下所示的S型函数。它比硬阶跃功能更平滑，这使它更加自然和逼真。大自然很少有冷硬的边缘！我们将继续使用这种光滑的S型函数来制作自己的神经网络。



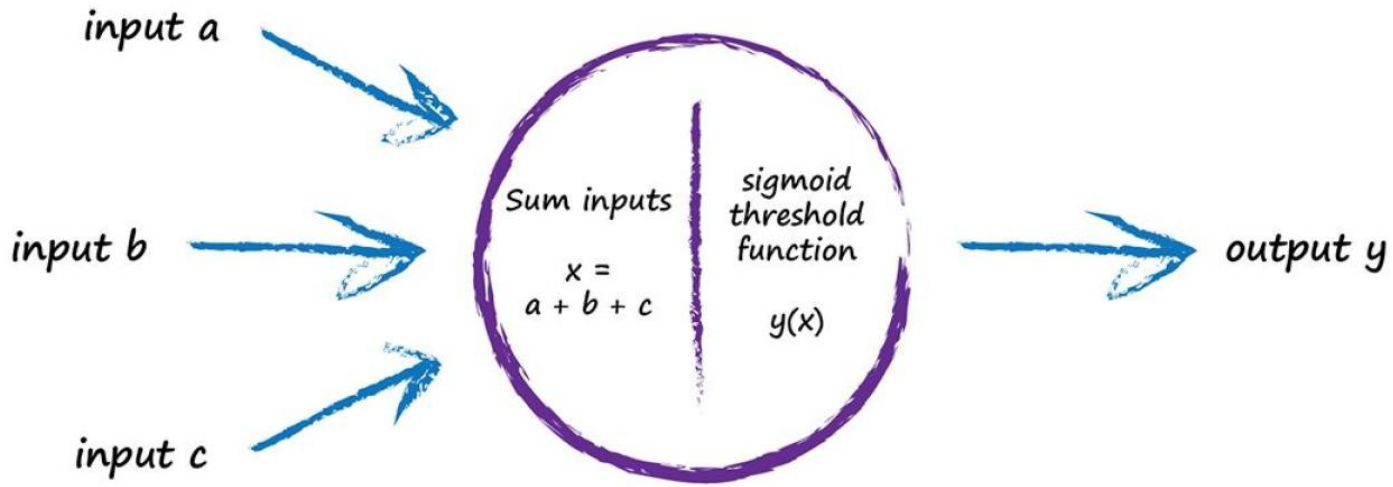


3. An Artificial Neuron（人工神经元）

让我们回到神经元，并考虑如何建模人工神经元。

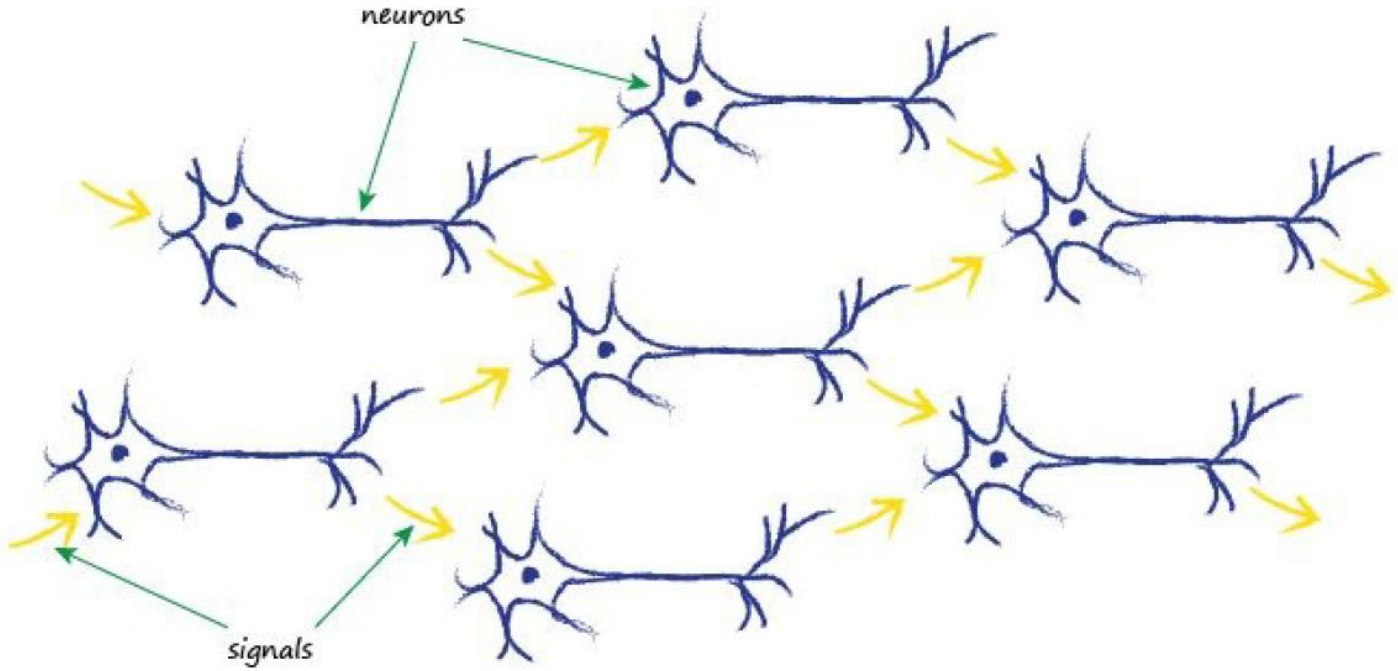
首先要意识到的是，真正的生物神经元需要很多输入，而不仅仅是一个。在布尔逻辑机有两个输入时就看到了这一点，因此拥有多个输入的想法并不是新的，也不是不同寻常的。我们如何处理所有这些输入？

我们只需要简单地将它们相加就可以合并起来，而总和就是控制输出的S型函数的输入。如果组合信号不够大，则S型阈值函数的作用是抑制输出信号。如果总和足够大，作用就是激发神经元。这反映了真实的神经元是如何工作的。



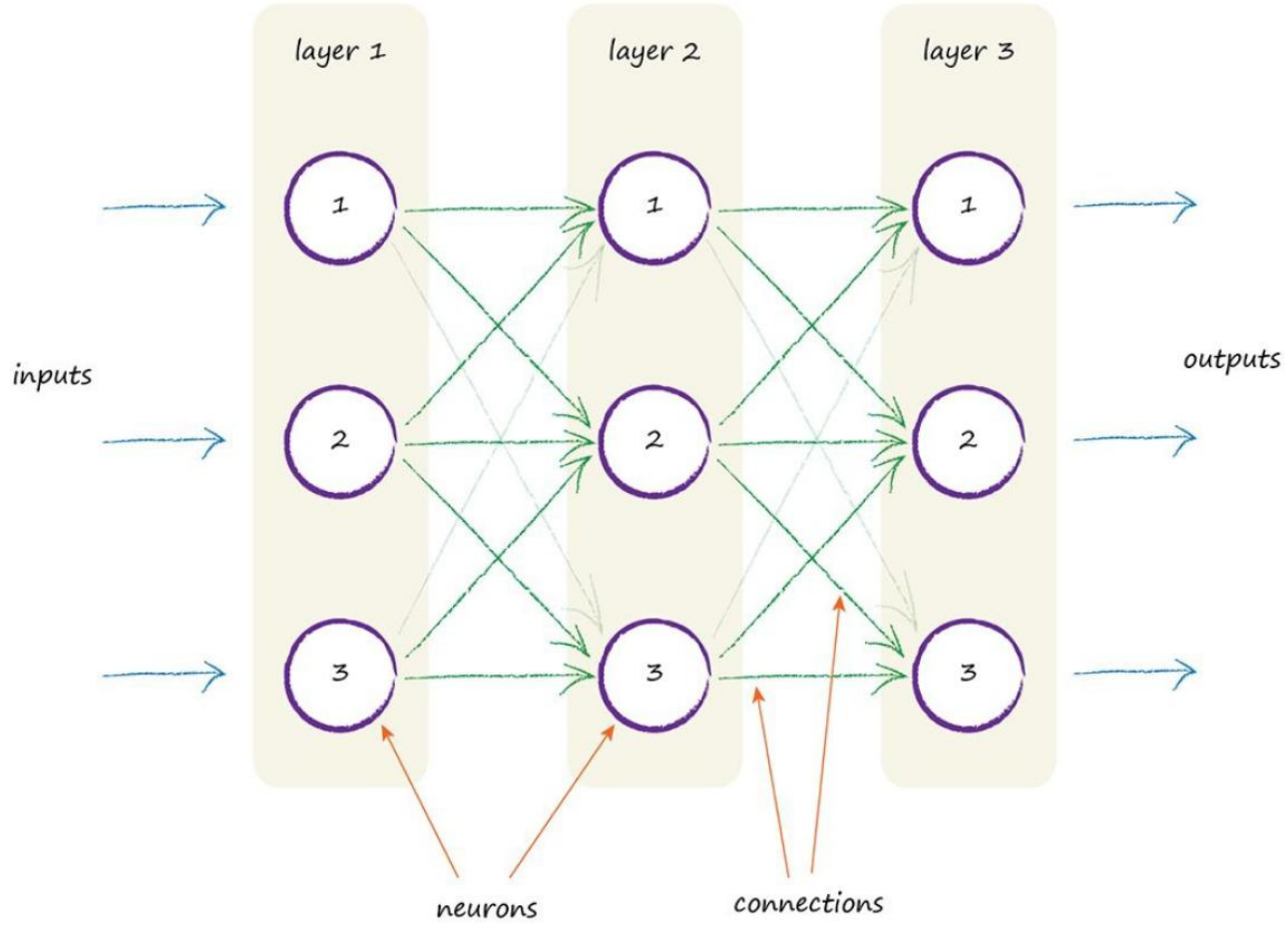
4.Several Neurons Connected（几个神经元的连接）

通过直观的方式，您可以感受到更多这种神经元可以进行的复杂的计算。电信号被树枝状晶体收集，并且它们合并形成更强的电信号。如果信号强度足以超过阈值，则神经元沿着轴突向末端发射信号，传递至下一个神经元的树突。每个神经元都从它之前的许多输入中获取输入，并且还可以向更多的神经元提供信号。



4. An Artificial Model（人工模型）

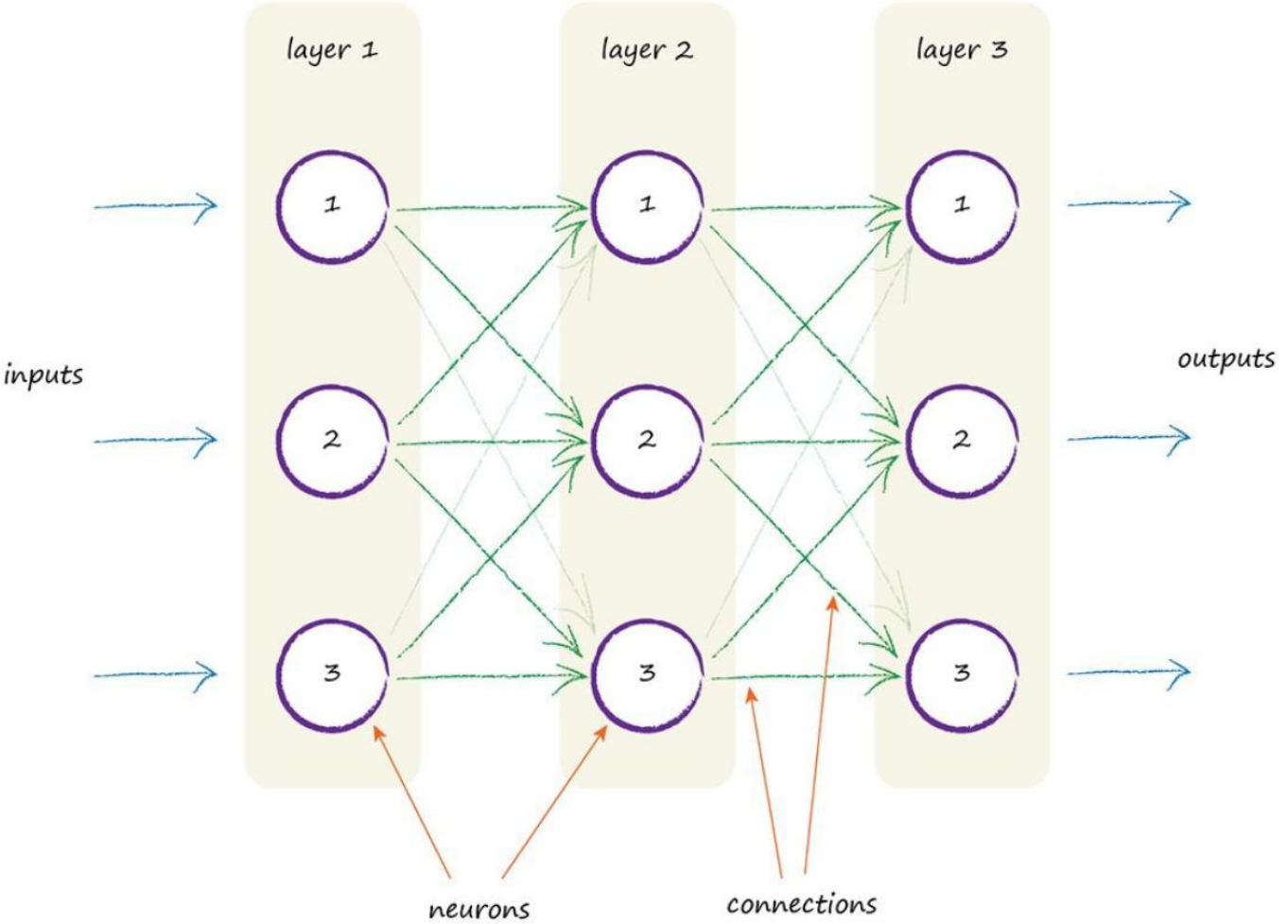
从自然界复制到人工模型的一种方法是具有神经元层，每个神经元在上一层和下一层相互连接。下图说明了此想法：您可以看到三层，每层都有三个人工神经元或节点。您还可以在上一层和下一层中看到每个节点与其他节点的连接。



4. An Artificial Model（人工模型）

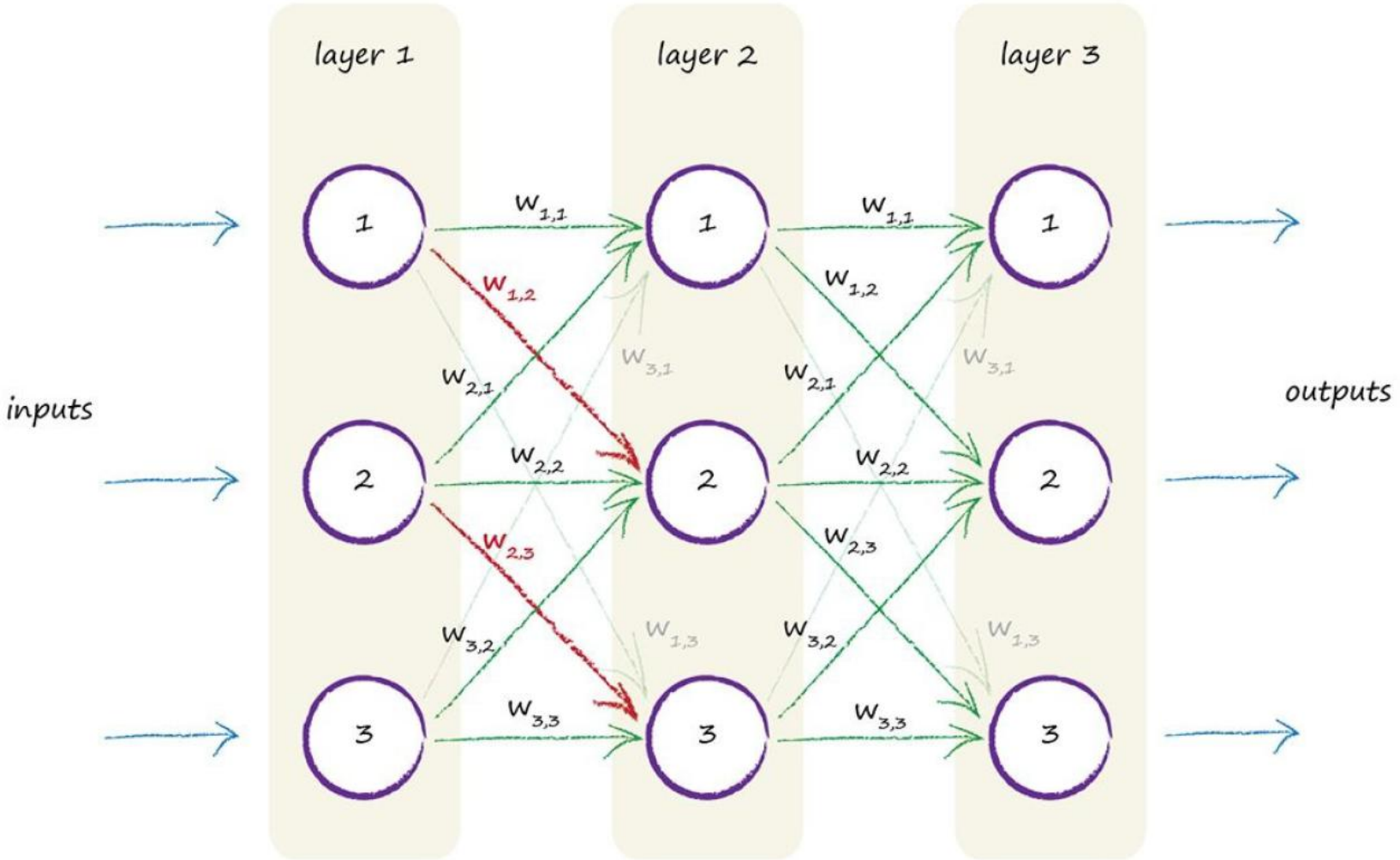
但是，学习过程在此体系结构的哪一部分进行了学习？会根据训练示例进行哪些调整？是否有一个参数可以像线性分类器的斜率那样进行优化？

最明显的是调整节点之间的连接强度。在一个节点内，我们可以调整输入的总和，也可以调整S型阈值函数的形状，但这比简单地调整节点之间的连接强度更复杂。

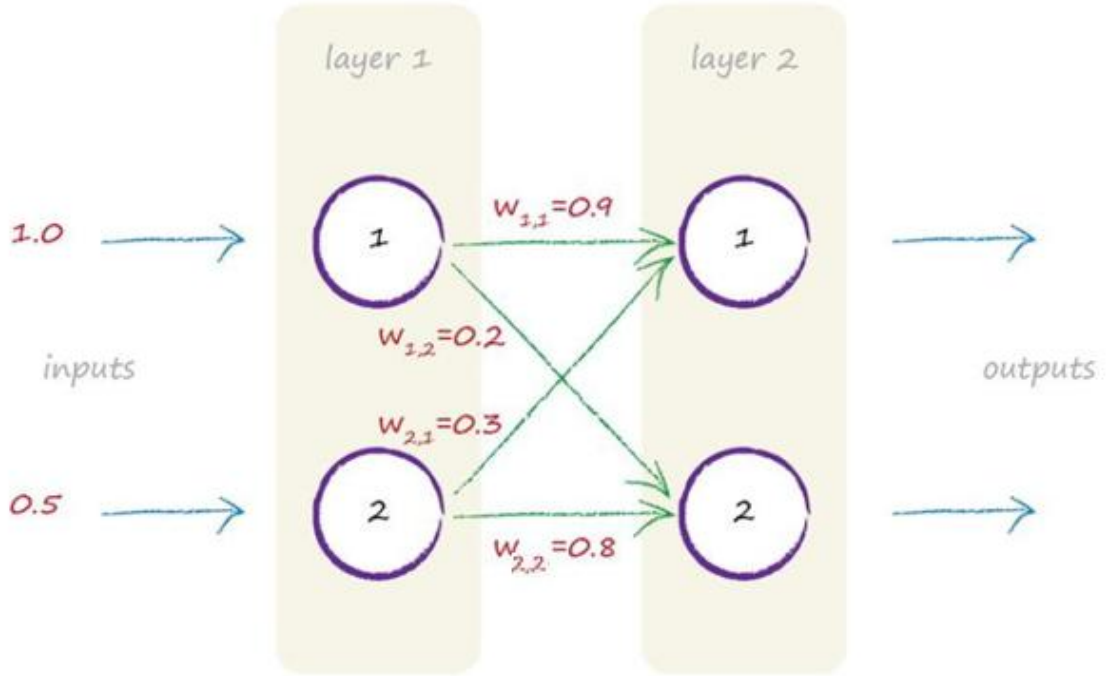


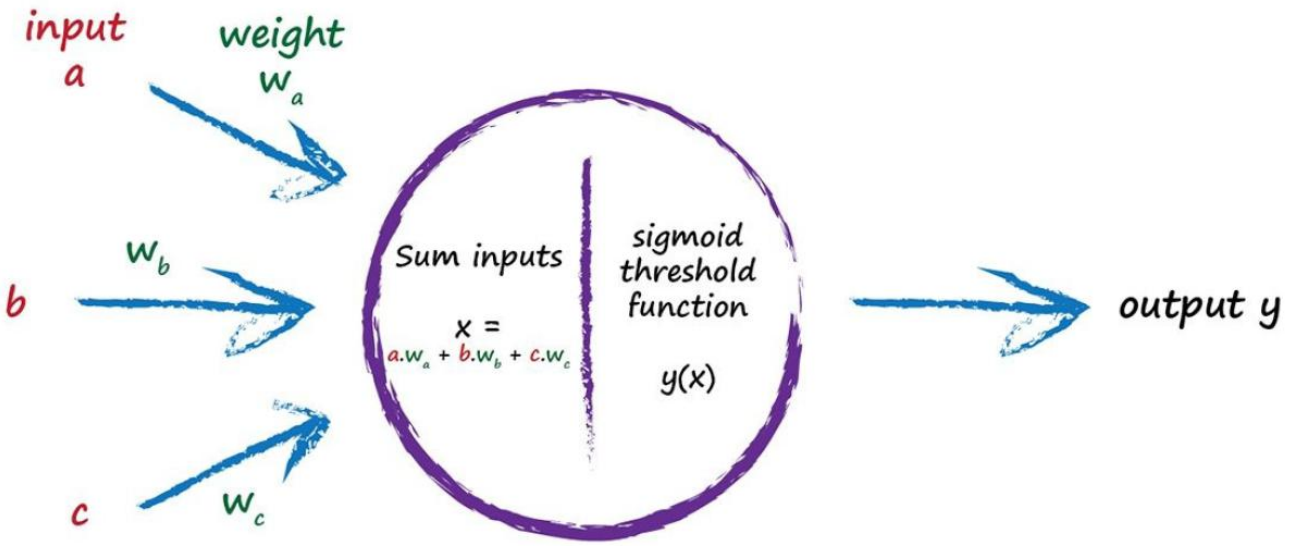
4. An Artificial Model（人工模型）

- 通过改善网络内部的链路权重来提高其输出，低权重会降低信号强度，高权重会放大信号强度。



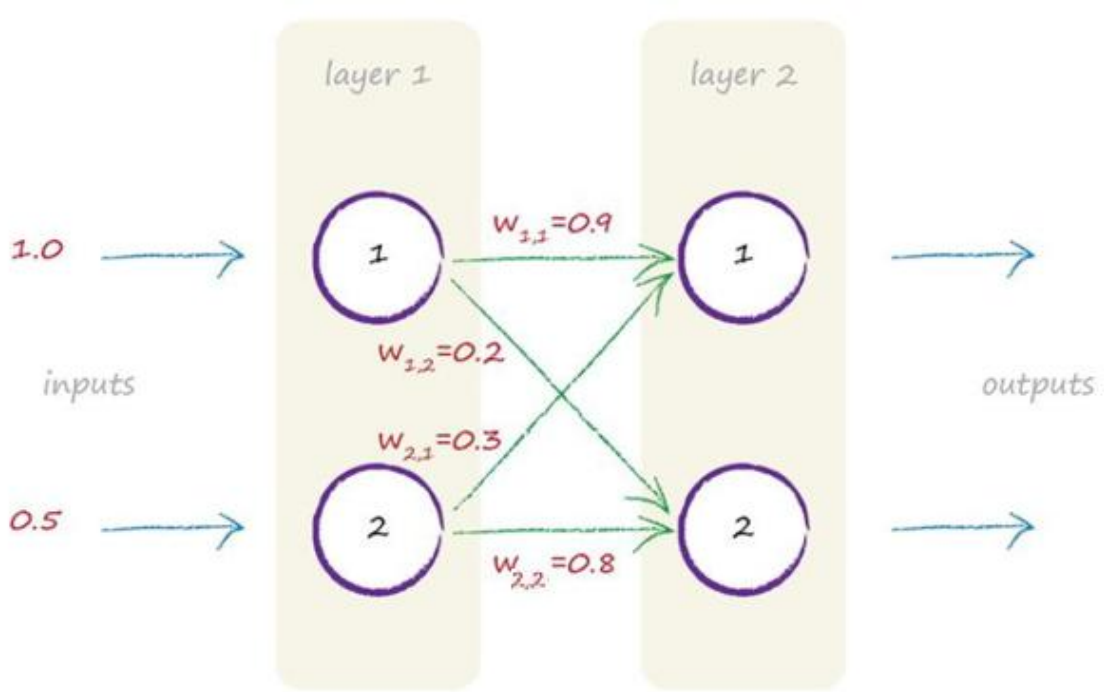
5.Following Signals Through A Neural Network
（跟随信号穿越神经网络）





该网络只有2层，每层有2个神经元，假设两个输入是1.0和0.5。
这些输入进入这个较小的神经网络，每个节点都使用激活函数将输入的总和转换为输出。
权量呢？这是一个很好的问题-他们应该从什么值开始？让我们来看看一些随机权重：
●w1,1 = 0.9●w1,2 = 0.2●w2,1 = 0.3●w2,2 = 0.8。随机起始值并不是一个坏主意，这是我们为较早的简单线性分类器选择初始斜率值时所做的事情。

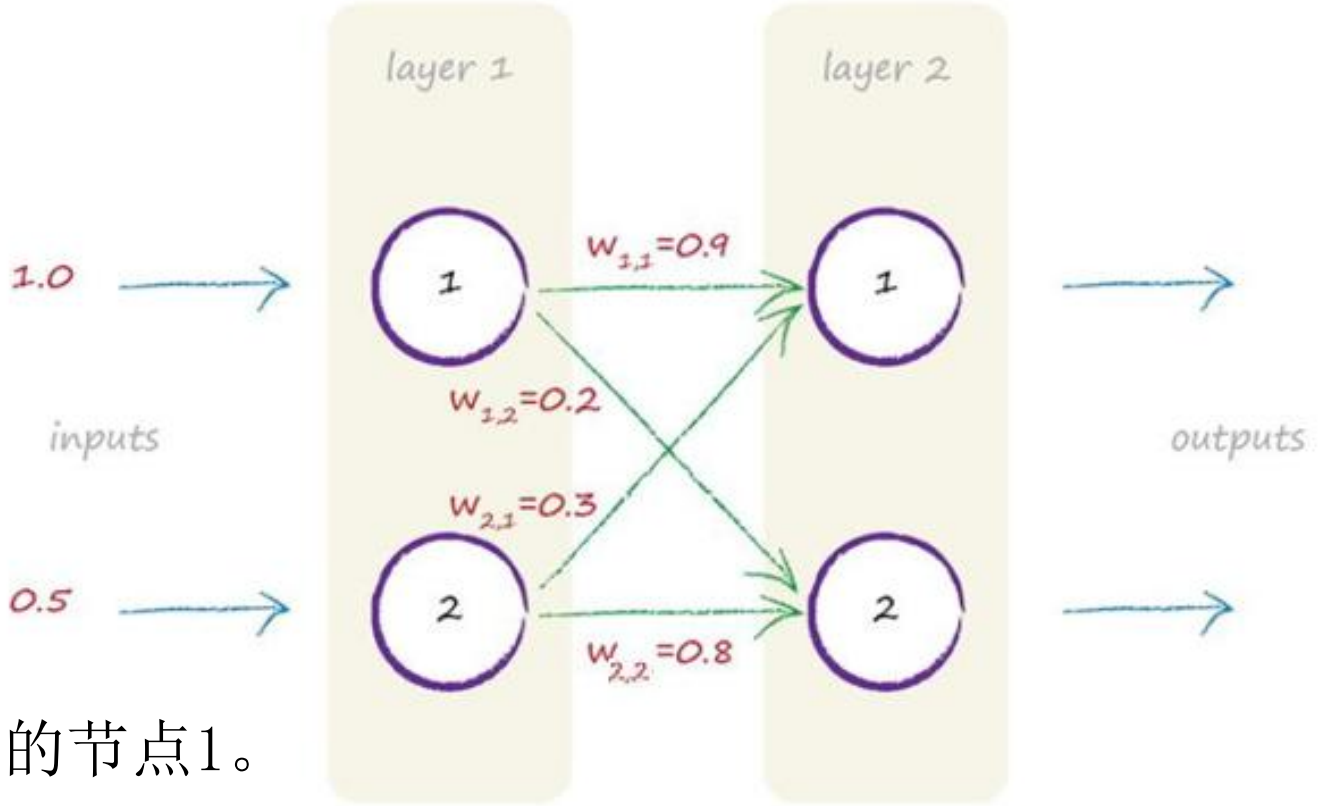
5.Following Signals Through A Neural Network
（跟随信号穿越神经网络）



Let’s start calculating!

首先，输入层1很简单——在那里不需要进行任何计算。

接下来是第二层，需要进行一些计算。



我们首先关注第2层中的节点1。
合并的输入为：

x = (output from first node * link weight) + (output from second node * link weight)

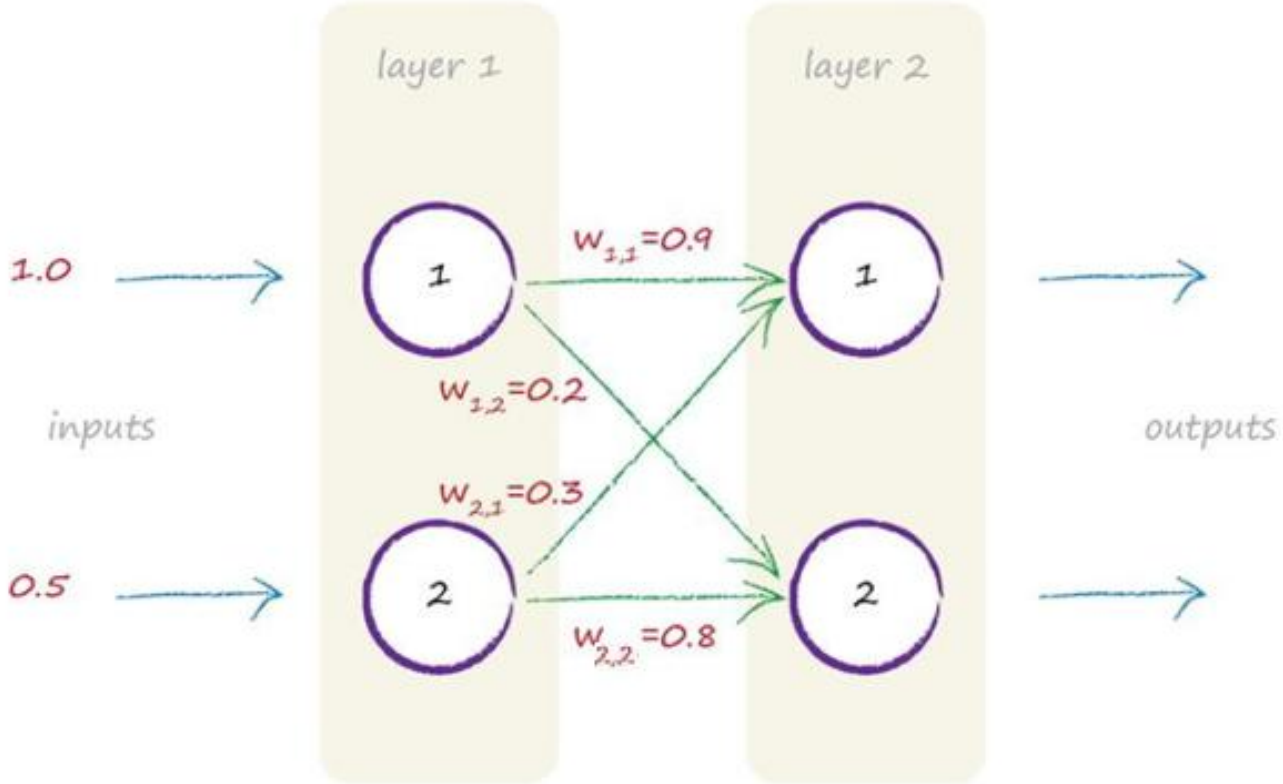
x = (1.0 * 0.9) + (0.5 * 0.3)

x = 0.9 + 0.15

x = 1.05

activation function

y = 1 / (1 + 0.3499) = 1 / 1.3499. So **y** = 0.7408.



第二层的节点2：

x = (output from first node * link weight) + (output from second node * link weight)

x = (1.0 * 0.2) + (0.5 * 0.8)

x = 0.2 + 0.4

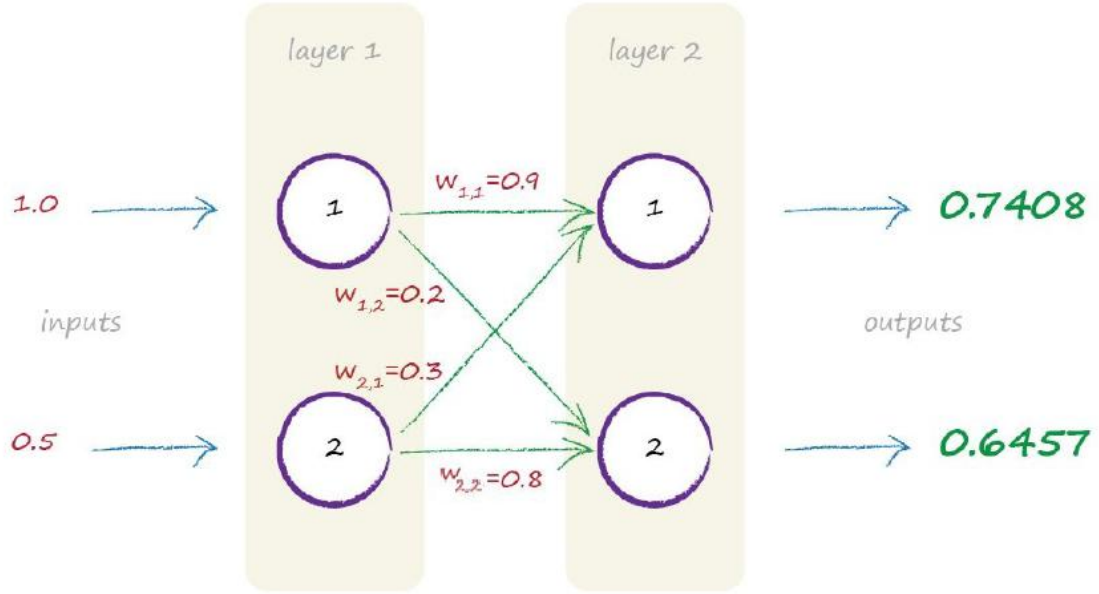
x = 0.6

activation function

$$\mathbf{y} = 1/(1 + 0.5488) = 1/(1.5488). \text{ So } \mathbf{y} = 0.6457.$$

下图仅从一个非常简化的网络中获得两个输出就可以了。 我不想手工计算更大的网络所有！幸运的是，计算机非常适合快速进行大量计算而不会感到无聊。
即使这样，我也不想写出计算机指令来进行一个多于2层，每层可能有4、8甚至100个节点的网络的计算，即使写出也会很无聊。

之前，我们是手动为每层只有2个节点的2层网络进行计算的。这项工作已经足够，但请想象一下，5层和100个节点的网络？仅写出所有必要的计算将是一项艰巨的任务……所有组合信号的组合，再乘以正确的权重，对每个节点，每一层都应用S型激活函数……太棒了！



这种简洁的方法是使用矩阵。

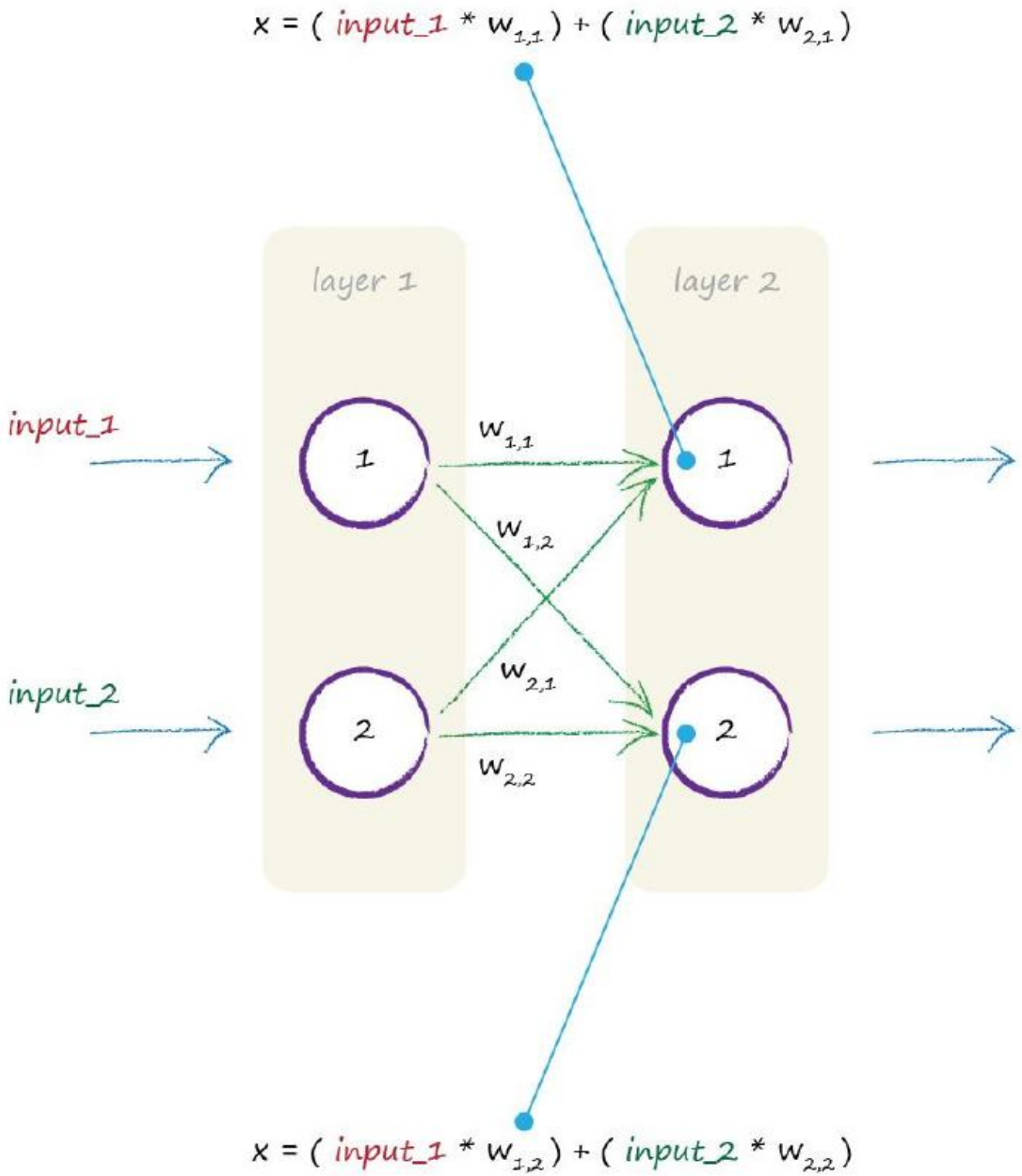
6.Matrix Multiplication is Useful .. Honest!
（矩阵乘法是有用的..真的！）

那么矩阵如何提供帮助呢？
首先，它能够将所有这些计算压缩成非常简单的简短形式。这对我们人类来说非常好，因为我们不喜欢做很多工作，因为它很无聊，而且我们还是容易出错。
第二个好处是，许多计算机编程语言都了解使用矩阵的工作，并且由于实际工作是重复的，因此他们可以识别并快速完成工作。

看看如果对神经网络用更有意义的单词替换数字会发生什么。
第一矩阵包含两层节点之间的权重。第二矩阵包含第一输入层的信号。
通过将这两个矩阵相乘得到的答案是将经过组合的调节信号输入到第二层的节点中。
因此，可以使用矩阵乘法表示将合并的调节信号算入第二层的每个节点的所有计算。

$$\begin{pmatrix} w_{1,1} & w_{2,1} \\ w_{1,2} & w_{2,2} \end{pmatrix} \begin{pmatrix} input\_1 \\ input\_2 \end{pmatrix} = \begin{pmatrix} (input\_1 * w_{1,1}) + (input\_2 * w_{2,1}) \\ (input\_1 * w_{1,2}) + (input\_2 * w_{2,2}) \end{pmatrix}$$

6. Matrix Multiplication is Useful .. Honest!



可以简明地表示为： $\mathbf{X} = \mathbf{W} \cdot \mathbf{I}$

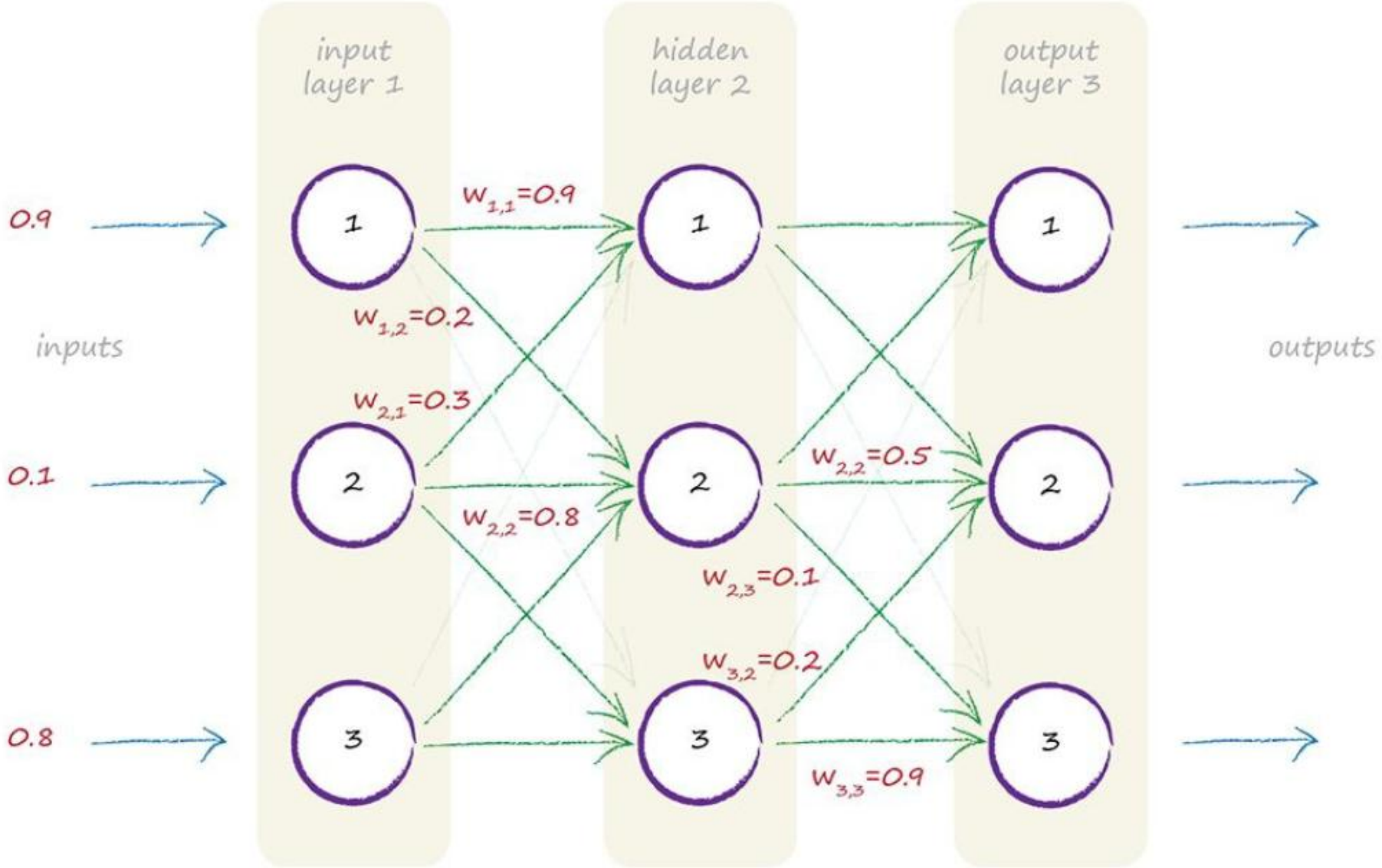
如果我们有更多的节点，矩阵将更大。
为我们提供了一个强大的工具来实现神经网。

那激活功能呢？这很容易，不需要矩阵乘法。
我们需要做的就是将S型函数应用于矩阵X的每个单独元素。

$$\mathbf{O} = \text{sigmoid}(\mathbf{X})$$

7. A Three Layer Example with Matrix Multiplication

如果有3层，只需将第二层的输出用作第三层的输入，再次简单地进行矩阵乘法。
这次我们将使用一个稍大的3层神经网络，每个网络有3个节点。



$$I = \begin{pmatrix} 0.9 \\ 0.1 \\ 0.8 \end{pmatrix}$$

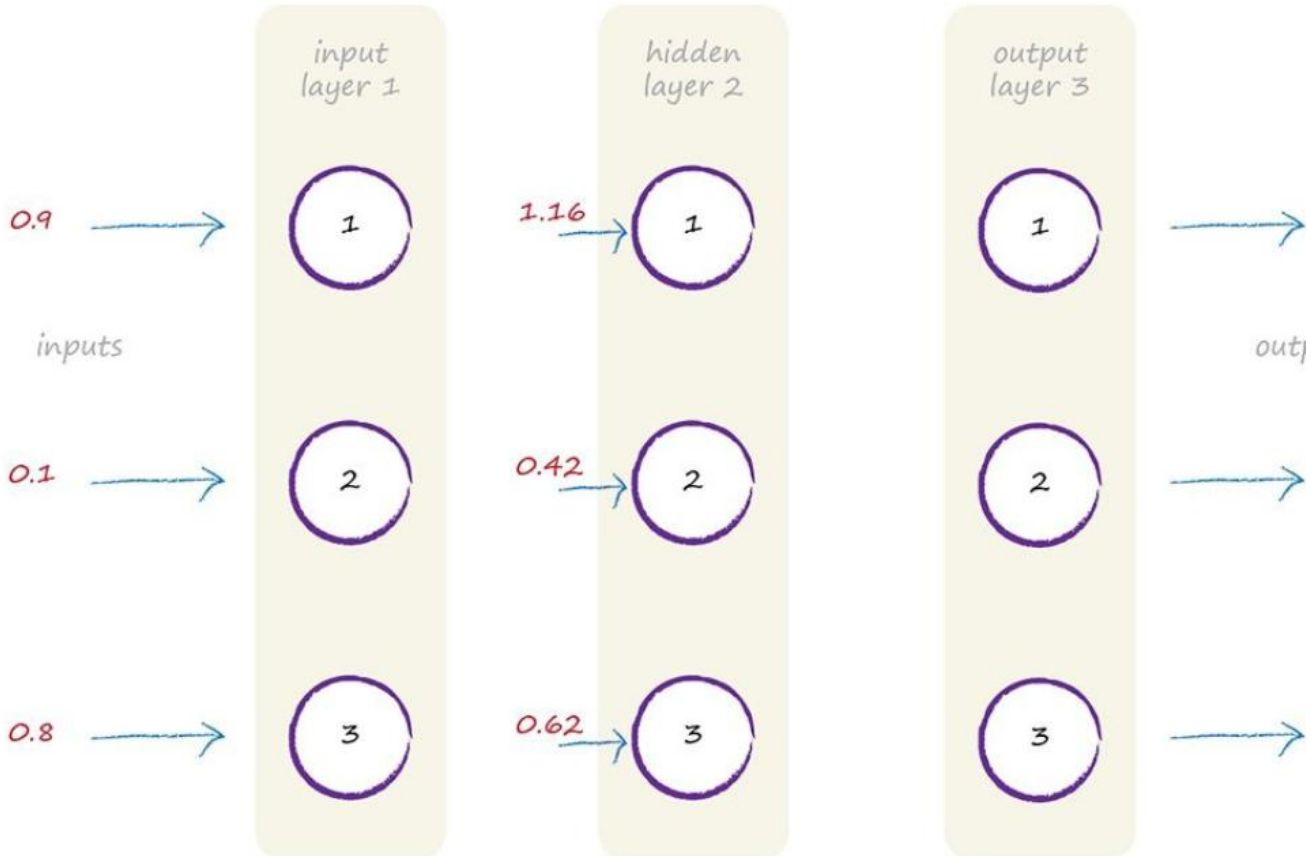
$$W_{input\_hidden} = \begin{pmatrix} 0.9 & 0.3 & 0.4 \\ 0.2 & 0.8 & 0.2 \\ 0.1 & 0.5 & 0.6 \end{pmatrix}$$

$$W_{hidden\_output} = \begin{pmatrix} 0.3 & 0.7 & 0.5 \\ 0.6 & 0.5 & 0.2 \\ 0.8 & 0.1 & 0.9 \end{pmatrix}$$

$$\mathbf{X_{hidden} = W_{input\_hidden} \cdot I}$$

$$X_{hidden} = \begin{pmatrix} 0.9 & 0.3 & 0.4 \\ 0.2 & 0.8 & 0.2 \\ 0.1 & 0.5 & 0.6 \end{pmatrix} \cdot \begin{pmatrix} 0.9 \\ 0.1 \\ 0.8 \end{pmatrix}$$

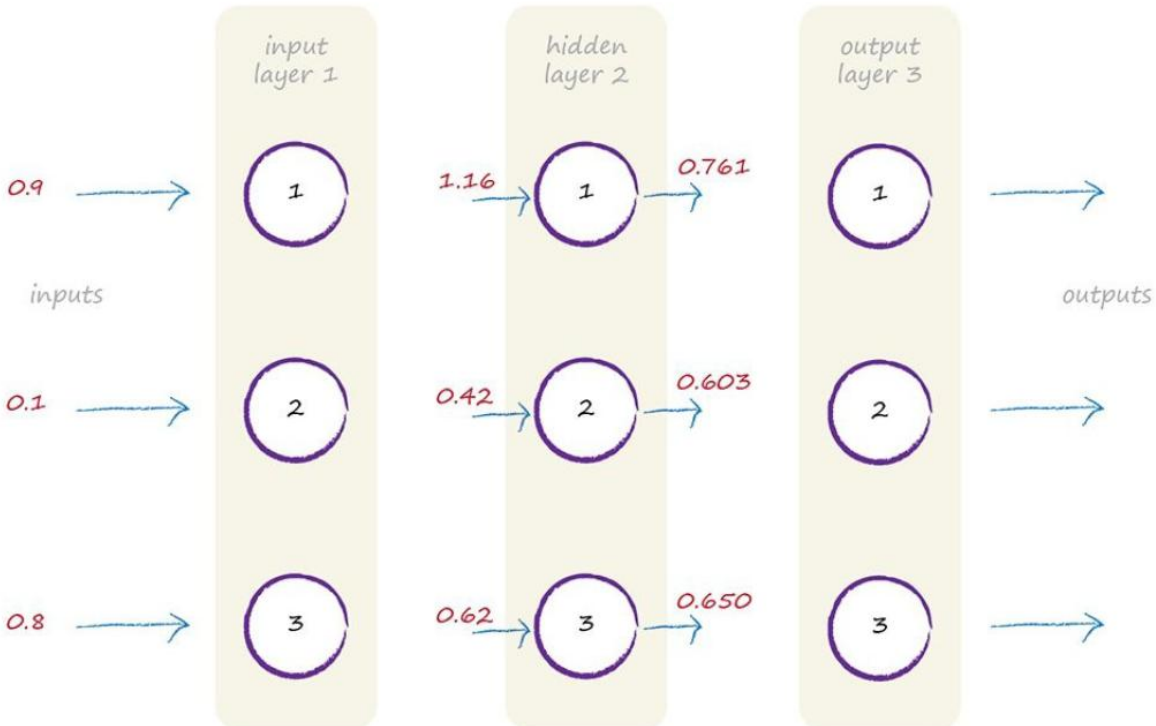
$$X_{hidden} = \begin{pmatrix} 1.16 \\ 0.42 \\ 0.62 \end{pmatrix}$$



$$\mathbf{O_{hidden}} = \text{sigmoid}(\mathbf{X_{hidden}})$$

$$O_{hidden} = \text{signmoid} \begin{pmatrix} 1.16 \\ 0.42 \\ 0.62 \end{pmatrix}$$

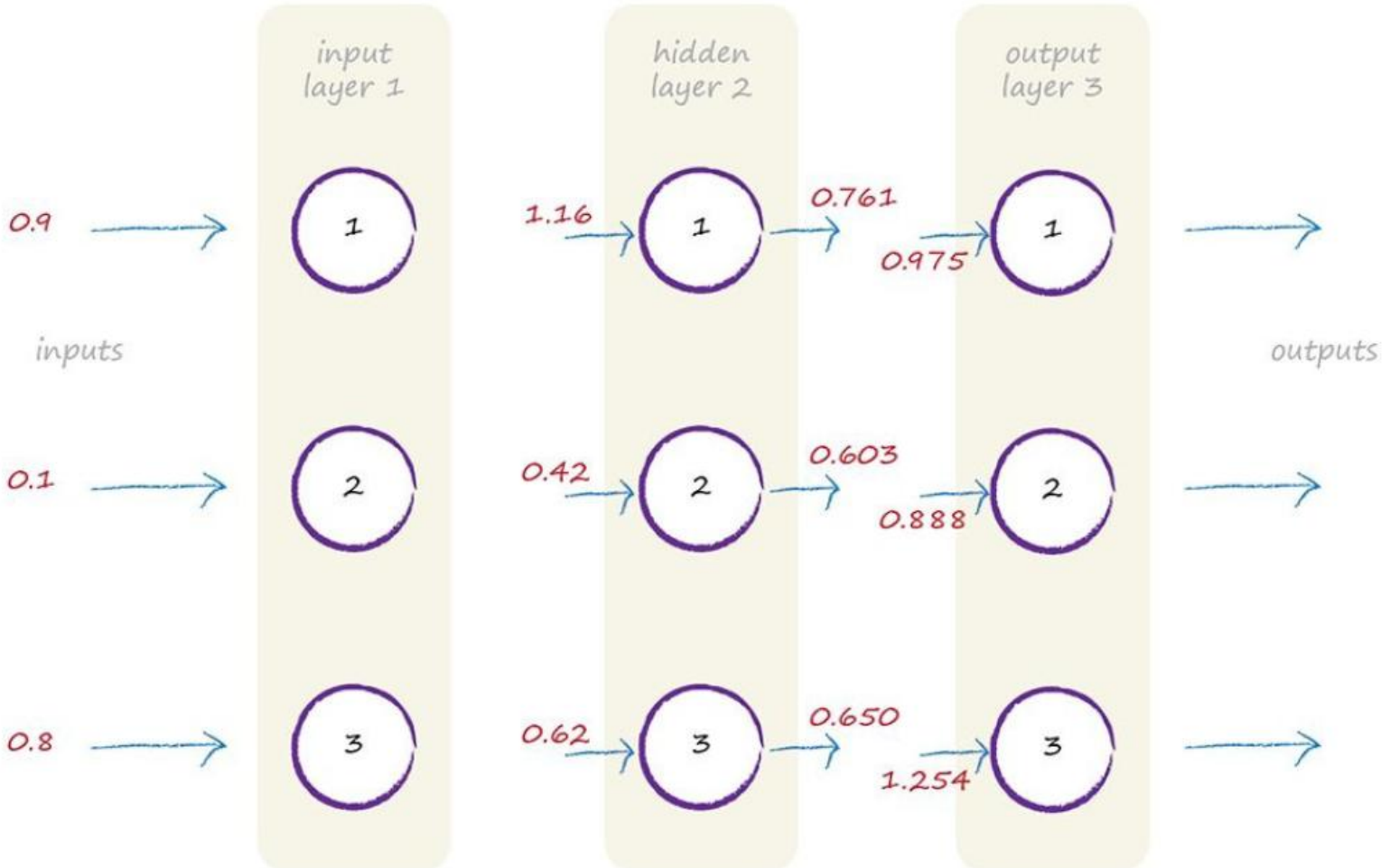
$$O_{hidden} = \begin{pmatrix} 0.761 \\ 0.603 \\ 0.650 \end{pmatrix}$$



$$X_{output} = W_{hidden\_output} \cdot O_{hidden}$$

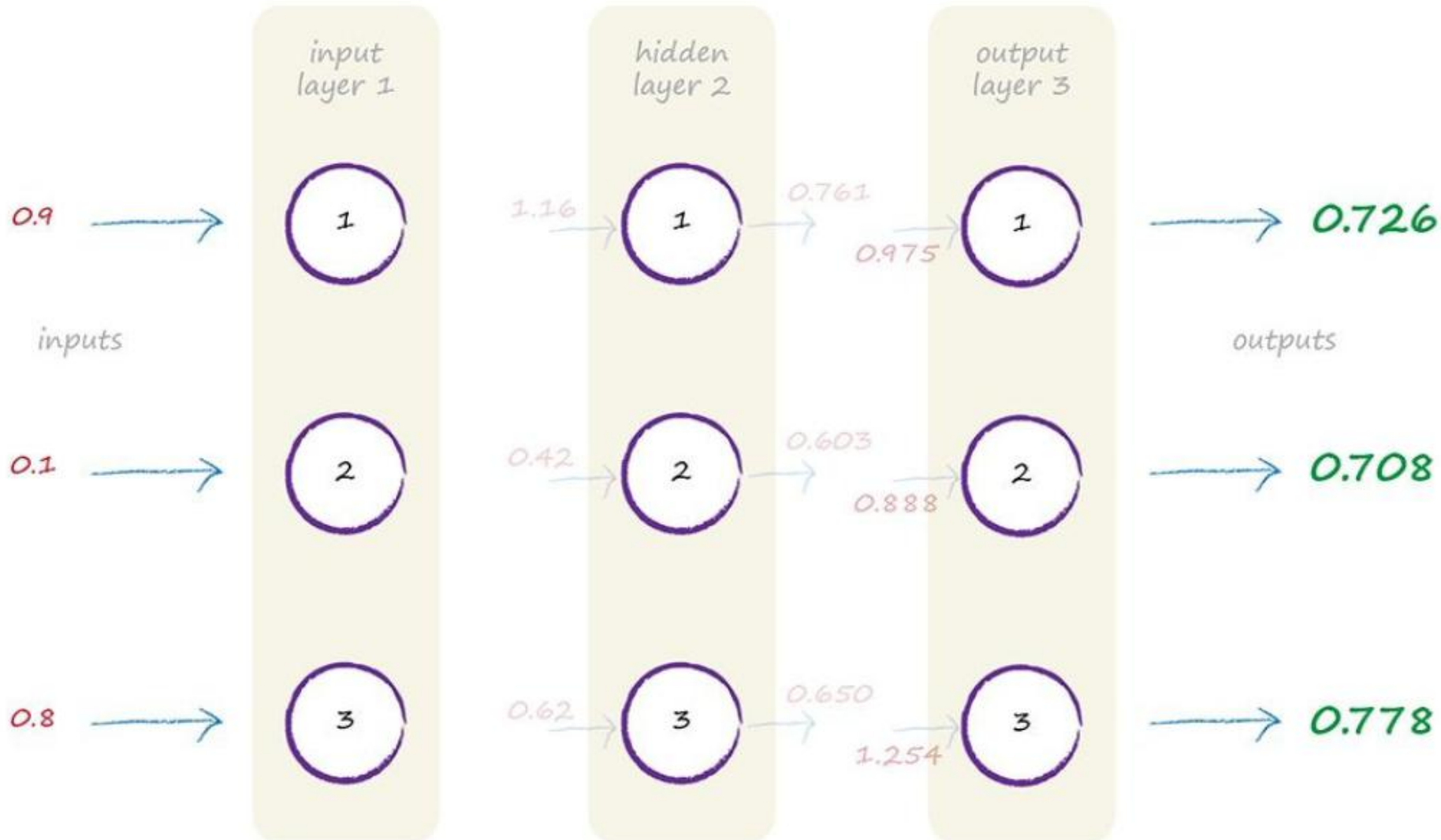
$$X_{output} = \begin{pmatrix} 0.3 & 0.7 & 0.5 \\ 0.6 & 0.5 & 0.2 \\ 0.8 & 0.1 & 0.9 \end{pmatrix} \cdot \begin{pmatrix} 0.761 \\ 0.603 \\ 0.650 \end{pmatrix}$$

$$X_{output} = \begin{pmatrix} 0.975 \\ 0.888 \\ 1.254 \end{pmatrix}$$



$$O_{output} = signmoid \begin{pmatrix} 0.975 \\ 0.888 \\ 1.254 \end{pmatrix}$$

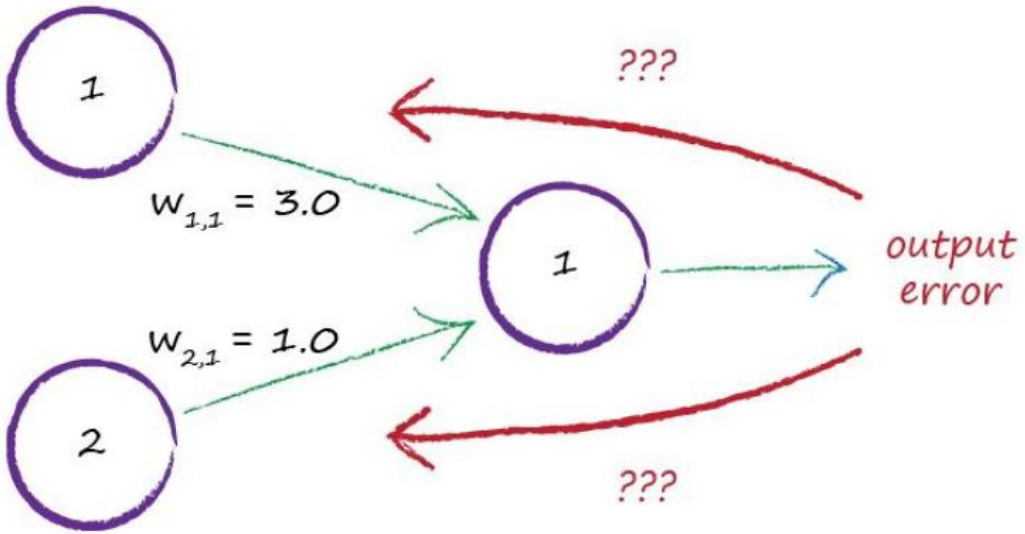
$$O_{output} = \begin{pmatrix} 0.726 \\ 0.708 \\ 0.778 \end{pmatrix}$$


input
layer 1
hidden
layer 2
output
layer 3
0.9
0.1
0.8
inputs
1
2
3
1.16
0.42
0.62
0.761
0.603
0.650
0.975
0.888
1.254
outputs
0.726
0.708
0.778

8. Learning Weights From More Than One Node
（从多个节点学习权重）

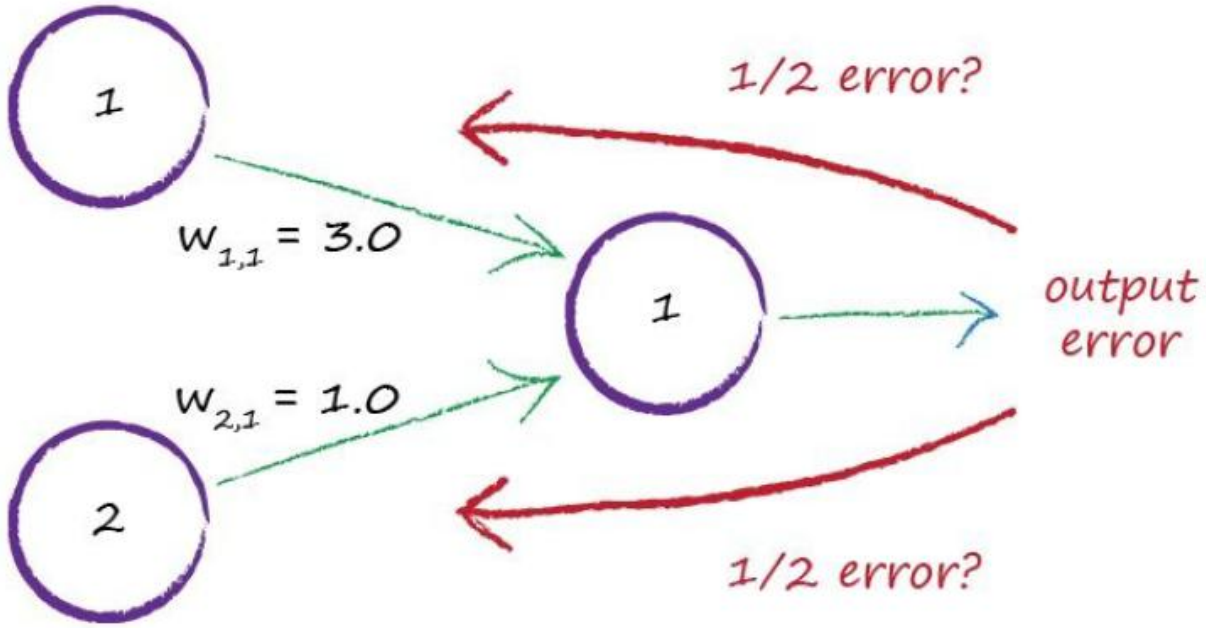
之前我们通过调整节点线性函数的斜率参数来细化一个简单的线性分类器。
我们使用误差，即节点产生的答案与我们知道的答案之间的差异，来指导改进。
当多个节点对输出及其误差有贡献时，我们如何更新链接权重？

使用所有误差仅更新一个权重是没有意义的，因为这会忽略另一个链接及其权重。
之所以出现该误差，是因为导致该误差的链接不止一个。



8.Learning Weights From More Than One Node
（从多个节点学习权重）

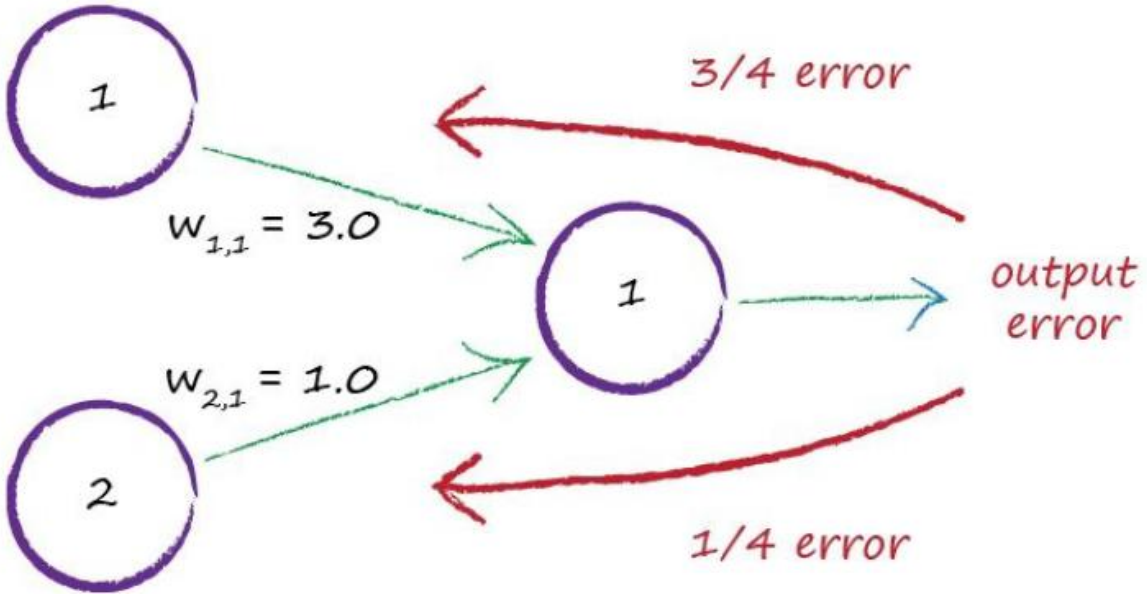
一种想法是在所有贡献节点之间平均分配错误，如下所示。这根本不是一个坏主意。虽然还没有在真正的神经网络中尝试过，但我相信它的效果不会太差。



8.Learning Weights From More Than One Node（从多个节点学习权重）

另一个想法是分割误差但不是平等地做。相反，我们为那些具有更大链接权重的连接提供了更多的误差。为什么？因为他们对误差的贡献更大。

下图说明了这个想法。这里有两个节点向输出节点提供信号。链接权重为 3.0 和 1.0。如果我们以与这些权重成比例的方式分割误差，我们可以看到输出误差的 3/4 应该用于更新第一个较大的权重，而 1/4 的误差用于更新第二个较小的权重。



8.Learning Weights From More Than One Node
（从多个节点学习权重）

我们可以将同样的想法扩展到更多节点。 按每个链接对误差的贡献比例分配错误，表示由链接的权重大小决定。

我们以两种方式使用权重。
首先，我们使用权重将信号从神经网络中的输入层向前传播到输出层（前向传播）。
其次，我们使用权重将误差从输出反向传播回网络（反向传播）。

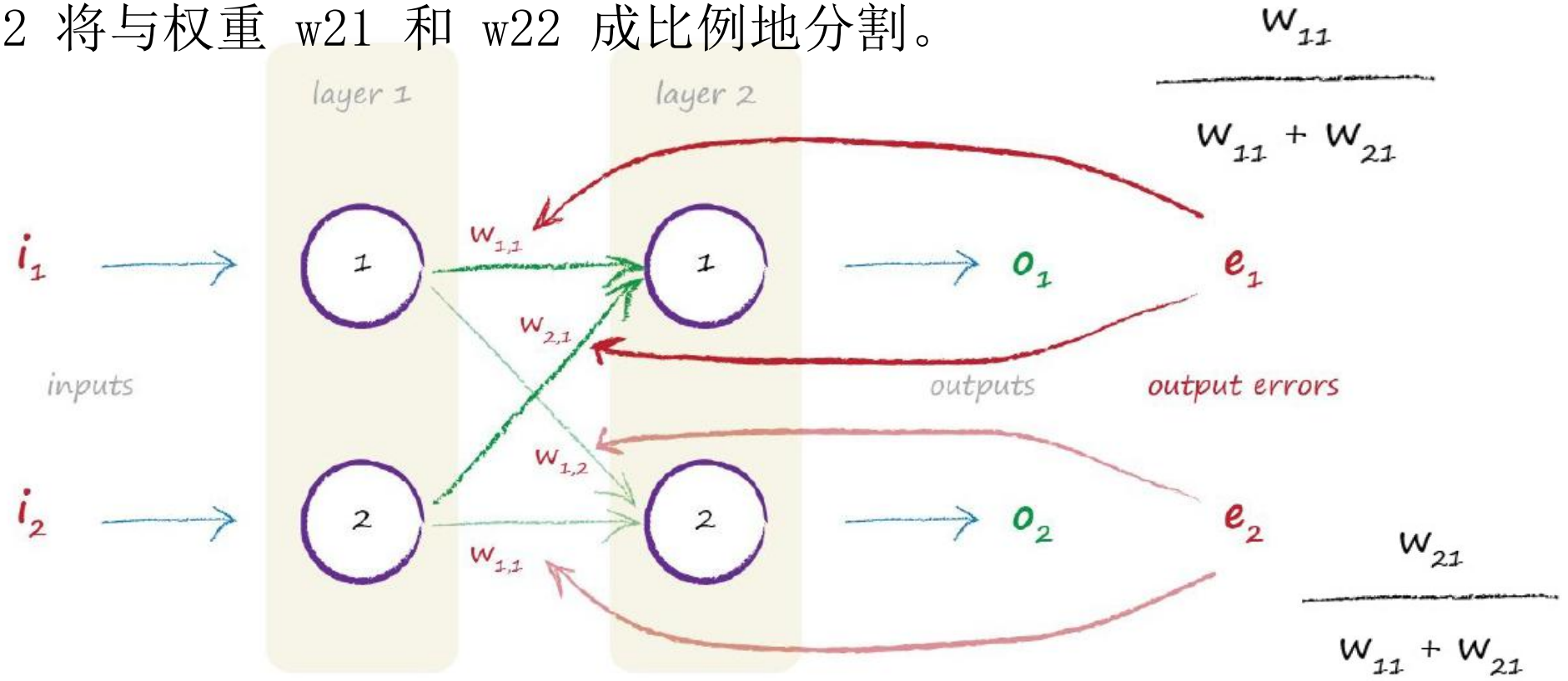
如果输出层有 2 个节点，我们会对第二个输出节点做同样的事情。
让我们接下来看看这个，来自更多输出节点的反向误差传播。

9.Backpropagating（多输出节点的反向传播）

两个输出节点都可能有误差，这两个错误都需要通过网络中内部链路权重细化。我们可以使用与以前相同的方法，将输出节点的误差以与它们的权重成比例的方式分配给贡献链接。我们有多个输出节点这一事实并没有真正改变任何东西，链接之间没有依赖关系。

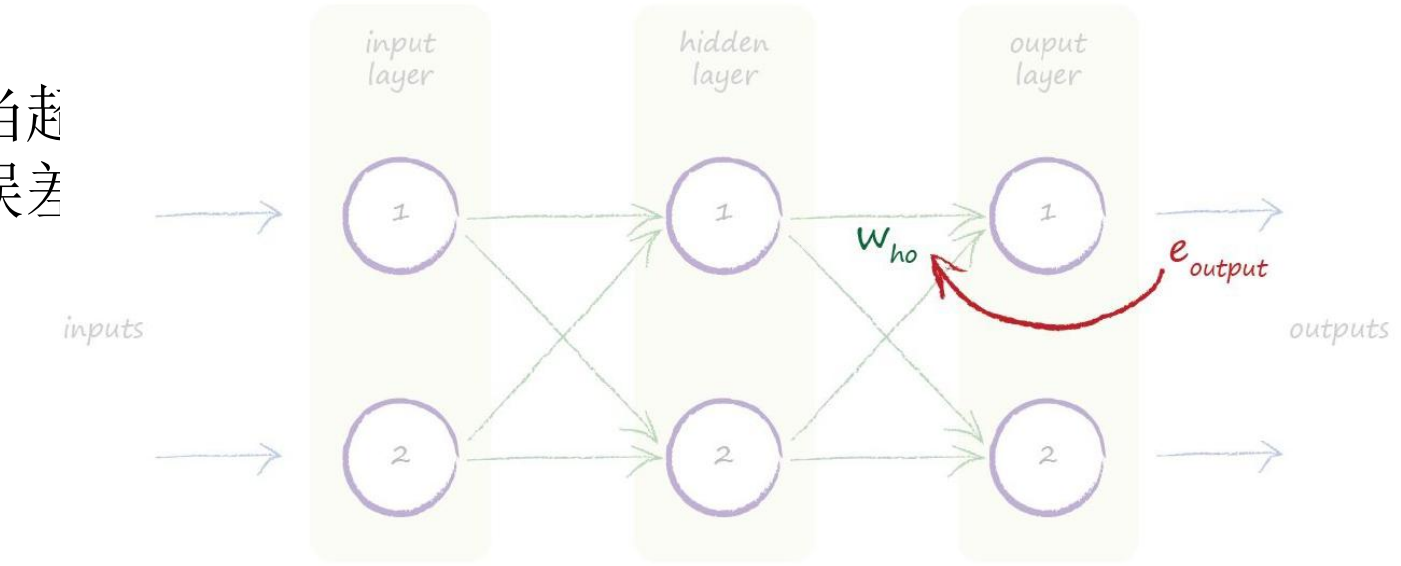
图中可以看出，误差 e1 与权重为 w11 和 w21 的连接链路成比例分割。

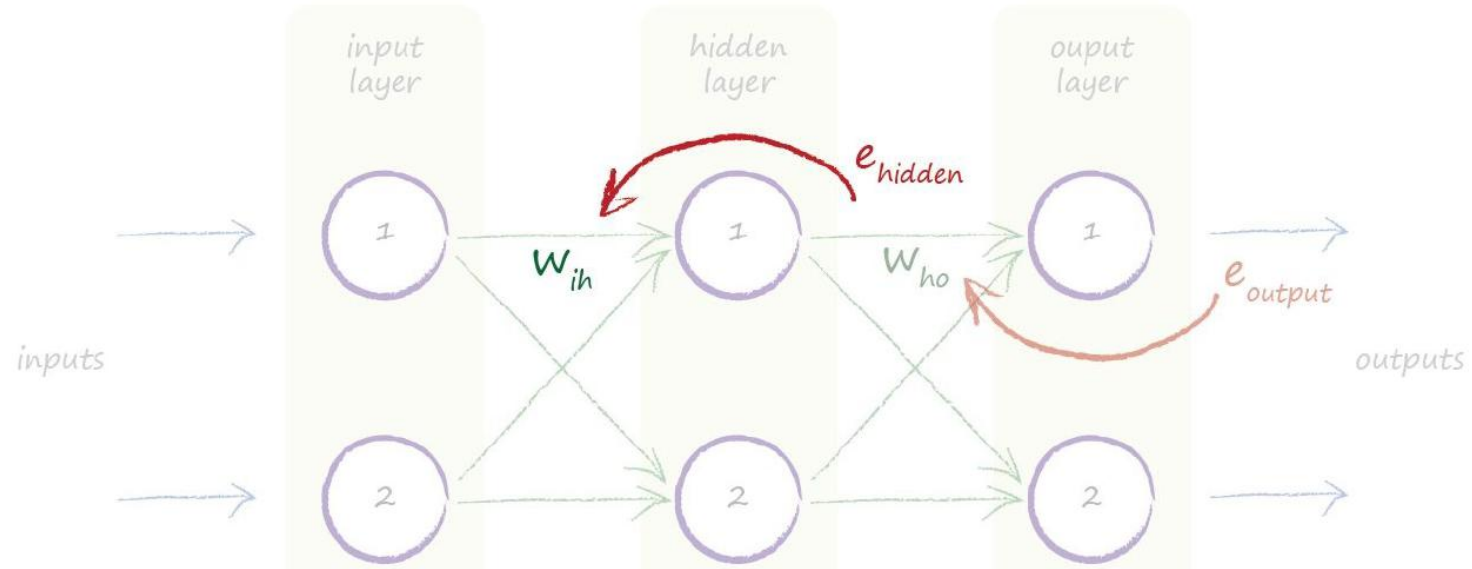
类似地，e2 将与权重 w21 和 w22 成比例地分割。



10. Backpropagating Errors To More Layers（多层的反向传播）

下一个问题是当超 层更远的层中的链接权重？将误差

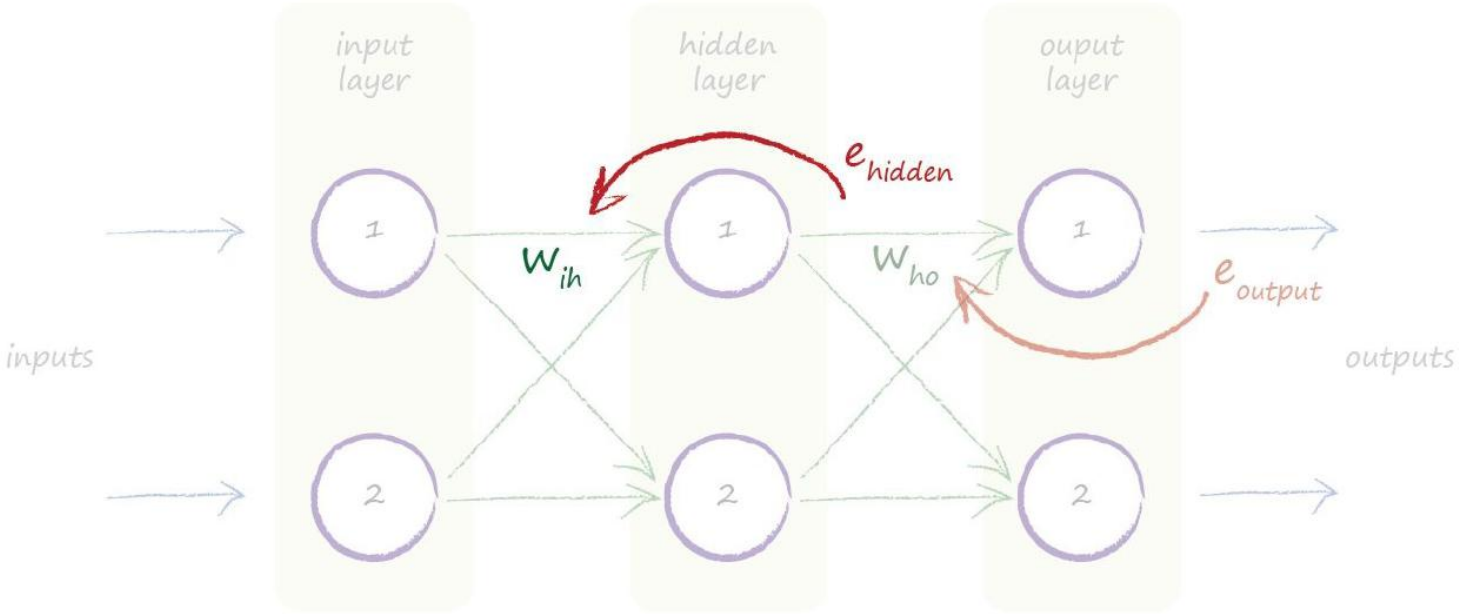




如果有更多层，我们会重复将相同的想法应用于从最终输出层向后工作的每一层。

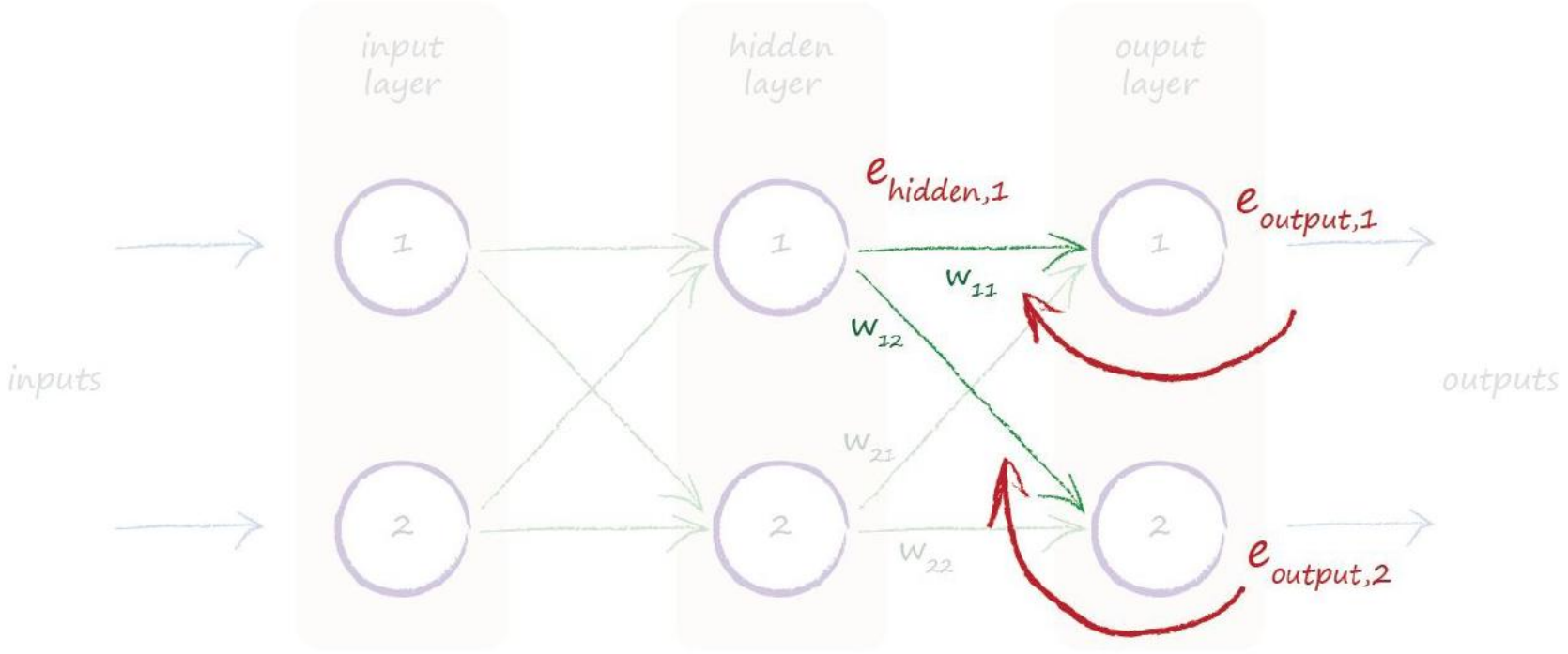
10. Backpropagating Errors To More Layers（多层的反向传播）

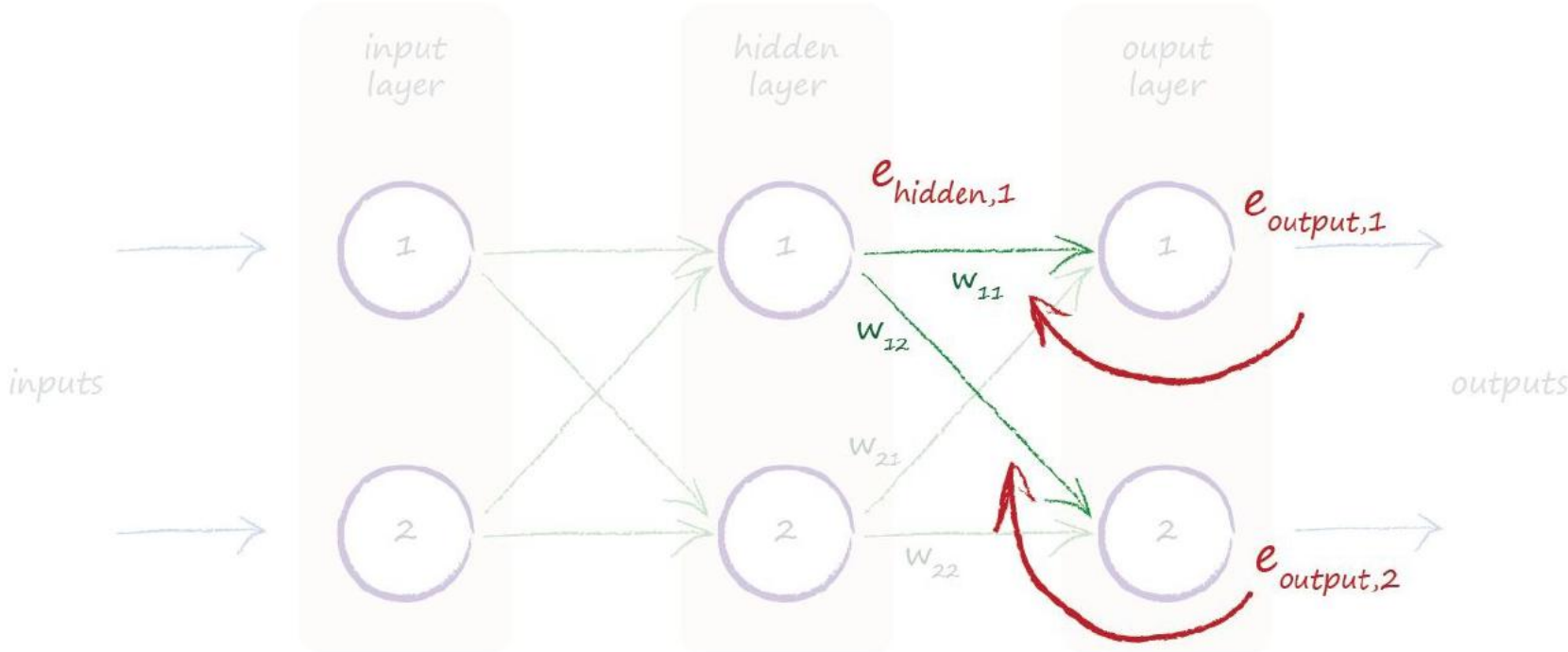
如果我们首先在输出层节点 e_{output} 的输出中使用误差，那么我们对隐藏层节点 e_{hidden} 使用什么误差？这是一个很好的问题，因为中间隐藏层中的节点没有明显的误差。我们通过前馈输入信号知道，是的，隐藏层中的每个节点确实有一个输出。
您会记得这是应用于该节点输入的加权总和的激活函数。但是我们如何解决这个误差呢？我们没有隐藏节点的目标或所需的输出。我们只有最终输出层节点的目标值，
这些来自训练示例。



10. Backpropagating Errors To More Layers（多层的反向传播）

让我们再次看一下图以获得灵感！隐藏层中的第一个节点有两条链路，将其连接到两个输出层节点。我们知道我们可以沿着这些链接中的每一个分割输出误差，这意味着对于从这个中间层节点出现的两个链接中的每一个，都有某种误差。我们可以重新组合这两个链接误差来形成这个节点的误差作为第二个最佳方法，因为我们实际上没有中间层节点的目标值。

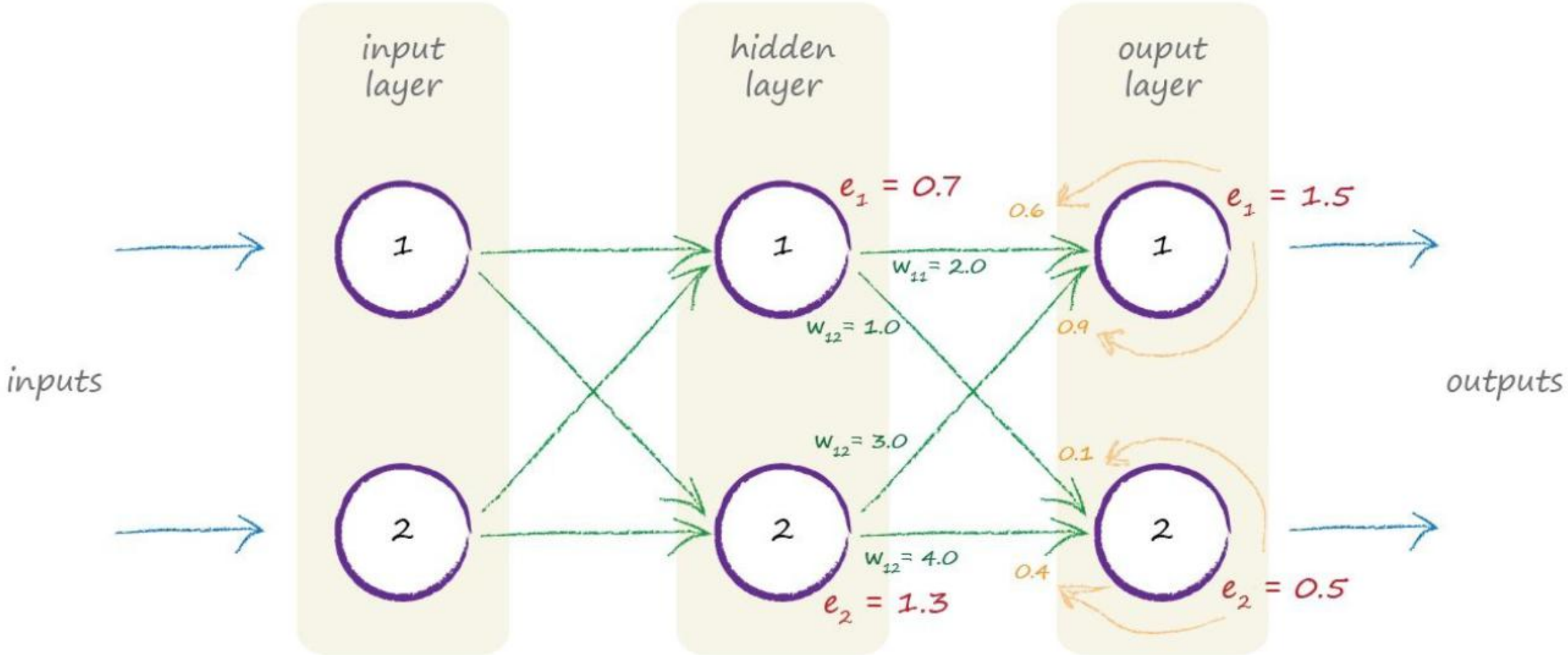




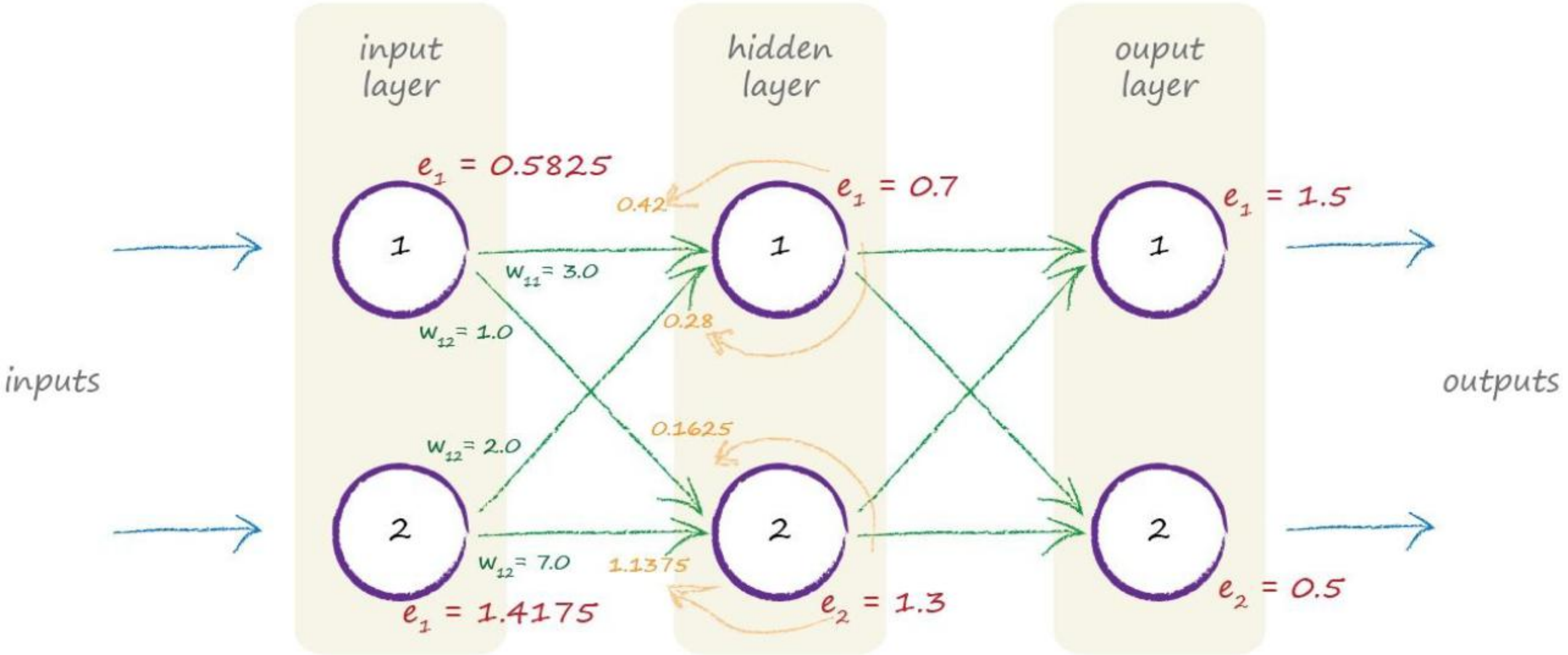
$e_{hidden,1}$ = sum of split errors on links w_{11} and w_{12}

$$= e_{output,1} * \frac{w_{11}}{w_{11} + w_{21}} + e_{output,2} * \frac{w_{12}}{w_{12} + w_{22}}$$

将误差传播回具有实际数字的简单 3 层网络。



将误差传播回具有实际数字的简单 3 层网络。



11.Backpropagating Errors with Matrix Multiplication（使用矩阵乘法反向传播误差）

$$error_{output} = \begin{pmatrix} e_1 \\ e_2 \end{pmatrix} \qquad error_{hidden} = \begin{pmatrix} (e_1 * w_{11}) + (e_2 * w_{12}) \\ (e_1 * w_{21}) + (e_2 * w_{22}) \end{pmatrix}$$

$$error_{hidden} = \begin{pmatrix} w_{11} & w_{12} \\ w_{21} & w_{22} \end{pmatrix} \cdot \begin{pmatrix} e_1 \\ e_2 \end{pmatrix}$$

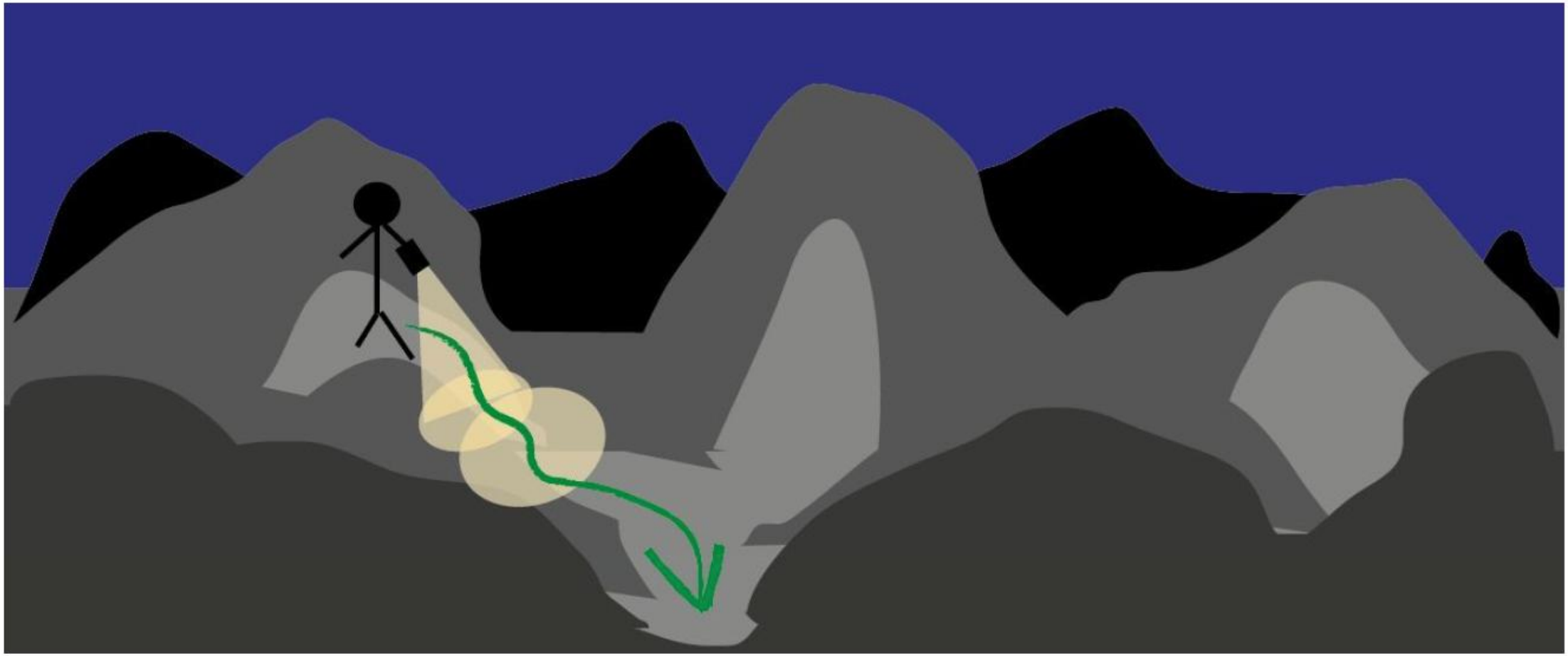
$$error_{hidden} = w^{T}_{hidden\_output} \cdot error_{output}$$

12. How Do We Actually Update Weights?
（我们实际如何更新权重？ ）

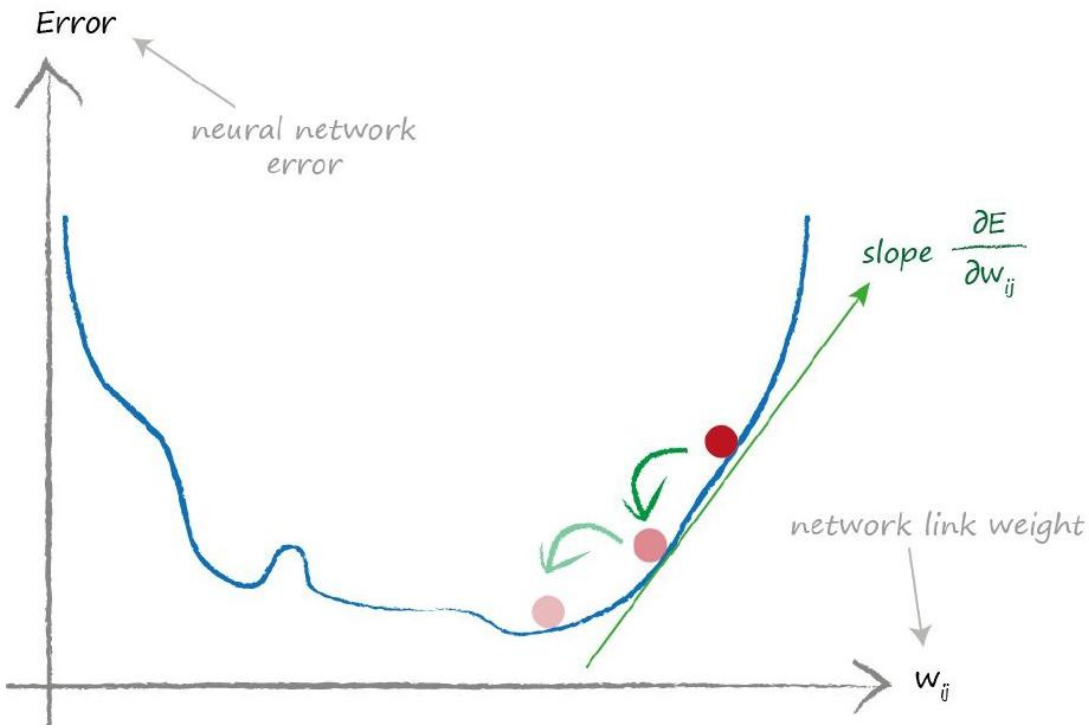
有太多权重的组合，及太多函数的函数的函数的组合。

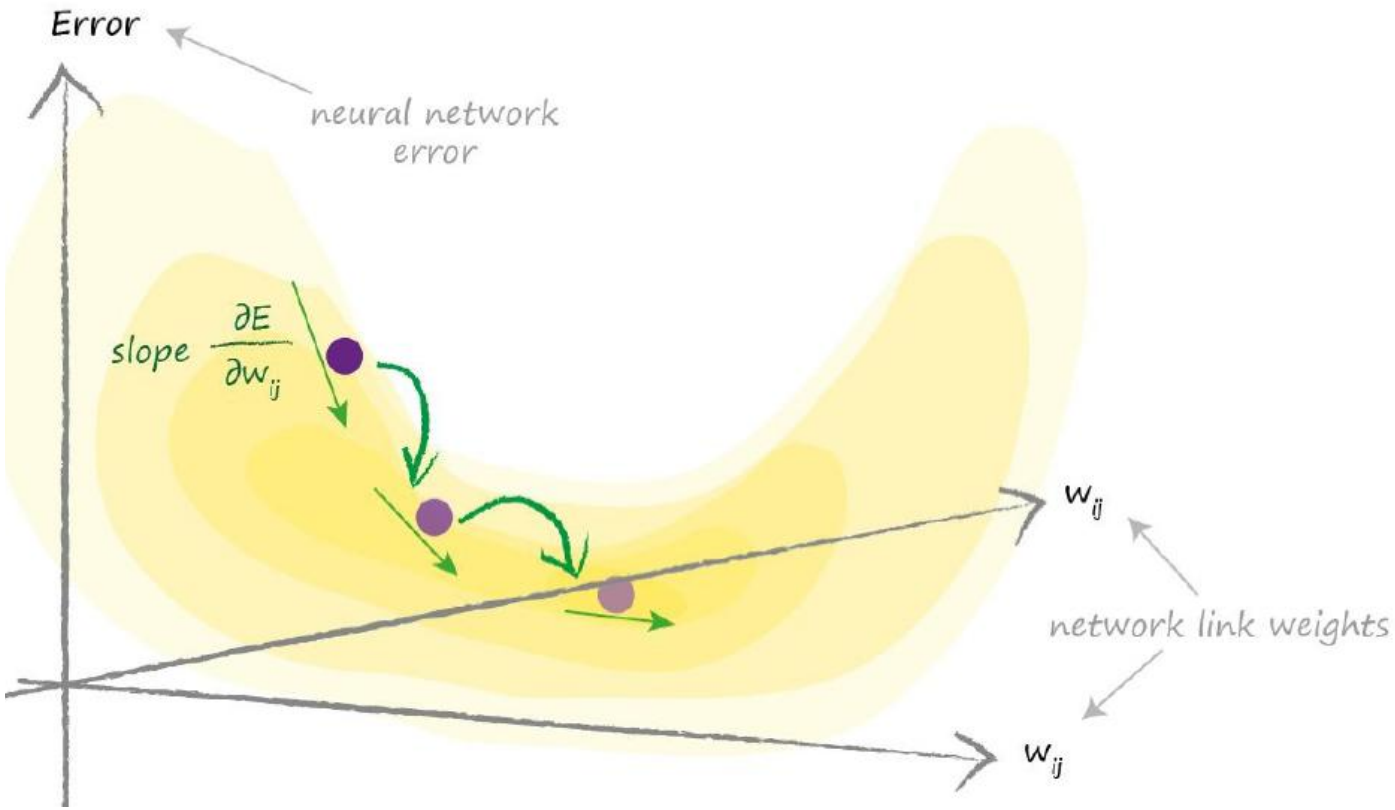
$$o_k = \frac{1}{1 + e^{-\sum_{j=1}^{3}\left(w_{j,k} \cdot \frac{1}{1+e^{-\sum_{i=1}^{3}(w_{i,j} \cdot x_i)}}\right)}}$$

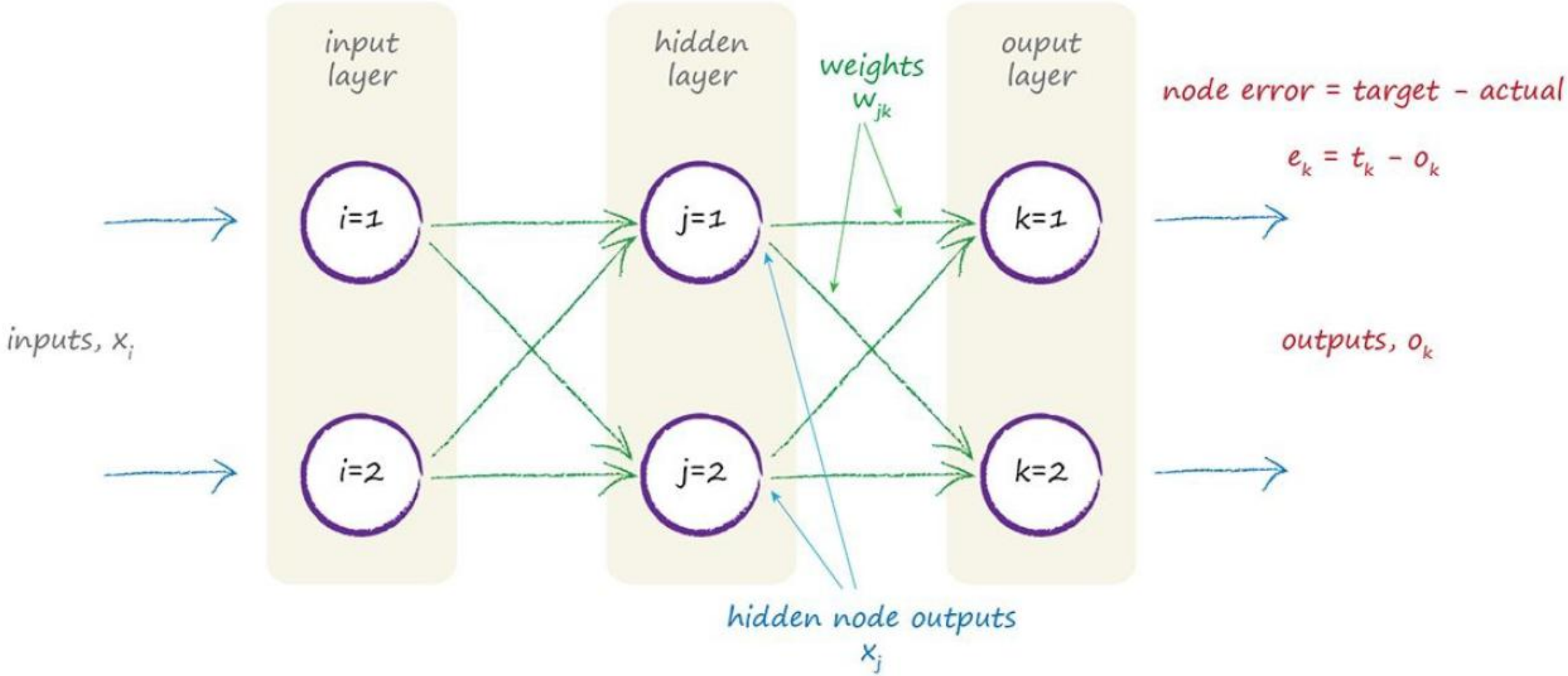
13. gradient descent（梯度下降）



为了做梯度下降，需要解出误差函数关于权重的梯度，这要求微积分。







$$\frac{\partial E}{\partial w_{jk}} = \frac{\partial}{\partial w_{jk}} \Sigma_n (t_n - o_n)^2$$

$$\frac{\partial E}{\partial w_{jk}} = \frac{\partial}{\partial w_{jk}} (t_k - o_k)^2$$

chain rule（链式法则）

$$\frac{\partial E}{\partial w_{jk}} = \frac{\partial E}{\partial o_k} \cdot \frac{\partial o_k}{\partial w_{jk}}$$

$$\frac{\partial E}{\partial w_{jk}} = -2(t_k - o_k) \cdot \frac{\partial o_k}{\partial w_{jk}}$$

$$\frac{\partial E}{\partial w_{jk}} = -2(t_k - o_k) \cdot \frac{\partial}{\partial w_{jk}} \text{sigmoid}(\Sigma_j w_{jk} \cdot o_j)$$

$$\frac{\partial}{\partial x} \text{sigmoid}(x) = \text{sigmoid}(x) \left(1 - \text{sigmoid}(x)\right)$$

$$\frac{\partial E}{\partial w_{jk}} = -2(t_k - o_k) \cdot \text{sigmoid}(\Sigma_j w_{jk} \cdot o_j)(1 - \text{sigmoid}(\Sigma_j w_{jk} \cdot o_j)) \cdot \frac{\partial}{\partial w_{jk}} (\Sigma_j w_{jk} \cdot o_j)$$

$$= -2(t_k - o_k) \cdot \text{sigmoid}(\Sigma_j w_{jk} \cdot o_j)(1 - \text{sigmoid}(\Sigma_j w_{jk} \cdot o_j)) \cdot o_j$$

$$\frac{\partial E}{\partial w_{jk}} = -(t_k - o_k) \cdot sigmoid(\Sigma_j w_{jk} \cdot o_j)(1 - sigmoid(\Sigma_j w_{jk} \cdot o_j)) \cdot o_j$$

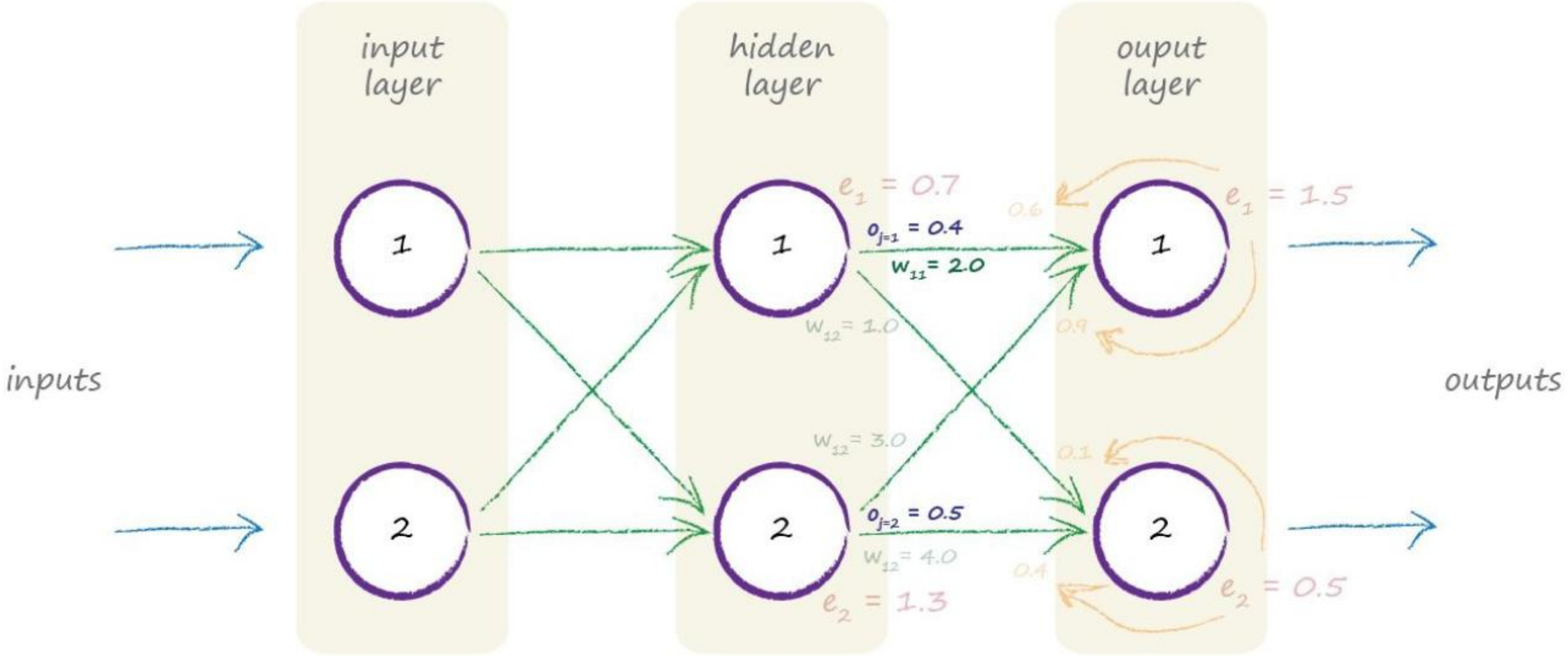
$$new\ w_{jk} = old\ w_{jk} - \alpha \cdot \frac{\partial E}{\partial w_{jk}}$$

$$\begin{pmatrix} \Delta w_{1,1} & \Delta w_{2,1} & \Delta w_{3,1} & \dots \\ \Delta w_{1,2} & \Delta w_{2,2} & \Delta w_{3,2} & \dots \\ \Delta w_{1,3} & \Delta w_{2,3} & \Delta w_{j,k} & \dots \\ \dots & \dots & \dots & \dots \end{pmatrix} = \begin{pmatrix} E_1 * S_1(1-S_1) \\ E_2 * S_2(1-S_3) \\ E_k * S_k(1-S_k) \\ \dots \end{pmatrix} \cdot \begin{pmatrix} O_1 & O_2 & O_j & \dots \end{pmatrix}$$

$$\Delta w_{jk} = \alpha * E_k * sigmoid(O_k) * (1 - sigmoid(O_k)) \cdot O_j^T$$

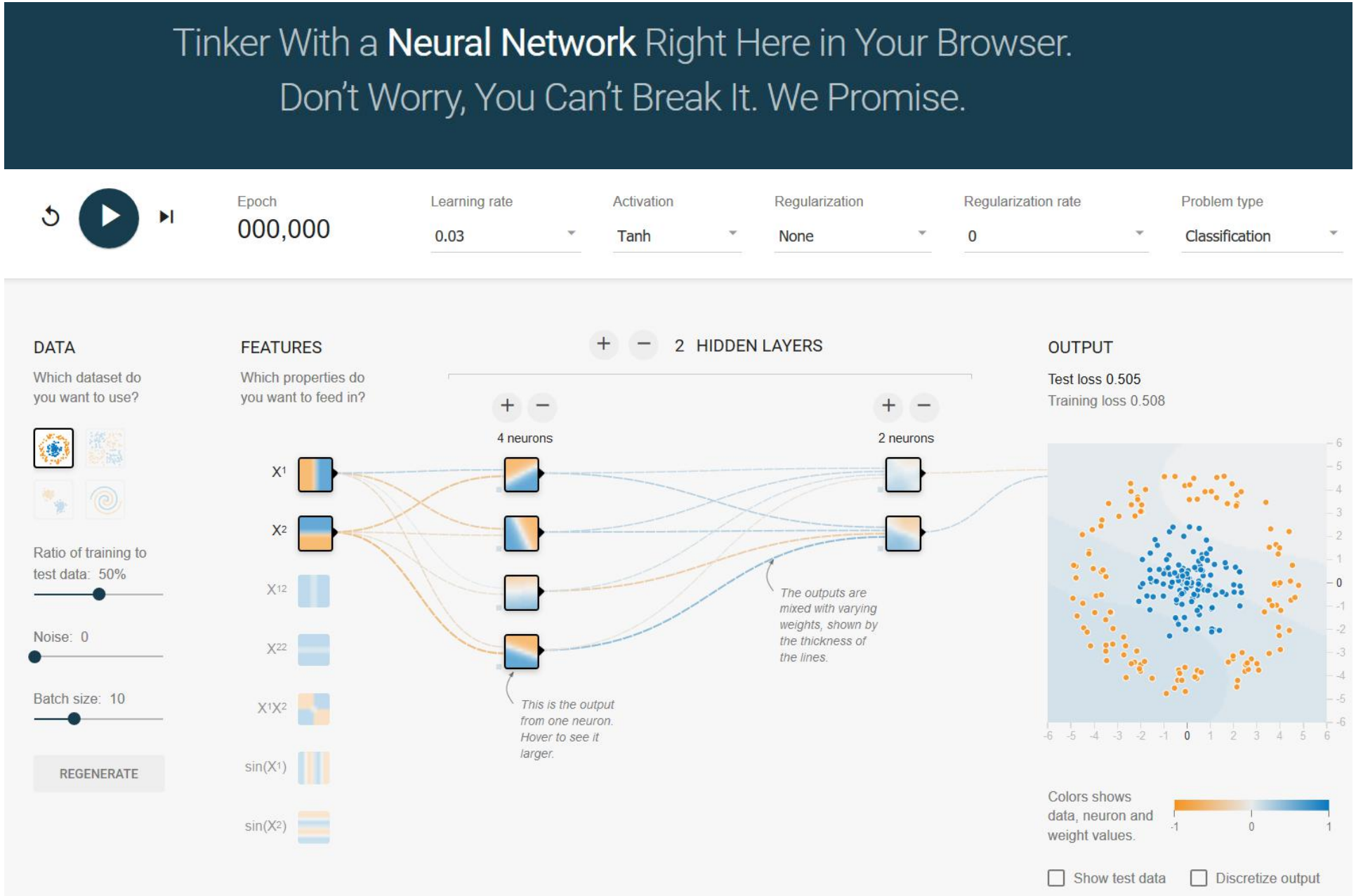
us layer

Weight Update Worked Example

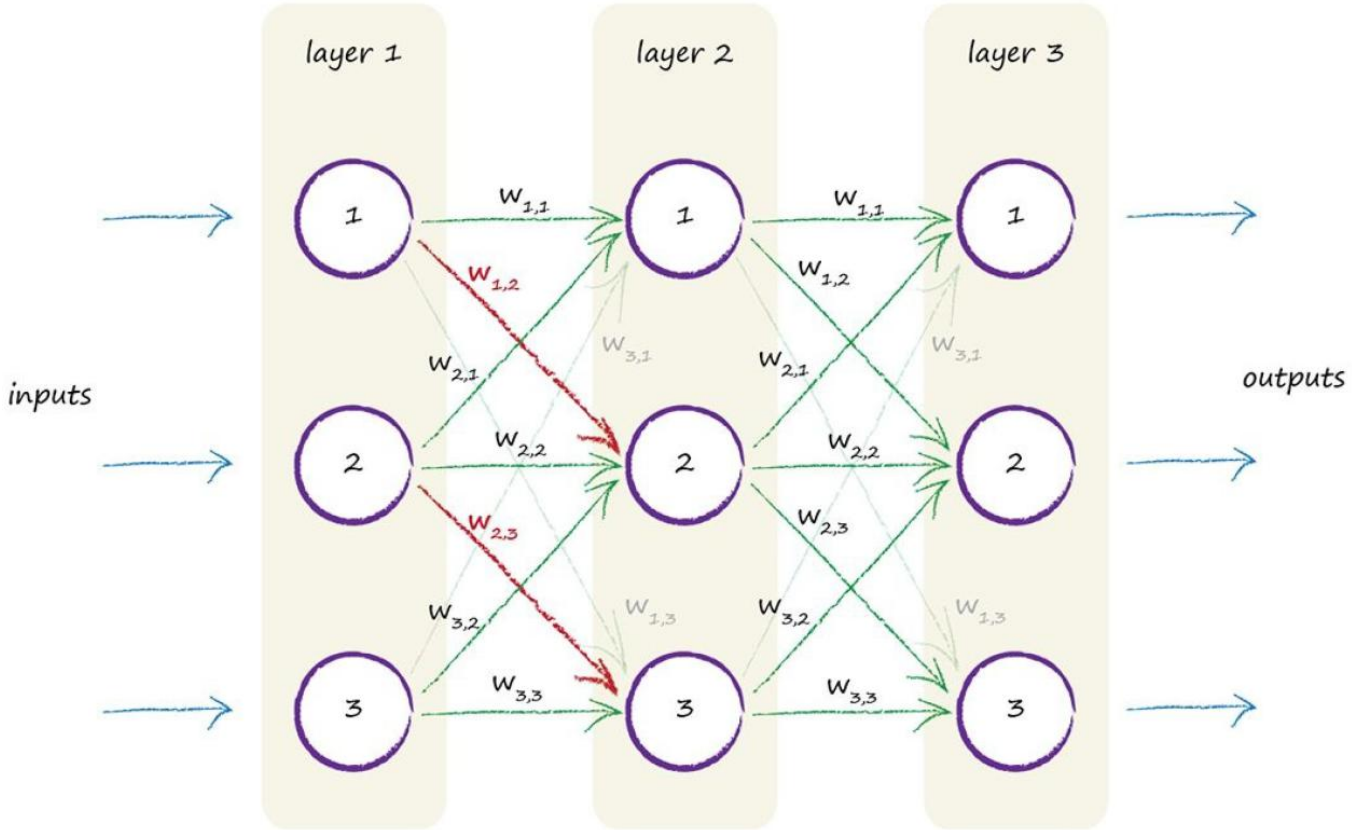


$$\frac{\partial E}{\partial w_{jk}} = -(t_k - o_k) \cdot \text{sigmoid}(\Sigma_j w_{jk} \cdot o_j)(1 - \text{sigmoid}(\Sigma_j w_{jk} \cdot o_j)) \cdot o_j$$

4. 构建神经网络


Tinker With a Neural Network Right Here in Your Browser.
Don't Worry, You Can't Break It. We Promise.
Epoch
000,000
Learning rate
0.03
Activation
Tanh
Regularization
None
Regularization rate
0
Problem type
Classification
DATA
Which dataset do you want to use?
Ratio of training to test data: 50%
Noise: 0
Batch size: 10
REGENERATE
FEATURES
Which properties do you want to feed in?
X1
X2
X12
X22
X1X2
sin(X1)
sin(X2)
2 HIDDEN LAYERS
4 neurons
2 neurons
The outputs are mixed with varying weights, shown by the thickness of the lines.
This is the output from one neuron. Hover to see it larger.
OUTPUT
Test loss 0.505
Training loss 0.508
Colors shows data, neuron and weight values.
Show test data
Discretize output

例子：手动构建



Pdf-part2_neural_network
Linear层调用

machine learning library

TensorFlow

theano

Caffe

Microsoft
CNTK

K Keras
A deep learning library

mxnet

Deep Learning library produced by Amazon
DSSTNE

scikits
learn
machine learning in Python


Thanks!
Obrigado!
Gracias!
谢谢！
Dziękuję!
Danke!
Grazie!
Merci!
شكرا لك !
감사합니다!
Kiitos!