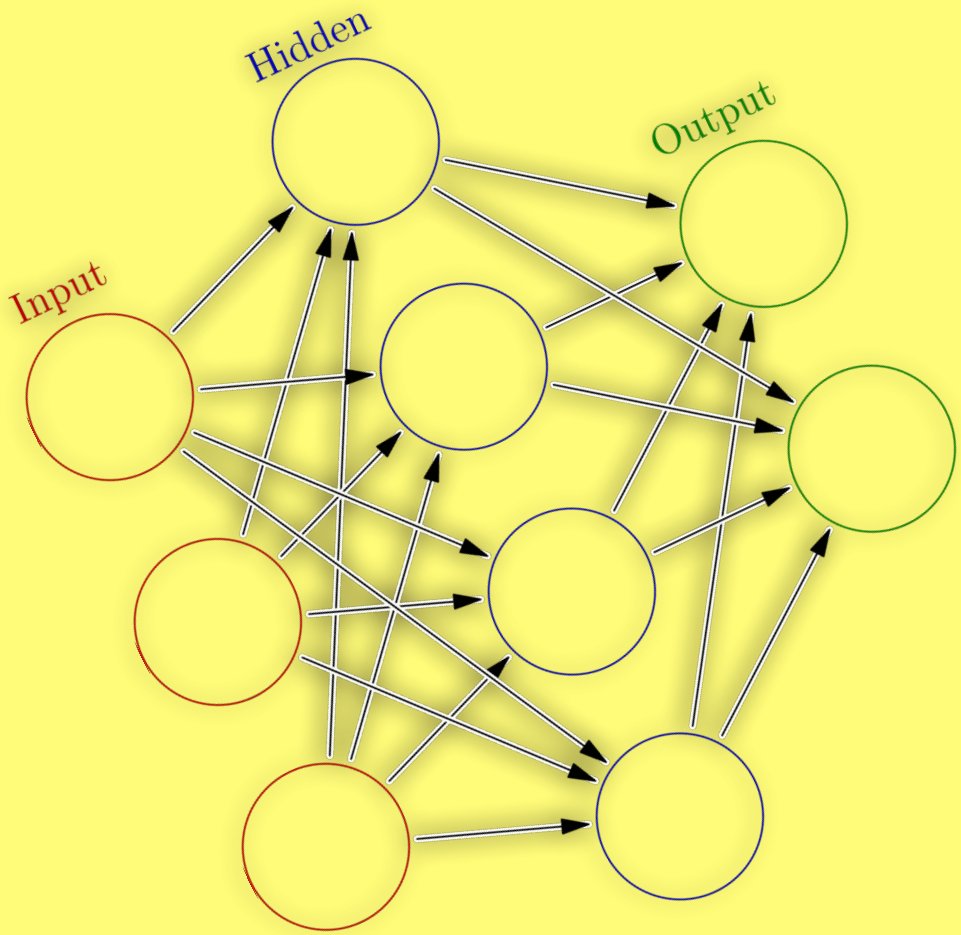
Architecture and Training of ANN



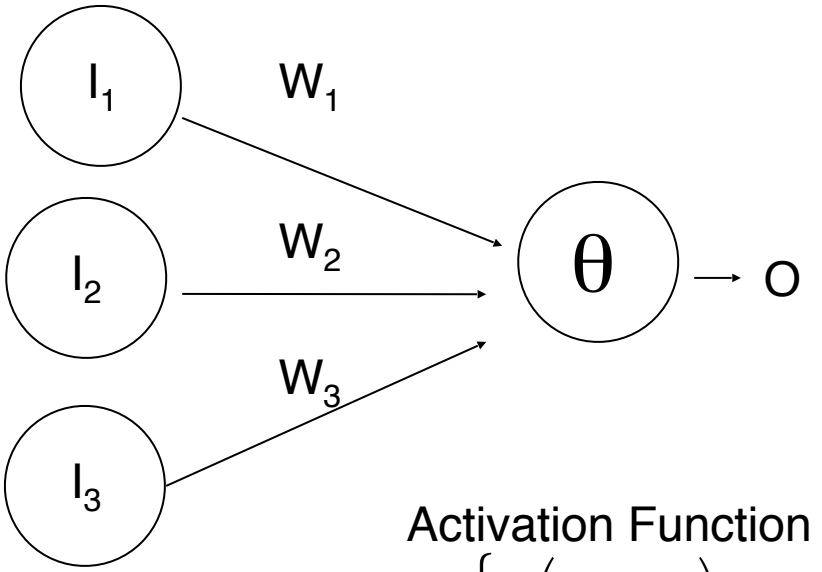
Re-creating the powerful human brain

Instructor

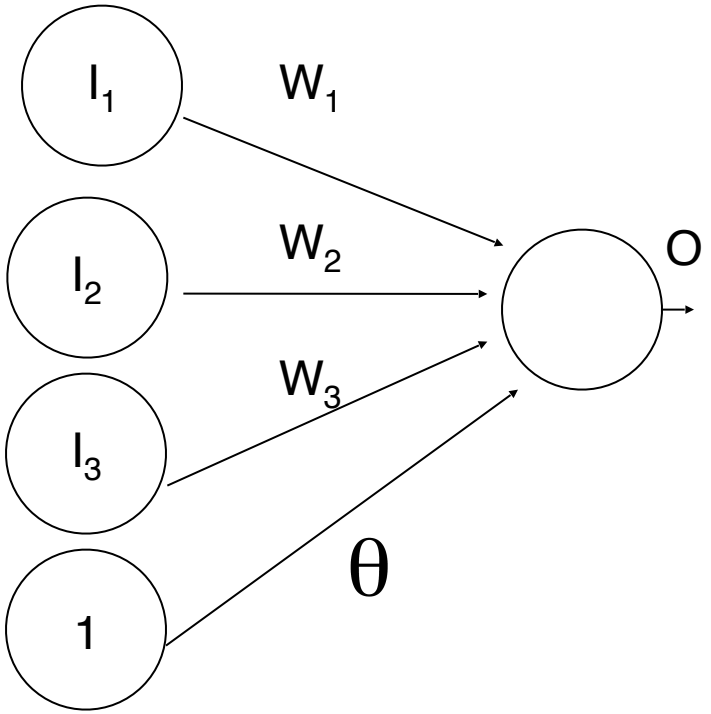
Mousum Dutta
Chief Data Scientist, Spotle.ai
Computer Science, IIT KGP

Architecture of Perceptron

- Perceptron is a linear and binary classifier
- Initial proposal of connectionist networks
- Essentially a linear discriminant composed of nodes, weights



or

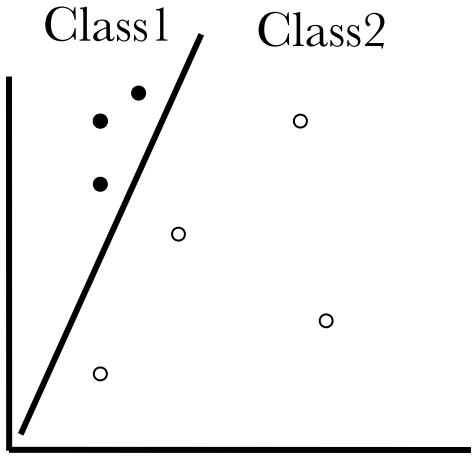


Activation Function

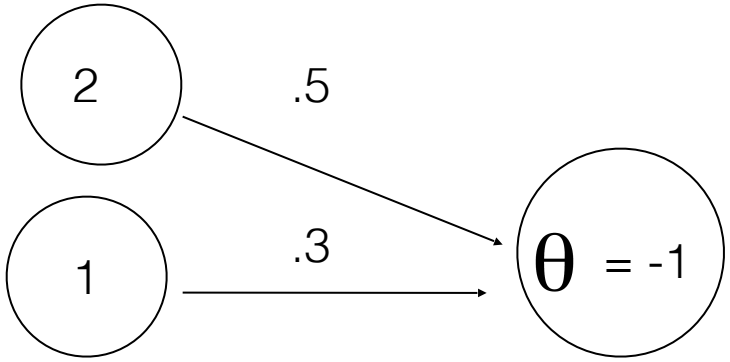
$$O = \begin{cases} 1 : \left(\sum_i w_i I_i \right) + \theta > 0 \\ 0 : otherwise \end{cases}$$

Perceptron

- Can add learning rate to speed up the learning process; just multiply in with delta computation
- Perceptron theorem: If a linear discriminant exists that can separate the classes without error, the training procedure is guaranteed to find that line or plane.



Perceptron Example



2(0.5) + 1(0.3) + -1 = 0.3 , O=1

Learning Procedure:

Randomly assign weights between 0 and 1.

Present inputs from training data

Get output O, update weights to push results toward our desired output T

Stop when no errors, or enough epochs completed, otherwise repeat

Training of Perceptron

$$w_i(t+1) = w_i(t) + \Delta w_i(t)$$

$$\Delta w_i(t) = (T - O) I_i$$

Weights include Threshold. T=Desired, O=Actual output.

Example: T=0, O=1, W1=0.5, W2=0.3, I1=2, I2=1,Theta=-1

$$w_1(t+1) = 0.5 + (0-1)(2) = -1.5$$

$$w_2(t+1) = 0.3 + (0-1)(1) = -0.7$$

$$w_\theta(t+1) = -1 + (0-1)(1) = -2$$

Can add learning rate to speed up the learning process; just multiply in with delta computation

How might you use a perceptron network?

We have some data regarding the diagnosis of patients with heart disease

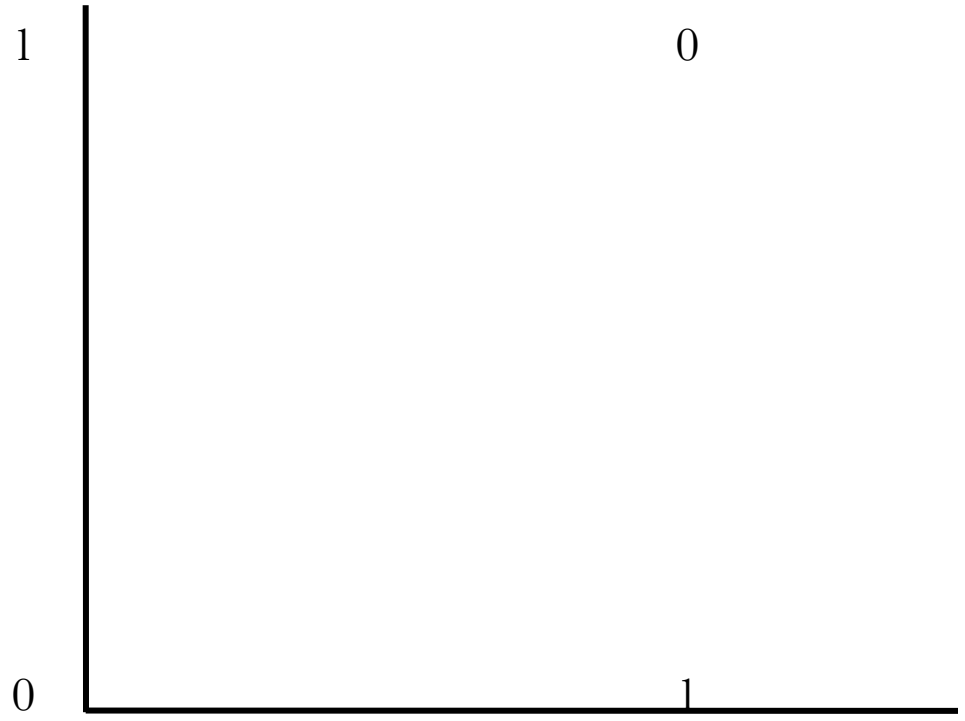
Age, Sex, Chest Pain Type, Resting BPS, Cholesterol, …, Heart Disease

Age	Sex	Chest Pain	BP	LDL	…	Heart Disease
67	1	4	120	229	…	1
37	1	3	130	250	…	0
41	0	2	130	204	…	0

Train network to predict heart disease of a new patient

Exclusive Or (XOR) Problem

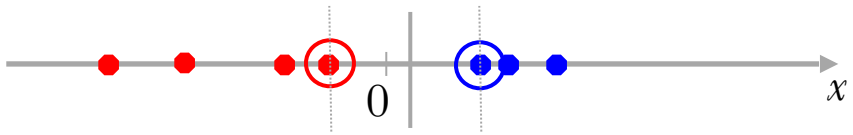
Input: 0,0 Output: 0
Input: 0,1 Output: 1
Input: 1,0 Output: 1
Input: 1,1 Output: 0



XOR Problem: Not Linearly Separable!

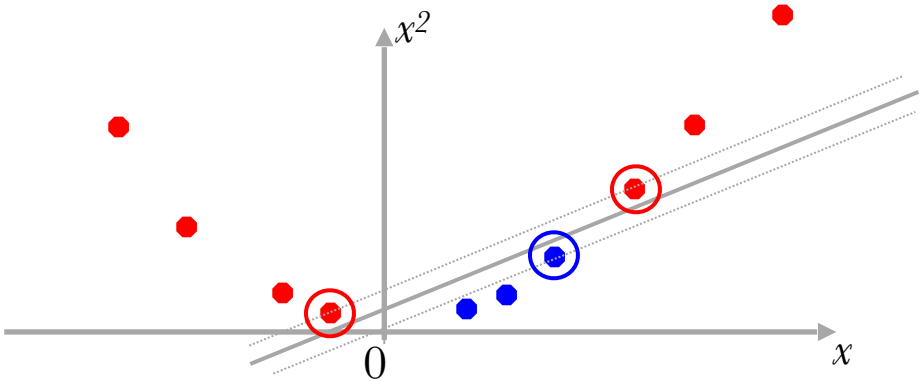
Non-linearity

- Datasets that are linearly separable with some noise work out great:



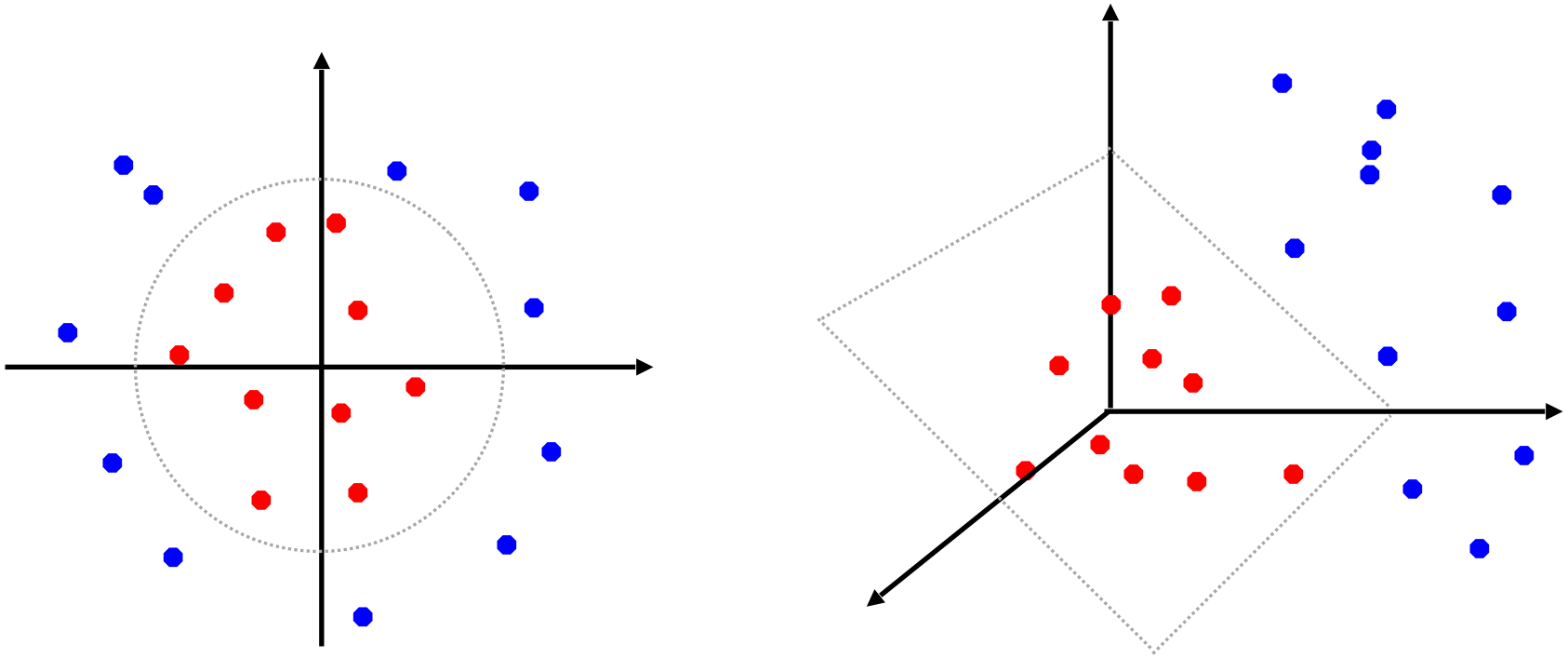
- But what are we going to do if the dataset is just too hard?

- How about… mapping data to a higher-dimensional space:



Non-linearity

General idea: The original feature space can always be mapped to some higher-dimensional feature space where the training set is separable:



LMS Gradient Descent

Using LMS (Least Mean Squeare), we want to minimize the error. We can do this by finding the direction on the error surface that most rapidly reduces the error rate; this is finding the slope of the error function by taking the derivative. The approach is called gradient descent.

To compute how much to change weight for link k:

$$\Delta w_k = -c\frac{\delta Error}{\delta w_k}$$

$$O_j = f(I\,W)$$

$$\frac{\delta O_j}{\delta w_k} = I_k f'\left(ActivationFunction(I_k W_k)\right)$$

Chain rule:

$$\frac{\delta Error}{\delta w_k} = \frac{\delta Error}{\delta O_j} * \frac{\delta O_j}{\delta w_k}$$

$$\boxed{\Delta w_k = -c\left(-(T_j - O_j)\right) I_k f'\left(ActivationFunction\right)}$$

$$\frac{\delta Error}{\delta O_j} = \frac{\delta \frac{1}{2}\sum_P (T_P - O_P)^2}{\delta O_j} = \frac{\delta \frac{1}{2}(T_j - O_j)^2}{\delta O_j} = -(T_j - O_j)$$

We can remove the sum since we are taking the partial derivative wrt Oj

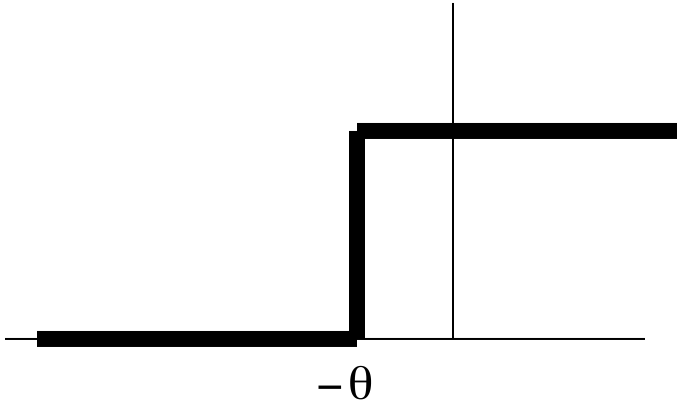
Activation Function

To apply the LMS learning rule, also known as the delta rule, we need a differentiable activation function.

$$\Delta w_k = cI_k\left(T_j - O_j\right)f'\left(ActivationFunction\right)$$

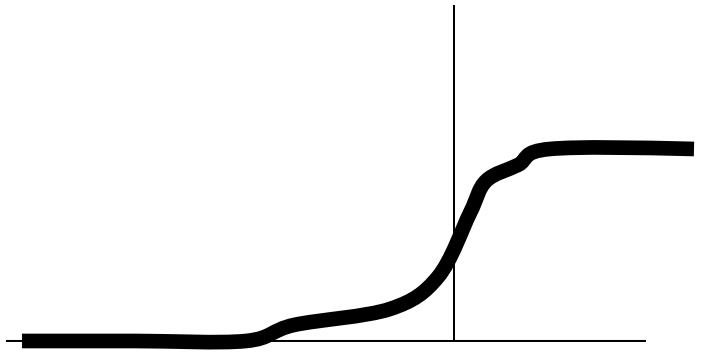
Old:

$$O = \begin{cases} 1 : \sum_i w_i I_i + \theta > 0 \\ 0 : otherwise \end{cases}$$



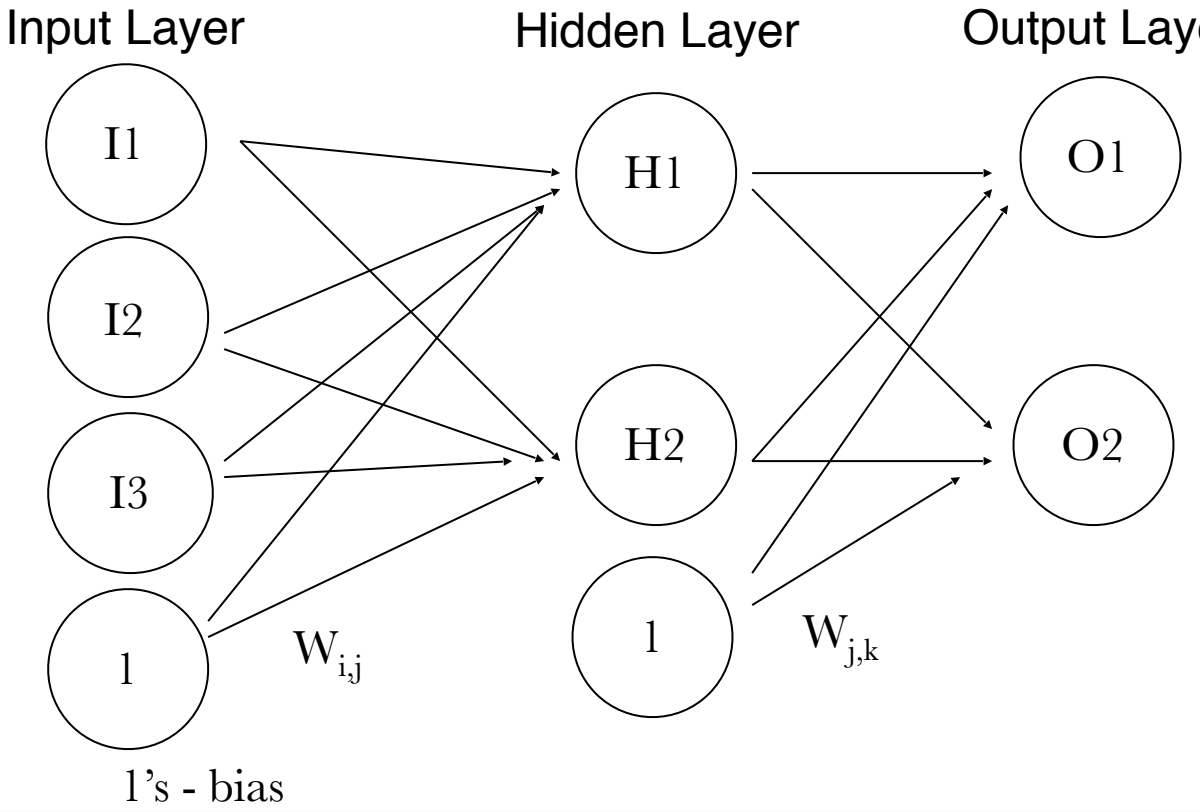
New:

$$O = \frac{1}{1 + e^{-\sum_i w_i I_i + \Theta}}$$



Backpropagation Networks

- To bypass the linear classification problem, we can construct multilayer networks. Typically we have fully connected, feedforward networks.
- Error is propagated backward to compute weights between Input layer and hidden layer



$$O(x) = \frac{1}{1 + e^{-\sum_j w_{j,x} H_j}}$$

$$H(x) = \frac{1}{1 + e^{-\sum_i w_{i,x} I_i}}$$

Backpropagation – Learning Procedure

Randomly assign weights (between 0-1)

Present inputs from training data, propagate to outputs

Compute outputs O, adjust weights according to the delta rule, backpropagating the errors. The weights will be nudged closer so that the network learns to give the desired output.

Repeat; stop when no errors, or enough epochs completed

Backpropagation - Modifying Weights

We had computed:

$$\Delta w_k = cI_k(T_j - O_j)f'(ActivationFunction), \quad f = \left(\frac{1}{1+e^{-sum}}\right)$$

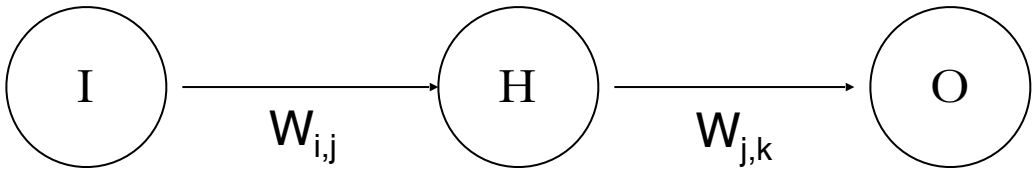
$$\Delta w_k = cI_k(T_j - O_j)(f(sum)(1 - f(sum)))$$

For the Output unit k, f(sum)=O(k). For the output units, this is:

$$\Delta w_{j,k} = cH_j(T_k - O_k)O_k(1 - O_k)$$

For the Hidden units (skipping some math), this is:

$$\Delta w_{i,j} = cH_j(1 - H_j)I_i \sum_k (T_k - O_k)O_k(1 - O_k)w_{j,k}$$



Backpropagation

- Very powerful - can learn any function, given enough hidden units! With enough hidden units, we can generate any function.
- Have the same problems of Generalization vs. Memorization. With too many units, we will tend to memorize the input and not generalize well. Some schemes exist to “prune” the neural network.
- Networks require extensive training, many parameters to fiddle with. Can be extremely slow to train. May also fall into local minima.
- Inherently parallel algorithm, ideal for multiprocessor hardware.
- Despite the cons, a very powerful algorithm that has seen widespread successful deployment.