机器学习: Part 3

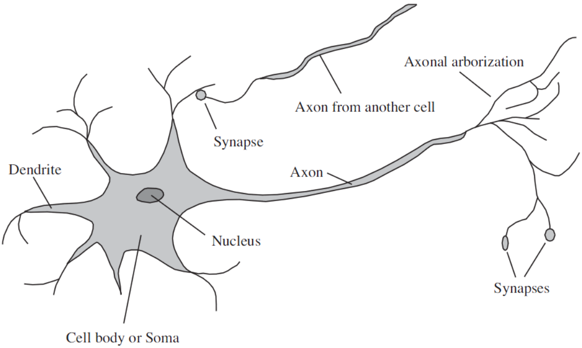
- 线性和逻辑回归
- 神经网络的反向传播算法

Slides based on those of Pascal Poupart

人脑

- 人脑是思维的器官，是智能的物质基础
- 人脑由称为神经元的神经细胞组成
- 成人的大脑中估计有1000 亿个神经元

神经元(Neuron)



- 一个神经元通常具有多个树突(dendrite)，主要用来接受传入信息
- 而轴突(axon)只有一条，轴突尾端有许多轴突末梢可以给其他多个神经元传递信息. 轴突末梢跟其他神经元的树突产生连接的位置叫做“突触”(synapse)

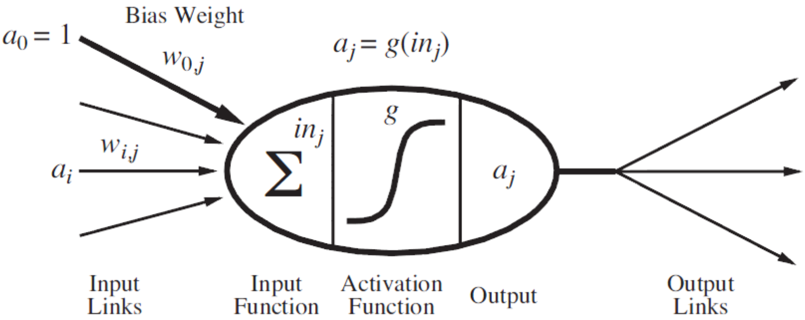
神经元的工作原理

- 神经元的树突在接收到特定的输入刺激后，其胞体就会被激活，并通过轴突向其它神经元输出兴奋，从而导致更多的神经元被激活。
- 神经元有两种状态：静息态和激活状态。神经元由静息态切换为激活状态，是因为其接受了来自其它神经元的输入，并达到或超过了必须达到的阈值。

人工神经网络(ANNs)

- 模拟大脑进行计算
- 组成:
  - 节点(也称单元)对应于神经元
  - 连接对应于突触
- 计算:
  - 节点间传送的数值信号对应于神经元之间的化学信号
  - 节点对数值信号进行修改对应于神经元激活率

神经元的一个简单数学模型

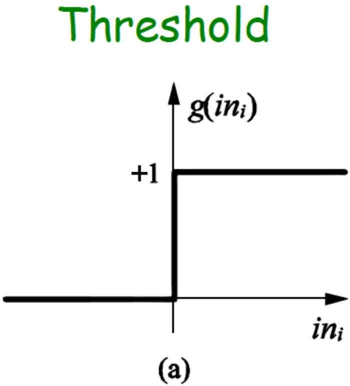


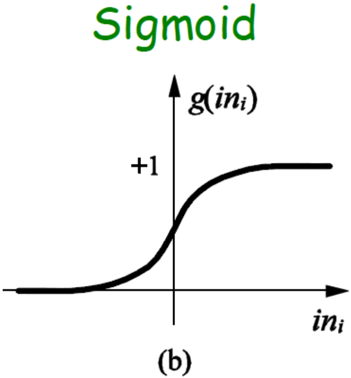
神经元“激活”当输入的线性组合超过某个阈值

激活函数(Activation Function)

- 应该是非线性的
  - 否则网络仅仅表示一个线性函数
- 用以模拟神经元的激活
  - 对于“正确的”输入，单元应该是“激活的”：输出接近1
  - 对于“错误的”输入，单元应该是“静息的”：输出接近0

常用激活函数

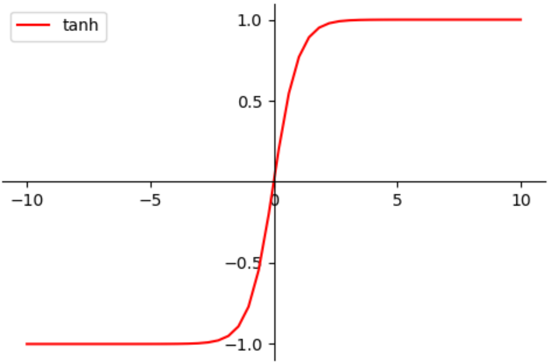




$$g(x) = 1/(1+e^{-x})$$

另一个激活函数

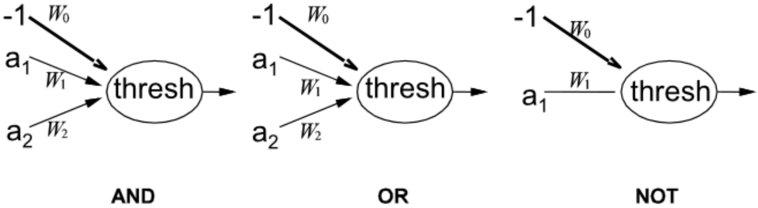
双曲正切函数(tanh)



$$tanh(x) = \frac{e^x - e^{-x}}{e^x + e^{-x}} = \frac{2}{1 + e^{-2x}} - 1 = 2g(2x) - 1$$

逻辑门

- McCulloch and Pitts(1943)设计ANNs表示布尔函数
- 为了表示与，或，非，以下单元的权重应该如何取值？

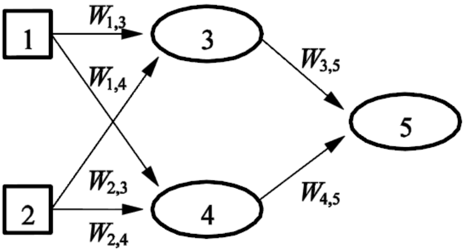


网络结构

- 前馈网络(Feed-forward network)
  - 有向非循环图
  - 没有内部状态
  - 简单地从输入计算输出
- 循环网络(Recurrent network)
  - 有向循环图
  - 具有内部状态的动态系统
  - 可以记忆信息

An example

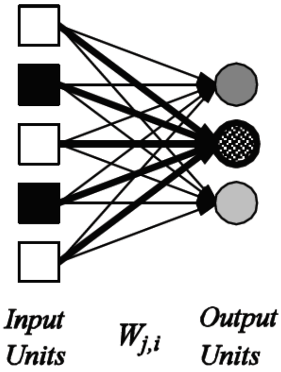
Simple network with two inputs, one hidden layer of two units, one output unit



$$a_5 = g(W_{3,5}a_3 + W_{4,5}a_4)$$
$$= g(W_{3,5}g(W_{1,3}a_1 + W_{2,3}a_2) + W_{4,5}g(W_{1,4}a_1 + W_{2,4}a_2))$$

感知器(Perceptron)

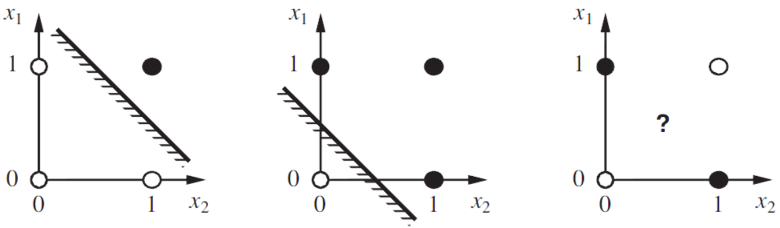
Single layer feed-forward network



阈值感知器假设空间

- 假设空间h_w: 具有参数w 的所有二元分类s.t. $w \cdot x \geq 0 \rightarrow 1$, $w \cdot x < 0 \rightarrow 0$
- 由于$w \cdot x$ 关于x 是线性的，感知器被称为线性分离器(linear separator)

所有布尔门都是线性可分的吗?



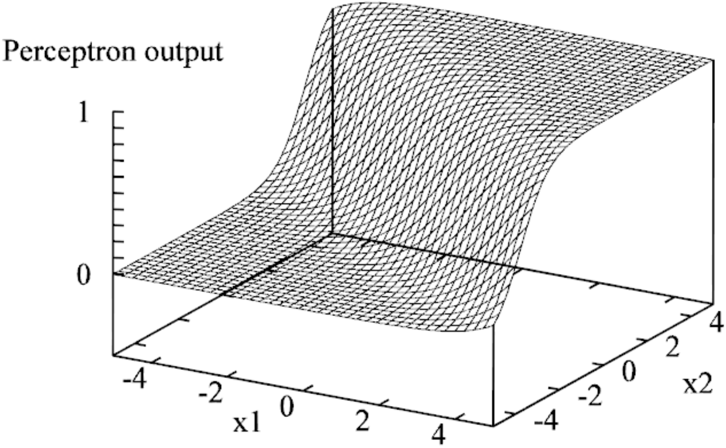
(a) x_1 **and** x_2　(b) x_1 **or** x_2　(c) x_1 **xor** x_2

阈值感知器中的线性可分性

- 黑色的点表示输入空间中函数值为1的点，白色的点表示输入空间中函数值为0的点
- 感知器在直线的非阴影部分返回1
- 在(c)中，不存在对输入正确分类的直线

Sigmoid感知机

表示“软的”线性分离器



损失函数

- 损失函数$L(x, y, y')$定义为当正确的答案是$f(x) = y$，预测$h(x) = y'$的效用损失量
- 通常使用一个简单的版本$L(y, y')$, 它独立于x
- 三个常用的损失函数:
    - 绝对值损失: $L_1(y, y') = |y - y'|$
    - 平方误差损失: $L_2(y, y') = (y - y')^2$
    - 0/1损失: $L_{0/1}(y, y') = 0$ if $y = y'$, else 1
- 令E为样例集. 总损失$L(E) = \sum_{e \in E} L(e)$

线性回归

- 用线性函数做回归
- 通过在权重空间中做优化搜索
- 使用梯度下降(gradient descent)
- 求解函数最小值的一种迭代方法
- 对权重做任意初始化
- 在每一步，与函数偏导成比例减少每个权重

$$w_i \leftarrow w_i - \alpha \partial Loss(w) / \partial w_i$$

- α 称为学习速率(learning rate)

线性回归

- $h_w(x) = w \cdot x = \sum_i w_i x_i$
- 平方误差损失: $Loss(w) = (y - h_w(x))^2$
- 求导的链式法则: $\partial g(f(x))/\partial x = g'(f(x))\partial f(x)/\partial x$
- $\partial Loss(w)/\partial w_i = -2(y - h_w(x))x_i$
- $w_i \leftarrow w_i + \alpha(y - h_w(x))x_i$

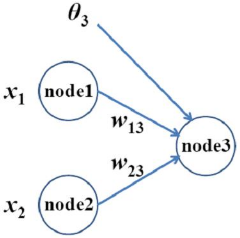
逻辑回归(Logistic regression)

- 逻辑函数是线性函数的sigmoid函数
- 逻辑回归：用逻辑函数做回归
- $g(x) = 1/(1 + e^{-x})$
- $h_w(x) = g(w \cdot x)$
- $g' = g(1 - g)$
- $Loss(w) = (y - h_w(x))^2$
- $\partial Loss(w)/\partial w_i = -2(y - h_w(x))g'(w \cdot x)x_i$
  $= -2(y - h_w(x))h_w(x)(1 - h_w(x))x_i$
- $w_i \leftarrow w_i + \alpha(y - h_w(x))h_w(x)(1 - h_w(x))x_i$

The algorithm

initialize w arbitrarily
repeat
    for each e in examples do
        $p \leftarrow g(w \cdot x(e))$
        $\delta \leftarrow y(e) - p$
        for each i do
            $w_i \leftarrow w_i + \alpha \delta p(1-p) x_i$
until some stopping criterion is satisfied
return w

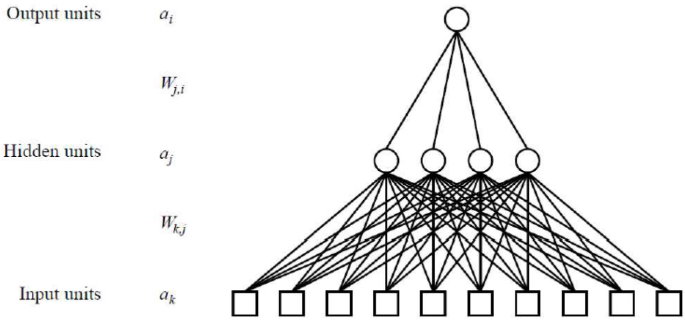
An exercise



- Node 3 use the tanh function as the activate function.
- $x_1 = 1$, $x_2 = 0.5$, $y = 1$, $\theta_3 = 0$, $w_{13} = 1$, $w_{23} = -1$
- $Loss = 0.5(y - a_3)^2$
- Compute $\partial Loss/\partial \theta_3$
- Note $tanh(x) = (e^x - e^{-x})/(e^x + e^{-x}) = 2g(2x) - 1$, and $tanh'(x) = 1 - tanh^2(x)$

多层前馈神经网络

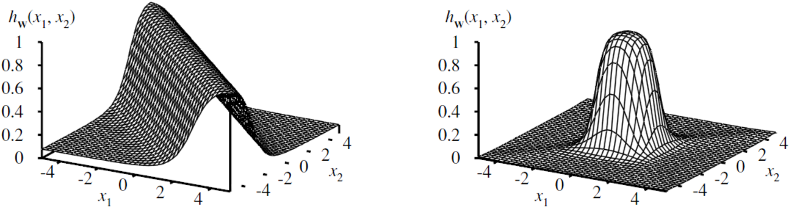
- 感知器只能表示线性分离器
- 多层网络可以表示什么函数？几乎任意函数！



$$a_i = g\left(\sum_j W_{ji}\, g\left(\sum_k W_{kj} a_k\right)\right)$$

多层网络

- 把两个平行但有反向峭壁的sigmoid单元相加形成一个山脊
- 把两个交叉的山脊相加形成一个山包

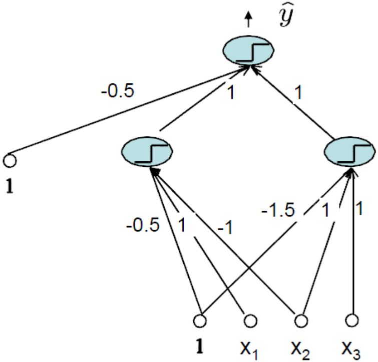


- 通过把不同高度的山包拼接在一起，可以逼近任意函数

  定理: 具有至少一个有足够多sigmoid单元的隐藏层的神经网络可以无限逼近任意函数。

表示布尔函数的神经网络：一个例子

Boolean function: $x_1 \wedge \neg x_2 \vee x_2 \wedge x_3$



OR units

AND units

任意布尔函数可以转换为合取范式(CNF)或析取范式(DNF)，因而可以用具有一个隐藏层的神经网络表示

权重训练

- 一个样例集合，每个样例由输入向量x和输出向量y组成
- 平方误差损失: $Loss = \sum_k Loss_k$, $Loss_k = (y_k - a_k)^2$, 其中a_k是神经网络的第k个输出
- 权重更新公式: $w_{ij} \leftarrow w_{ij} - \alpha \partial Loss / \partial w_{ij}$
- 给定任意的网络结构，我们如何高效地计算梯度？
- 答案: 反向传播算法

前向和反向阶段

前向阶段:

- 把输入向前传播以计算每个单元的输出
- Output a_j at unit j: $a_j = g(in_j)$ where $in_j = \sum_i w_{ij} a_i$

反向阶段:

- 把误差反向传播
- For an output unit j: $\Delta_j = g'(in_j)(y_j - a_j)$
- For an hidden unit i: $\Delta_i = g'(in_i) \sum_j w_{ij} \Delta_j$

多层网络学习的后向传播算法

```
function BACK-PROP-LEARNING(examples, network) returns a neural network
  inputs: examples, a set of examples, each with input vector x and output vector y
          network, a multilayer network with L layers, weights w_{i,j}, activation function g
  local variables: Δ, a vector of errors, indexed by network node
  for each weight w_{i,j} in network do
      w_{i,j} ← a small random number

  repeat
     for each example (x, y) in examples do
        /* Propagate the inputs forward to compute the outputs */
        for each node i in the input layer do
            a_i ← x_i
        for ℓ = 2 to L do
            for each node j in layer ℓ do
                in_j ← Σ_i w_{i,j} a_i
                a_j ← g(in_j)
        /* Propagate deltas backward from output layer to input layer */
        for each node j in the output layer do
            Δ[j] ← g'(in_j) × (y_j − a_j)
        for ℓ = L − 1 to 1 do
            for each node i in layer ℓ do
                Δ[i] ← g'(in_i) Σ_j w_{i,j} Δ[j]
        /* Update every weight in network using deltas */
        for each weight w_{i,j} in network do
           w_{i,j} ← w_{i,j} + α × a_i × Δ[j]
  until some stopping criterion is satisfied
  return network
```

输出层

$$
\begin{aligned}
\frac{\partial Loss_k}{\partial w_{j,k}} &= -2(y_k - a_k)\frac{\partial a_k}{\partial w_{j,k}} = -2(y_k - a_k)\frac{\partial g(in_k)}{\partial w_{j,k}} \\
&= -2(y_k - a_k)g'(in_k)\frac{\partial in_k}{\partial w_{j,k}} = -2(y_k - a_k)g'(in_k)\frac{\partial}{\partial w_{j,k}}\left(\sum_j w_{j,k}a_j\right) \\
&= -2(y_k - a_k)g'(in_k)a_j = -a_j\Delta_k \,,
\end{aligned}
$$

隐藏层

$$
\begin{aligned}
\frac{\partial Loss_k}{\partial w_{i,j}} &= -2(y_k - a_k)\frac{\partial a_k}{\partial w_{i,j}} = -2(y_k - a_k)\frac{\partial g(in_k)}{\partial w_{i,j}} \\
&= -2(y_k - a_k)g'(in_k)\frac{\partial in_k}{\partial w_{i,j}} = -2\Delta_k \frac{\partial}{\partial w_{i,j}}\left(\sum_j w_{j,k}a_j\right) \\
&= -2\Delta_k w_{j,k}\frac{\partial a_j}{\partial w_{i,j}} = -2\Delta_k w_{j,k}\frac{\partial g(in_j)}{\partial w_{i,j}} \\
&= -2\Delta_k w_{j,k} g'(in_j)\frac{\partial in_j}{\partial w_{i,j}} \\
&= -2\Delta_k w_{j,k} g'(in_j)\frac{\partial}{\partial w_{i,j}}\left(\sum_i w_{i,j}a_i\right) \\
&= -2\Delta_k w_{j,k} g'(in_j) a_i = -a_i\Delta_j \ ,
\end{aligned}
$$

Forward and backward phases

Forward phase:

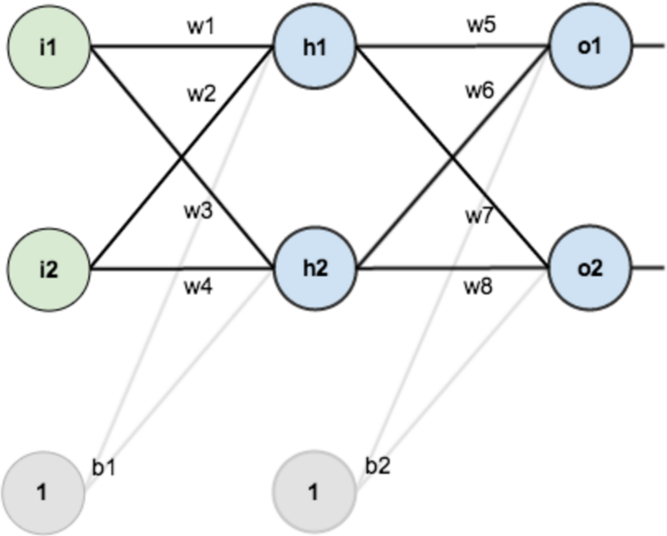
- Propagate inputs forward to compute the output of each unit
- Output a_j at unit j: $a_j = g(in_j)$ where $in_j = \sum_i w_{ij} a_i$

Backward phase:

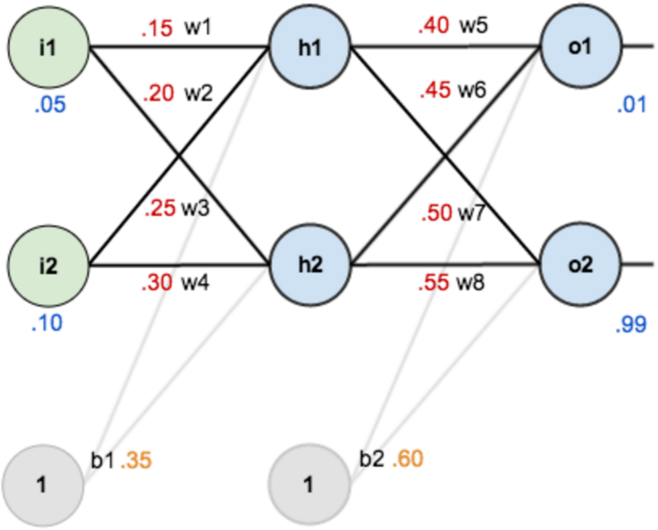
- Propagate errors backward
- For an output unit j:
  $\Delta_j = g'(in_j)(y_j - a_j) = a_j(1 - a_j)(y_j - a_j)$
- For an hidden unit i:
  $\Delta_i = g'(in_i) \sum_j w_{ij} \Delta_j = a_i(1 - a_i) \sum_j w_{ij} \Delta_j$

Weight updating: $w_{ij} \leftarrow w_{ij} + \alpha a_i \Delta_j$

The network structure



The numbers


i1
.05
i2
.10
.15 w1
.20 w2
.25 w3
.30 w4
h1
h2
.40 w5
.45 w6
.50 w7
.55 w8
o1
.01
o2
.99
1
b1 .35
1
b2 .60

前向传递

- $in_{h_1} = w_1 i_1 + w_2 i_2 + b_1 = 0.05 * 0.15 + 0.10 * 0.20 + 0.35 = 0.3775$
- $out_{h_1} = g(in_{h_1}) = \frac{1}{1+e^{-0.3775}} = 0.593269992$
- $out_{h_2} = 0.596884378$
- $in_{o_1} = w_5 out_{h_1} + w_6 out_{h_2} + b_2 = 0.40 * 0.593269992 + 0.45 * 0.596884378 + 0.60 = 1.105905967$
- $out_{o_1} = g(in_{o_1}) = \frac{1}{1+e^{-1.105905967}} = 0.75136507$
- $out_{o_2} = 0.772928465$

反向传递

Let $\alpha = 0.5$

- $\Delta_{o_1} = 0.75136507(1 - 0.75136507)(0.01 - 0.75136507) = -0.138498562$
- $w_5^+ = w_5 + \alpha \cdot out_{h_1} \cdot \Delta_{o_1} = 0.40 - 0.5 * 0.593269992 * 0.138498562 = 0.35891648$
- $w_6^+ = w_6 + \alpha \cdot out_{h_2} \cdot \Delta_{o_1} = 0.45 - 0.5 * 0.596884378 * 0.138498562 = 0.408666186$

反向传递

- $\Delta_{o_2} = 0.772928465(1 - 0.772928465)(0.99 - 0.772928465) = 0.0380982366$
- $w_7^+ = w_7 + \alpha \cdot out_{h_1} \cdot \Delta_{o_2} = 0.50 + 0.5 * 0.593269992 * 0.0380982366 = 0.511301270$
- $w_8^+ = w_8 + \alpha \cdot out_{h_2} \cdot \Delta_{o_2} = 0.55 + 0.5 * 0.596884378 * 0.0380982366 = 0.561370121$

反向传递

- $\Delta_{h_1} = g'(in_{h_1})(w_5\Delta_{o_1} + w_7\Delta_{o_2}) = 0.593269992(1 - 0.593269992)(0.40 * (-0.138498562) + 0.50 * 0.0380982366) = -0.241300709 * 0.036350306$
- $w_1^+ = w_1 + \alpha \cdot i_1 \cdot \Delta_{h_1} = 0.15 - 0.5 * 0.05 * 0.241300709 * 0.036350306 = 0.149780716$
- $w_2^+ = 0.19956143$
- $w_3^+ = 0.24975114$
- $w_4^+ = 0.29950229$

餐馆例子

- 首先我们需要确定网络的结构.
- 我们有10个属性，因而我们需要10个输入单元.
- 我们应该有一个还是两个隐藏层？每层多少个节点？它们是否是全连接的？
- 没有好的理论告诉我们答案.
- 我们可以使用交叉验证：试几个不同的结构，看哪个效果最好？
- 结果是对这个问题合适的一个网络有一个隐藏层，它包含4个节点.

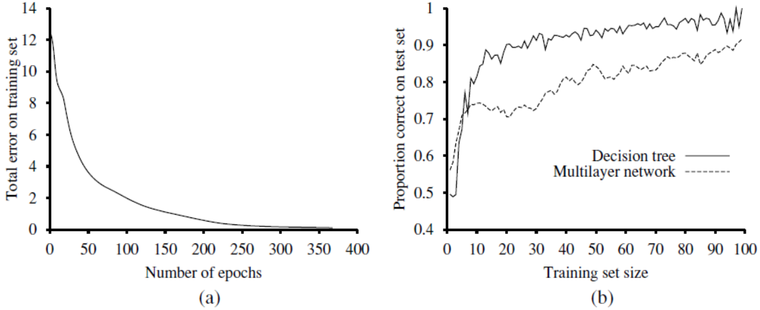


Figure 18.25 (a) Training curve showing the gradual reduction in error as weights are modified over several epochs, for a given set of examples in the restaurant domain. (b) Comparative learning curves showing that decision-tree learning does slightly better on the restaurant problem than back-propagation in a multilayer network.

epoch: 一次遍历所有样例的权重更新