



这个例子说明了一些简单的线性边界，如上面的 $g_1$ 和 $g_2$ ，在经过一层感知机模型，经线性组合后，可以得到一些非线性的复杂边界（AND运算） $G(x)$ 。

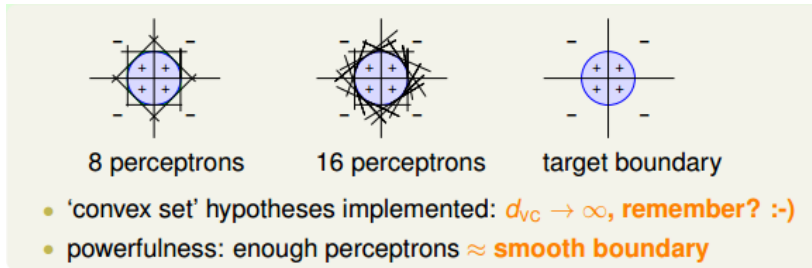
除此之外，或（OR）运算和非（NOT）运算都可以由感知机建立相应的模型，非常简单。

所以说，linear aggregation of perceptrons实际上是非常powerful的模型同时也是非常complicated模型。

再看下面一个例子，如果二维平面上有个圆形区域，圆内表示+1，圆外表示-1。这样复杂的圆形边界是没有办法使用单一perceptron来解决的。如果使用8个perceptrons，用刚才的方法线性组合起来，能够得到一个很接近圆形的边界（八边形）。如果使用16个perceptrons，那么得到的边界更接近圆形（十六边形）。

因此，使用的perceptrons越多，就能得到各种任意的convex set，即凸多边形边界。

convex set的VC Dimension趋向于无穷大（ $2^N$ ）。这表示只要perceptrons够多，我们能得到任意可能的情况，可能的模型。但是，这样模型复杂度可能会变得很大，从而造成过拟合（overfitting）。



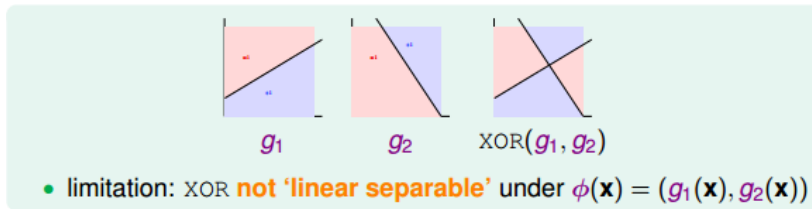
总的来说，足够数目的perceptrons线性组合能够得到比较平滑的边界和稳定的模型，这也是aggregation的特点之一。

但是，也有单层perceptrons线性组合做不到的事情。

例如刚才我们将的AND、OR、NOT三种逻辑运算都可以由单层perceptrons做到，而如果是异或（XOR）操作，就没有办法只用单层perceptrons实现。

这是因为XOR得到的是非线性可分的区域，没有办法由 $g_1$ 和 $g_2$ 线性组合实现。

所以说linear aggregation of perceptrons模型的复杂度还是有限制的。



那么，为了实现XOR操作，可以使用多层perceptrons，也就是说一次transform不行，我们就用多层的transform，这其实就是Basic Neural Network的基本原型。

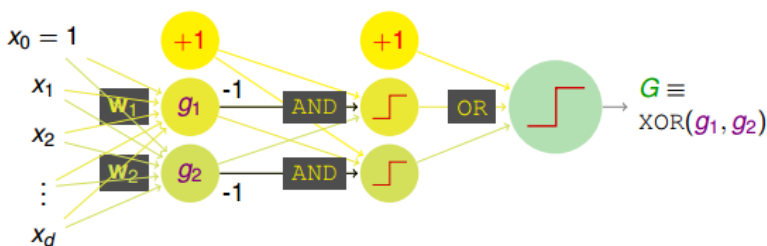
下面我们就尝试使用两层perceptrons来实现XOR的操作。

首先，根据布尔运算，异或XOR操作可以拆分成：

$$XOR(g_1, g_2) = OR(AND(-g_1, g_2), AND(g_1, -g_2))$$

这种拆分实际上就包含了两层transform。

第一层仅有AND操作，第二层是OR操作。这种两层的感知机模型如下所示：



这样，从AND操作到XOR操作，从简单的aggregation of perceptrons到multi-layer perceptrons，感知机层数在增加，模型的复杂度也在增加，使最后得到的G能更容易解决一些非线性的复杂问题。这就是基本神经网络的基本模型。

perceptron (simple)  
 $\Rightarrow$  aggregation of perceptrons (powerful)  
 $\Rightarrow$  multi-layer perceptrons (more powerful)

这里所说的感知机模型实际上就是在模仿人类的神经元模型（这就是Neural Network名称的由来）。  
 感知机模型每个节点的输入就对应神经元的树突dendrite，感知机每个节点的输出就对应神经元的轴突axon。

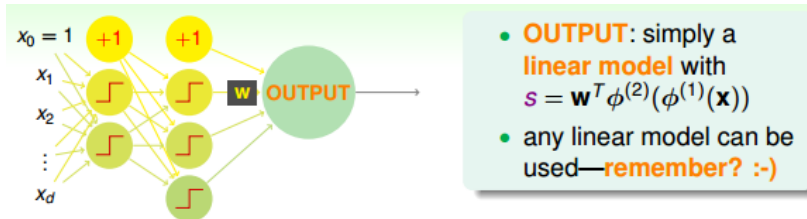
## 2. Neural Network Hypothesis

感知机模型其实就是Neural Network。

输入部分经过一层一层的运算，相当于一层一层的transform，最后通过最后一层的权重，得到一个分数score。

即在OUTPUT层，输出的就是一个线性模型。

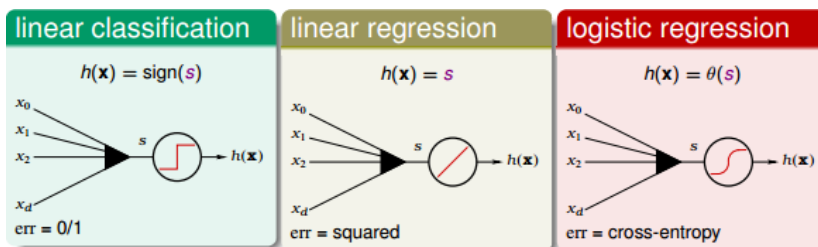
得到s后，下一步再进行处理。



已经了解了三种线性模型：linear classification，linear regression，logistic regression。那么，对于OUTPUT层的分数s，根据具体问题，可以选择最合适的线性模型。

如果是binary classification问题，可以选择linear classification模型；如果是linear regression问题，可以选择linear regression模型；如果是soft classification问题，则可以选择logistic regression模型。

eg: linear regression为例，选择squared error来进行衡量。



对于中间层，每个节点对应一个perceptron，都有一个transform运算。transformation function是阶梯函数sign()。

如果每个节点的transformation function都是线性运算（跟OUTPUT端一样），那么由每个节点的线性模型组合成的神经网络模型也必然是线性的。这跟直接使用一个线性模型在效果上并没有什么差异，模型能力不强，反而花费了更多不必要的力气。所以一般来说，中间节点不会选择线性模型。

如果每个节点的transformation function都是阶梯函数（即sign()函数）。这是一个非线性模型，但是由于阶梯函数是离散的，并不是处处可导，所以在优化计算时比较难处理。所以，一般也不选择阶梯函数作为transformation function。

既然线性函数和阶梯函数都不太适合作为transformation function，那么最常用的一种transformation function就是tanh(s)，其表达式如下：

$$\tanh(s) = \frac{\exp(s) - \exp(-s)}{\exp(s) + \exp(-s)}$$

tanh(s)函数是一个平滑函数，类似“s”型。

当|s|比较大的时候，tanh(s)与阶梯函数相近；当|s|比较小的时候，tanh(s)与线性函数比较接近。

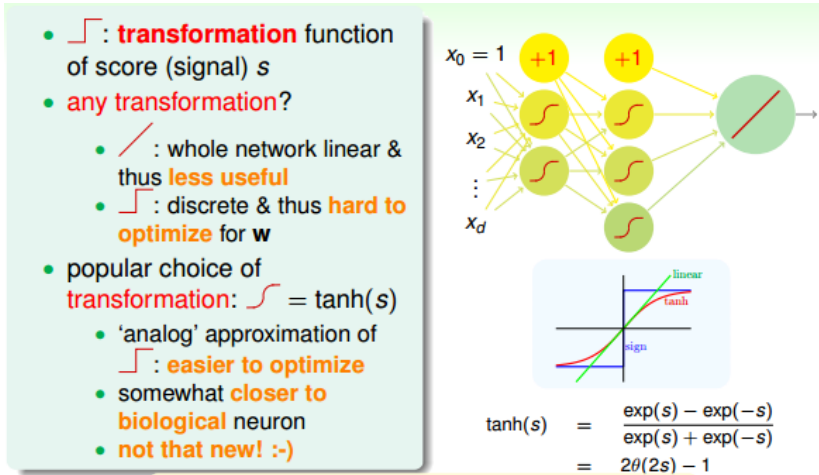
从数学上来说，由于处处连续可导，便于最优化计算。而且形状上类似阶梯函数，具有非线性的性质，可以得到比较复杂强大的模型。

tanh(x)函数与sigmoid函数存在下列关系：

$$\tanh(s) = 2\theta(2s) - 1$$

其中,

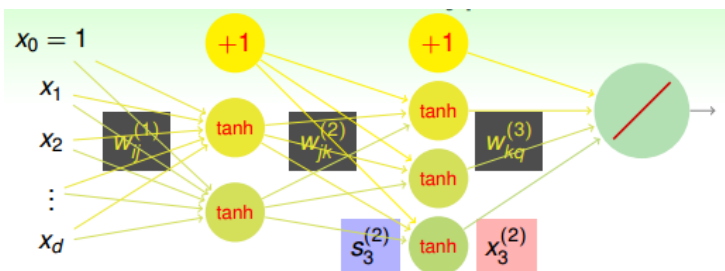
$$\theta(s) = \frac{1}{1 + \exp(-s)}$$



实际应用中, 可以选择其它的transformation function, 不同的transformation function, 则有不同的推导过程。

Neural Network Hypothesis的结构分析

如下图所示, 该神经网络左边是输入层, 中间两层是隐藏层, 右边是输出层。整体上来说, 我们设定输入层为第0层, 然后往右分别是第一层、第二层, 输出层即为第3层。



Neural Network Hypothesis中,  $d^{(0)}, d^{(1)}, \dots, d^{(L)}$  分别表示神经网络的第几层, 其中 $L$ 为总层数。

每一层的权重  $w_{ij}^{(l)}$ , 上标满足  $1 \leq l \leq L$ , 表示是位于哪一层。

下标满足  $0 \leq i \leq d^{(l-1)}$ , 表示前一层输出的个数加上bias项 (常数项)。

下标满足  $1 \leq j \leq d^{(l)}$ , 表示该层节点的个数 (不包括bias项)。

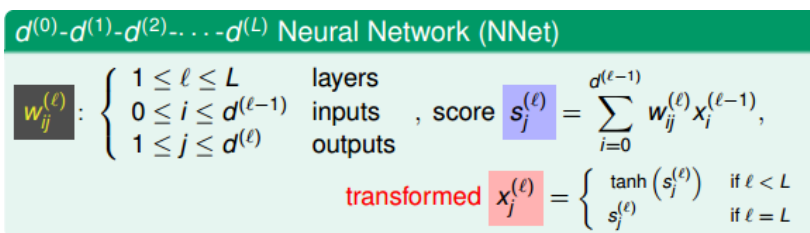
对于每层的分数score, 它的表达式为:

$$s_j^{(l)} = \sum_{i=0}^{d^{(l-1)}} w_{ij}^{(l)} x_i^{(l-1)}$$

对于每层的transformation function, 它的表达式为:

$$x_j^{(l)} = \begin{cases} \tanh(s_j^{(l)}), & \text{if } l < L \\ s_j^{(l)}, & \text{if } l = L \end{cases}$$

因为是regression模型, 所以在输出层 ( $l=L$ ) 直接得到  $x_j^{(l)} = s_j^{(l)}$ 。



每一层输入到输出的运算过程, 实际上都是一种transformation, 而转换的关键在于每个权重值  $w_{ij}^{(l)}$ 。

每层网络利用输入 $x$ 和权重 $w$ 的乘积, 在经过tanh函数, 得到该层的输出, 从左到右, 一层一层地进行。

$x$ 和 $w$ 的乘积  $\sum_{i=0}^{d^{(l-1)}} w_{ij}^{(l)} x_i^{(l-1)}$  越大, 那么  $\tanh(wx)$  就会越接近1, 表明这种transformation效果越好。乘积越大, 表明两个向量内

积越大，越接近平行，则表明 $w$ 和 $x$ 有模式上的相似性。从而，更进一步说明了如果每一层的输入向量 $x$ 和权重向量 $w$ 具有模式上的相似性，比较接近平行，那么transformation的效果就比较好，就能得到表现良好的神经网络模型。

也就是说，神经网络训练的核心就是pattern extraction，即从数据中找到数据本身蕴含的模式和规律。通过一层一层找到这些模式，找到与输入向量 $x$ 最契合的权重向量 $w$ ，最后再由 $G$ 输出结果。

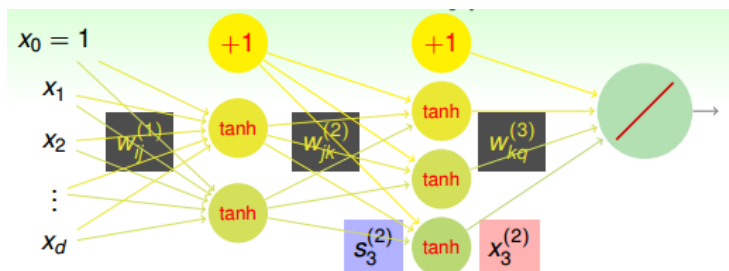
- each layer: **transformation** to be **learned** from data

$$\phi^{(\ell)}(\mathbf{x}) = \tanh \left( \begin{bmatrix} \sum_{i=0}^{d^{(\ell-1)}} w_{i1}^{(\ell)} x_i^{(\ell-1)} \\ \vdots \end{bmatrix} \right)$$

—whether  $\mathbf{x}$  'matches' weight vectors in pattern

NNet: **pattern extraction** with layers of **connection weights**

### 3. Neural Network Learning



目标是找到最佳的 $w_{ij}^{(l)}$ 让 $E_{in}(w_{ij}^{(l)})$ 最小化。

如果只有一层隐藏层，就相当于 aggregation of perceptrons。可以使用gradient boosting算法来一个一个确定隐藏层每个神经元的权重，输入层到隐藏层的权重可以通过C&RT算法计算的到。

如果隐藏层个数有两个或者更多，就要考虑使用其它方法。

根据error function的思想，从输出层来看，可以得到每个样本神经网络预测值与实际值之间的squared error:

$$e_n = (y_n - \text{NNet}(x_n))^2, \text{ 这是单个样本点的error.}$$

那么，只要能建立 $e_n$ 与每个权重 $w_{ij}^{(l)}$ 的函数关系，就可以利用GD或SGD算法对 $w_{ij}^{(l)}$ 求偏微分，不断迭代优化 $w_{ij}^{(l)}$ 值，最终得到使 $e_n$ 最小时对应的 $w_{ij}^{(l)}$ 。

- goal: learning all  $\{w_{ij}^{(\ell)}\}$  to **minimize**  $E_{in}(\{w_{ij}^{(\ell)}\})$
- one hidden layer: simply **aggregation of perceptrons**  
—**gradient boosting** to determine hidden neuron one by one
- multiple hidden layers? **not easy**
- let  $e_n = (y_n - \text{NNet}(\mathbf{x}_n))^2$ :  
can apply **(stochastic) GD** after computing  $\frac{\partial e_n}{\partial w_{ij}^{(l)}}$ !

为了建立 $e_n$ 与各层权重 $w_{ij}^{(l)}$ 的函数关系，求出 $e_n$ 对 $w_{ij}^{(l)}$ 的偏导数 $\frac{\partial e_n}{\partial w_{ij}^{(l)}}$ ，我们先来看输出层如何计算 $\frac{\partial e_n}{\partial w_{i1}^{(L)}}$ 。

$e_n$ 与 $w_{i1}^{(L)}$ 的函数关系为：

$$e_n = (y_n - \text{NNet}(\mathbf{x}_n))^2 = (y_n - s_1^{(L)})^2 = \left( y_n - \sum_{i=0}^{d^{(L-1)}} w_{i1}^{(L)} x_i^{(L-1)} \right)^2$$

计算 $e_n$ 对 $w_{i1}^{(L)}$ 的偏导数，得到：

**specially (output layer)**  
 $(0 \leq i \leq d^{(L-1)})$

$$\begin{aligned} & \frac{\partial e_n}{\partial w_{i1}^{(L)}} \\ = & \frac{\partial e_n}{\partial s_1^{(L)}} \cdot \frac{\partial s_1^{(L)}}{\partial w_{i1}^{(L)}} \\ = & -2(y_n - s_1^{(L)}) \cdot (x_i^{(L-1)}) \end{aligned}$$

以上是输出层求偏导的结果。

如果是其它层，即  $l \neq L$ ，偏导计算可以写成如下形式：

**generally ( $1 \leq \ell < L$ )**  
 $(0 \leq i \leq d^{(\ell-1)}; 1 \leq j \leq d^{(\ell)})$

$$\begin{aligned} & \frac{\partial e_n}{\partial w_{ij}^{(\ell)}} \\ = & \frac{\partial e_n}{\partial s_j^{(\ell)}} \cdot \frac{\partial s_j^{(\ell)}}{\partial w_{ij}^{(\ell)}} \\ = & \delta_j^{(\ell)} \cdot (x_i^{(\ell-1)}) \end{aligned}$$

上述推导中，令  $e_n$  与第  $l$  层第  $j$  个神经元的分数  $s_j^{(l)}$  的偏导数记为  $\delta_j^{(l)}$ 。即：

$$\frac{\partial e_n}{\partial s_j^{(l)}} = \delta_j^{(l)}$$

当  $l = L$  时， $\delta_1^{(L)} = -2(y_n - s_1^{(L)})$ ；当  $l \neq L$  时， $\delta_j^{(l)}$  是未知的，下面将进行运算推导，看看不同层之间的  $\delta_j^{(l)}$  是否有递推关系。

$$s_j^{(\ell)} \xRightarrow{\tanh} x_j^{(\ell)} \xRightarrow{w_{jk}^{(\ell+1)}} \begin{bmatrix} s_1^{(\ell+1)} \\ \vdots \\ s_k^{(\ell+1)} \\ \vdots \end{bmatrix} \Rightarrow \dots \Rightarrow e_n$$

如上图所示，第  $l$  层第  $j$  个神经元的分数  $s_j^{(l)}$  经过  $\tanh$  函数，得到该层输出  $x_j^{(l)}$ ，再与下一层权重  $w_{jk}^{(\ell+1)}$  相乘，得到第  $l+1$  层的分数  $s_k^{(\ell+1)}$ ，直到最后的输出层  $e_n$ 。

那么，利用上面  $s_j^{(l)}$  到  $s_j^{(l+1)}$  这样的递推关系，我们可以对偏导数  $\delta_j^{(l)}$  做一些中间变量替换处理，得到如下表达式：

$$\begin{aligned} \delta_j^{(\ell)} = \frac{\partial e_n}{\partial s_j^{(\ell)}} &= \sum_{k=1}^{d^{(\ell+1)}} \frac{\partial e_n}{\partial s_k^{(\ell+1)}} \frac{\partial s_k^{(\ell+1)}}{\partial x_j^{(\ell)}} \frac{\partial x_j^{(\ell)}}{\partial s_j^{(\ell)}} \\ &= \sum_k \left( \delta_k^{(\ell+1)} \right) \left( w_{jk}^{(\ell+1)} \right) \left( \tanh' \left( s_j^{(\ell)} \right) \right) \end{aligned}$$

上式中有一个求和项，其中  $k$  表示下一层即  $l+1$  层神经元的个数。表明  $l$  层的  $s_j^{(l)}$  与  $l+1$  层的所有  $s_k^{(l+1)}$  都有关系。因为  $s_j^{(l)}$  参与到每个  $s_k^{(l+1)}$  的运算中了。

这样，得到了  $\delta_j^{(l)}$  与  $\delta_k^{(l+1)}$  的递推关系。

也就是说如果知道了  $\delta_k^{(l+1)}$  的值，就能推导出  $\delta_j^{(l)}$  的值。

而最后一层，即输出层的  $\delta_1^{(L)} = -2(y_n - s_1^{(L)})$ ，那么就能一层一层往前推导，得到每一层的  $\delta_j^{(l)}$ ，从而可以计算出  $e_n$  对各个  $w_{ij}^{(l)}$  的偏导数  $\frac{\partial e_n}{\partial w_{ij}^{(l)}}$ 。

计算完偏微分之后，就可以使用GD或SGD算法进行权重的迭代优化，最终得到最优解。



神经网络中，这种从后往前的推导方法称为Backpropagation Algorithm，即我们常常听到的BP神经网络算法。它的算法流程如下所示：

### Backprop on NNet

```

initialize all weights  $w_{ij}^{(\ell)}$ 
for  $t = 0, 1, \dots, T$ 
  ① stochastic: randomly pick  $n \in \{1, 2, \dots, N\}$ 
  ② forward: compute all  $x_i^{(\ell)}$  with  $\mathbf{x}^{(0)} = \mathbf{x}_n$ 
  ③ backward: compute all  $\delta_j^{(\ell)}$  subject to  $\mathbf{x}^{(0)} = \mathbf{x}_n$ 
  ④ gradient descent:  $w_{ij}^{(\ell)} \leftarrow w_{ij}^{(\ell)} - \eta x_i^{(\ell-1)} \delta_j^{(\ell)}$ 
return  $\mathcal{G}_{\text{NNet}}(\mathbf{x}) = \left( \dots \tanh \left( \sum_j w_{jk}^{(2)} \cdot \tanh \left( \sum_i w_{ij}^{(1)} x_i \right) \right) \right)$ 

```

上面采用的是SGD的方法，即每次迭代更新时只取一个点，这种做法一般不够稳定。

所以通常会采用mini-batch的方法，即每次选取一些数据，例如  $\frac{N}{10}$ ，来进行训练，最后求平均值更新权重 $w$ 。这种做法的实际效果会比较好一些。

## 4. Optimization and Regularization

$$E_{in}(\mathbf{w}) = \frac{1}{N} \sum_{n=1}^N \text{err} \left( \left( \dots \tanh \left( \sum_j w_{jk}^{(2)} \cdot \tanh \left( \sum_i w_{ij}^{(1)} x_{n,i} \right) \right) \right), y_n \right)$$

下面我们将主要分析神经网络的优化问题。由于神经网络由输入层、多个隐藏层、输出层构成，结构是比较复杂的非线性模型，因此  $E_{in}(w)$  可能有许多局部最小值，是non-convex的，找到全局最小值（global minimum）就会困难许多。而我们使用GD或SGD算法得到的很可能就是局部最小值（local minimum）。

基于这个问题，不同的初始值权重  $w_{ij}^{(l)}$  通常会得到不同的local minimum。也就是说最终的输出 $G$ 与初始权重  $w_{ij}^{(l)}$  有很大的关系。在选取  $w_{ij}^{(l)}$  上有个技巧，就是通常选择比较小的值，而且最好是随机random选择。这是因为，如果权重  $w_{ij}^{(l)}$  很大，那么根据tanh函数，得到的值会分布在两侧比较平缓的位置（类似于饱和saturation），这时候梯度很小，每次迭代权重可能只有微弱的变化，很难在全局上快速得到最优解。而随机选择的原因是通常对权重  $w_{ij}^{(l)}$  如何选择没有先验经验，只能通过random，从普遍概率上选择初始值，随机性避免了人为因素的干预，可以说更有可能经过迭代优化得到全局最优解。

- generally **non-convex** when multiple hidden layers
  - not easy to reach **global minimum**
  - GD/SGD with **backprop** only gives **local minimum**
- different initial  $w_{ij}^{(\ell)} \Rightarrow$  different **local minimum**
  - somewhat '**sensitive**' to initial weights
  - **large weights**  $\Rightarrow$  **saturate** (small gradient)
  - advice: try **some random & small ones**

下面从理论上看一下神经网络模型的VC Dimension。对于tanh这样的transfer function，其对应的整个模型的复杂度  $d_{vc} = O(VD)$ 。

其中 $V$ 是神经网络中神经元的个数（不包括bias点）， $D$ 表示所有权重值的数量。

所以，如果 $V$ 足够大的时候，VC Dimension也会非常大，这样神经网络可以训练出非常复杂的模型。但同时也可能会造成过拟合overfitting。所以，神经网络中神经元的数量 $V$ 不能太大。

为了防止神经网络过拟合，一个常用的方法就是使用regularization。之前我们就介绍过可以在error function中加入一个regularizer，例如熟悉的L2 regularizer  $\Omega(w)$ ：

$$\Omega(w) = \sum (w_{ij}^{(l)})^2$$

但是，使用L2 regularizer 有一个缺点，就是它使每个权重进行等比例缩小（shrink）。也就是说大的权重缩小程度较大，小的权重缩小程度较小。这会带来一个问题，就是等比例缩小很难得到值为零的权重。而我们恰恰希望某些权重  $w_{ij}^{(l)} = 0$ ，即权重的解是松散（sparse）的。因为这样能有效减少VC Dimension，从而减小模型复杂度，防止过拟合发生。

那么为了得到sparse解，有什么方法呢？我们之前就介绍过可以使用L1 regularizer：  $\sum |w_{ij}^{(l)}|$ ，但是这种做法存在一个缺点，就是包含绝对值不容易微分。

除此之外，另外一种比较常用的方法就是使用weight-elimination regularizer。weight-elimination regularizer类似于L2 regularizer，只不过是L2 regularizer上做了尺度的缩小，这样能使large weight和small weight都能得到同等程度的缩小，从而让更多权重最终为零。

weight-elimination regularizer的表达式如下：

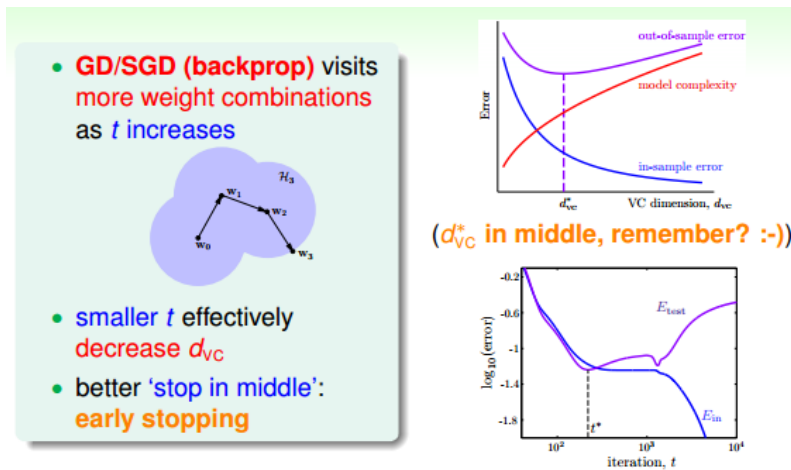
$$\sum \frac{(w_{ij}^{(l)})^2}{1 + (w_{ij}^{(l)})^2}$$

- 'shrink' weights:  
large weight → large shrink; small weight → small shrink
- want  $w_{ij}^{(\ell)} = 0$  (sparse) to effectively decrease  $d_{VC}$ 
  - L1 regularizer:  $\sum |w_{ij}^{(\ell)}|$ , but not differentiable
  - weight-elimination ('scaled' L2) regularizer:  
large weight → median shrink; small weight → median shrink

除了weight-elimination regularizer之外，还有另外一个很有效的regularization的方法，就是Early Stopping。

简而言之，就是神经网络训练的次数t不能太多。因为，t太大的时候，相当于给模型寻找最优值更多的可能性，模型更复杂，VC Dimension增大，可能会overfitting。

而t不太大时，能有效减少VC Dimension，降低模型复杂度，从而起到regularization的效果。 $E_{in}$ 和 $E_{test}$ 随训练次数t的关系如下图所示：



可以使用validation进行验证选择最佳训练次数 $t$ 。