Neural Networks

Multi-Layer Networks

CSCI 4850/5850

Linear = Limited

Single-layer networks are inherently constrained in the kinds of input-output mappings (i.e. functions) that they can produce: **regardless of the learning algorithm used**!

In particular, if input vectors are not **linearly independent**, then the range of allowable outputs is limited.

For perfect performance, the target, t, for input vector, p, must be…

For classification, this is the **linear separability** constraint

$$t^p = g(\boldsymbol{w} \cdot \boldsymbol{x}^p) = g\left(\sum_{n=1}^{N} v_n(\boldsymbol{w} \cdot \boldsymbol{x}^n)\right) = g\left(\sum_{n=1}^{N} v_n g^{-1}(t^n)\right) \text{ where } \quad \boldsymbol{x}^p = \sum_{n=1}^{N} v_n \boldsymbol{x}^n$$

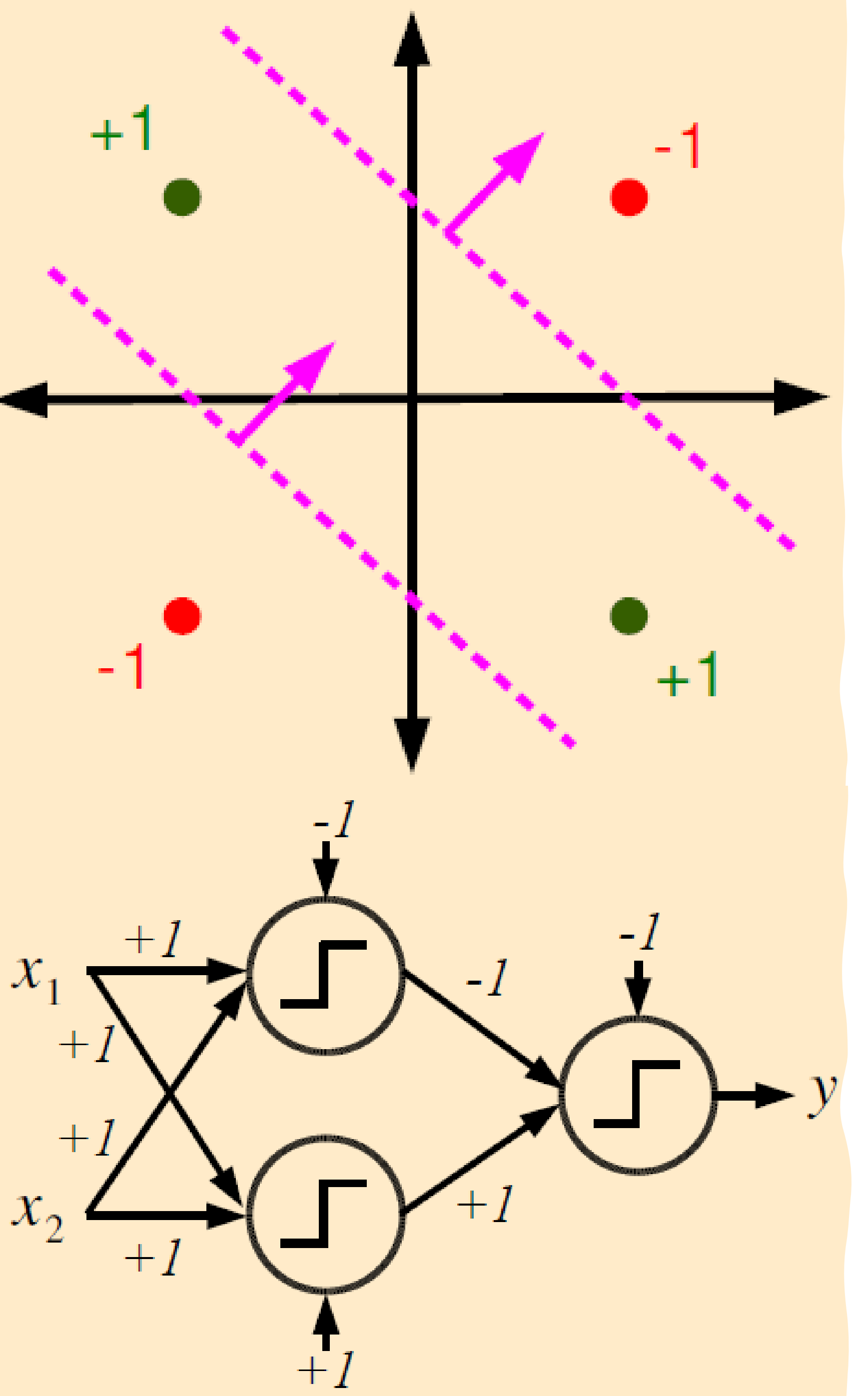
Constraints of Single-layer Nets


```
Classes:
Blue = 0
Red = 1

Data (Top):
[-1 -1] = 1
[-1  1] = 1
[ 1 -1] = 0
[ 1  1] = 0

Data (Bottom):
[-1 -1] = 0
[-1  1] = 1
[ 1 -1] = 1
[ 1  1] = 0
```

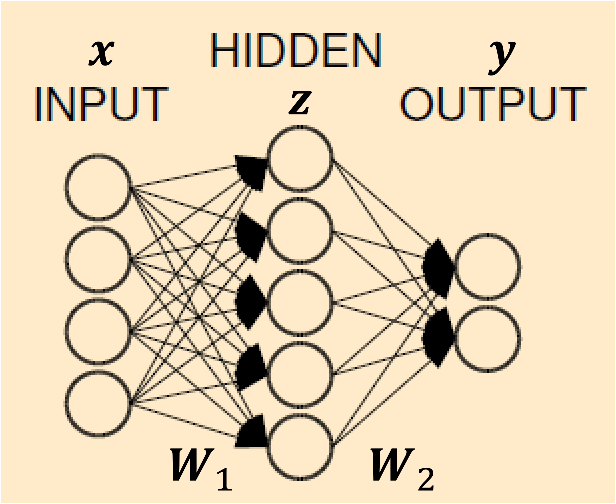
The Utility of Multiple Layers



- Multilayer networks can solve **non-linearly separable** problems by combining multiple partitions of the input vector space
- Different partitions are computed using different **hidden units**

Key: Linear/Non-linear Activation Functions

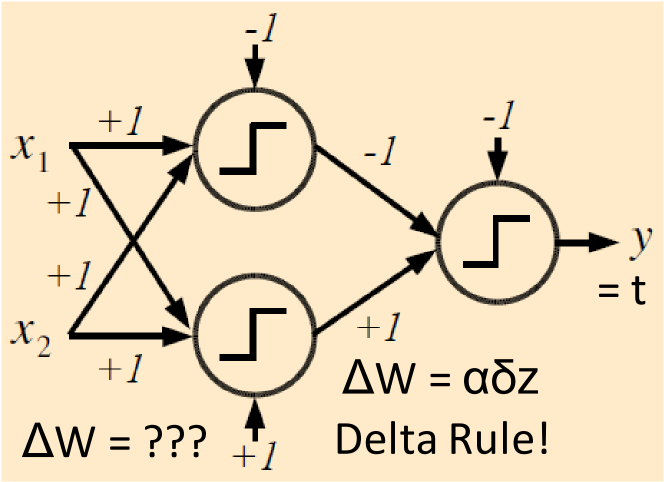
- Even a multilayer network will provide **no benefit** if the hidden units all utilize a **linear/affine activation function**



$$y = W_2 z = W_2(W_1 x) = (W_2 W_1)x = W_m x$$

What to do?

- For single-layer networks, the **target activation** for **output units** is known from the *training data*
- But, it's not clear what the activations of the hidden units need to be...

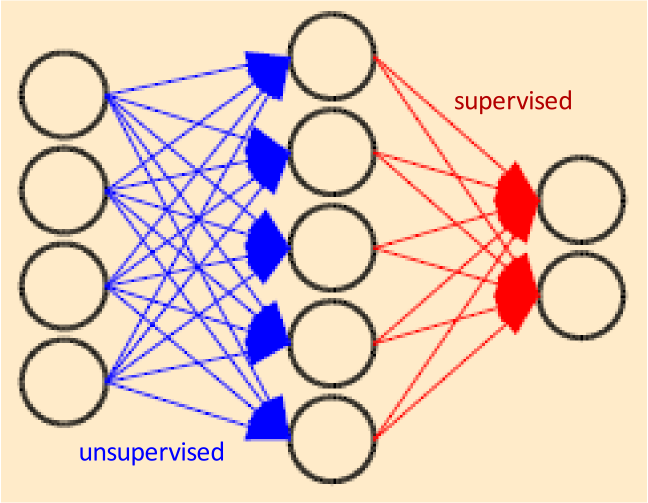


z = ???

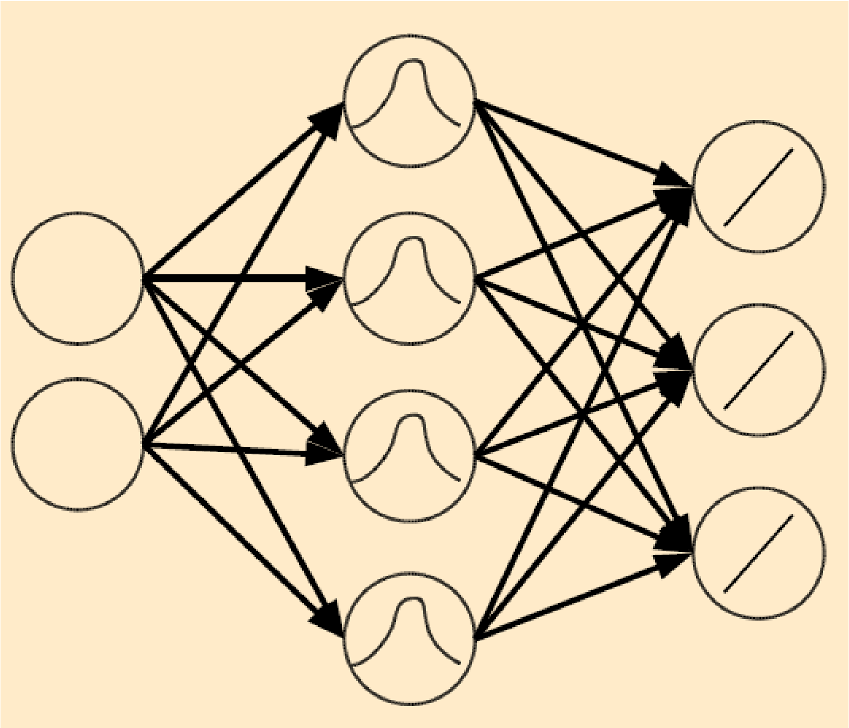
One Potential Solution

Apply *unsupervised* learning

Different kind of learning rule that tries to adapt the weights into something useful without *target* information



Radial Basis Function Networks



Counterpropagation Networks - Hecht-Neilsen, 1987

$$y_k(\boldsymbol{x}) = \sum_{j=1}^{M} w_{kj}\phi_j(\boldsymbol{x})$$

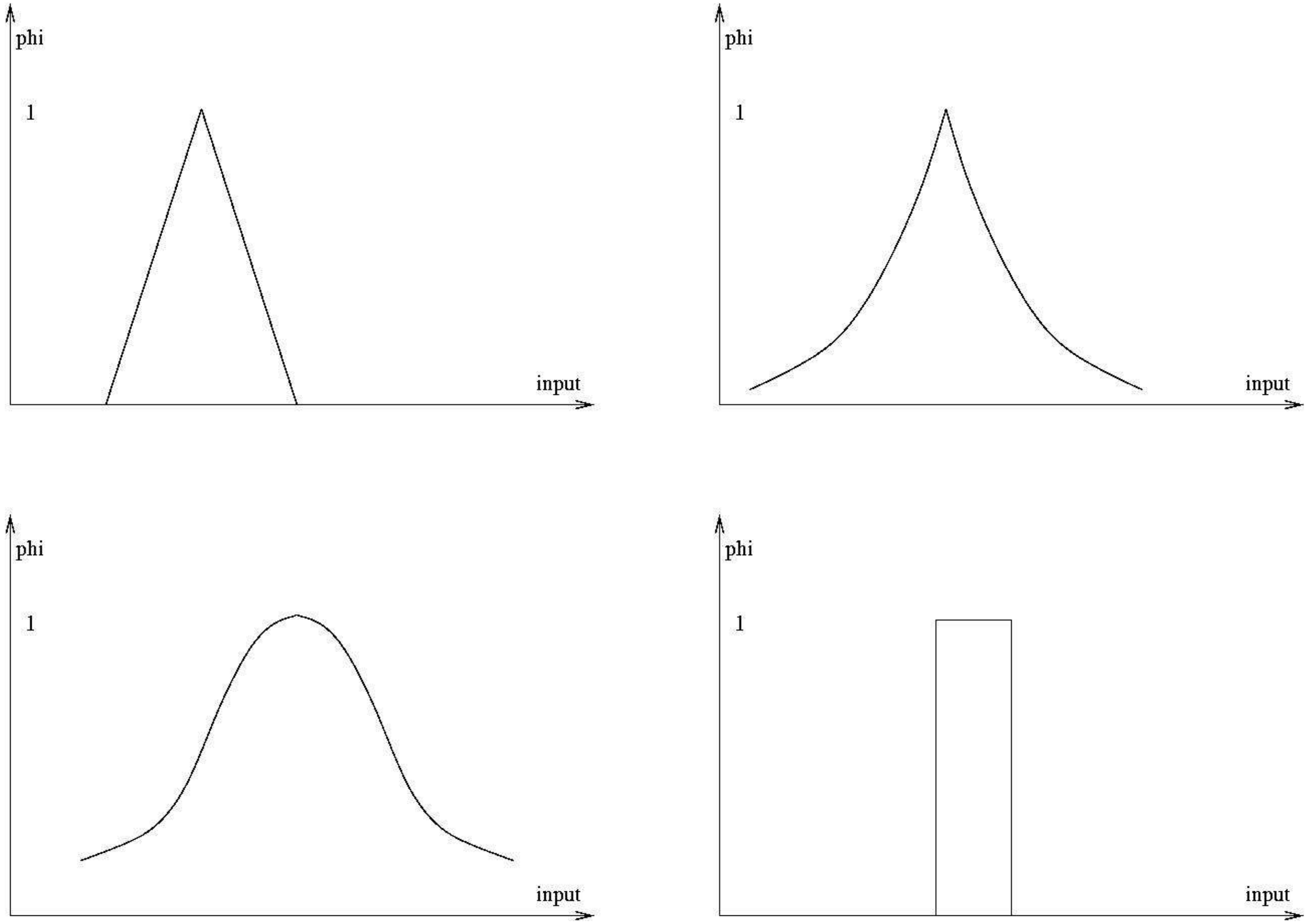
$$\phi_j(\mathbf{x}) = \exp\left(-\frac{\|\boldsymbol{x}-\boldsymbol{\mu}_j\|}{2\sigma_j^2}\right)$$

Can be augmented with neighborhood topologies as well!

Mixes unsupervised with supervised approaches

There are learning rules for the other parameters (sigma)…

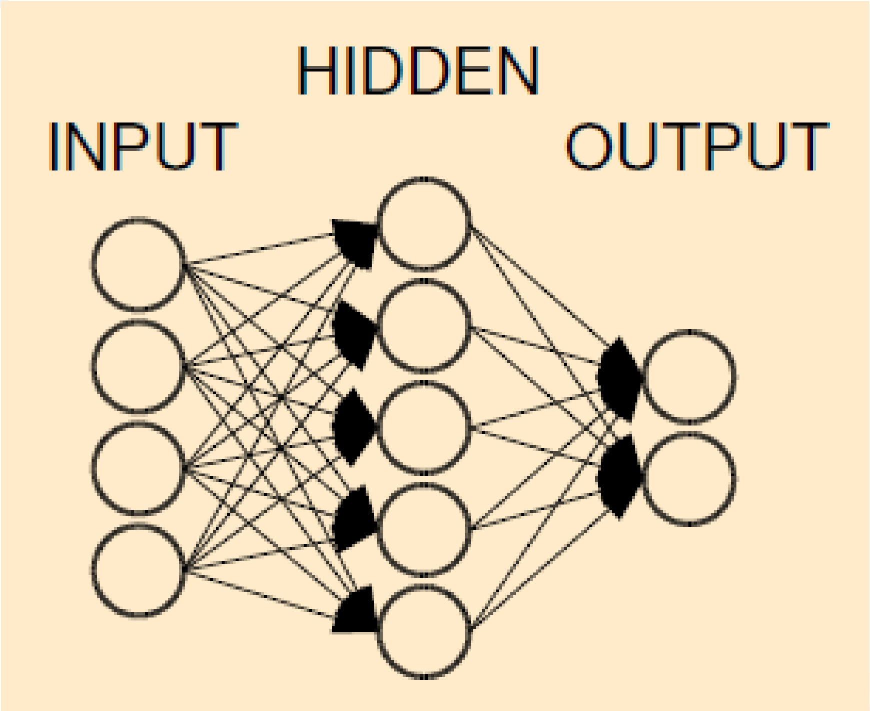
Interpolation – Radial Basis Functions



$$\phi_j(\boldsymbol{x}) = \phi(\|\boldsymbol{x} - \boldsymbol{\mu}_j\|)$$

Universal Approximation

- It has been proven that given a **two-layer network** and **enough *nonlinear* hidden units**, essentially *any* function can be approximated to an arbitrary degree of precision
- Hornik, Stinchcombe, and White, 1989
- A nonlinear activation function is required…



Can we obtain information about the error to help us?

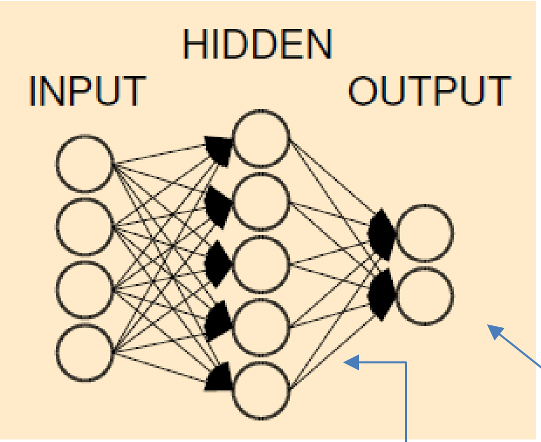
- As we saw before with the SSE of a single-layer network, the training data (input and output vectors) are *fixed.*
- Hidden unit activations are dependent on the data in the same way as output units.
- Thus, the weights are the only real parameters that have an influence on the activations!

$$\boldsymbol{y} = F(\boldsymbol{x}) \qquad\qquad \frac{\partial \boldsymbol{y}}{\partial w_{ij}} = \frac{\partial F(\boldsymbol{x})}{\partial w_{ij}}$$

It’s “complex” but not incomputable…

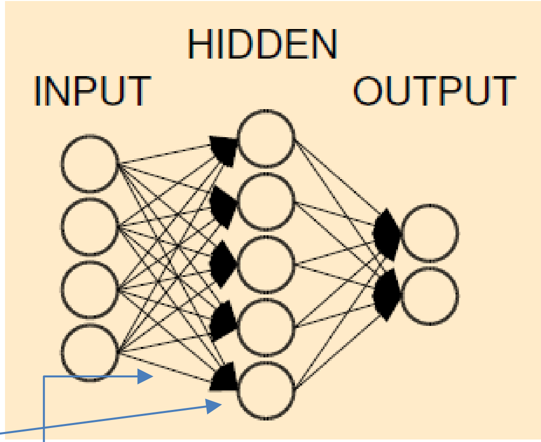
The chain rule makes it all possible…

$$\frac{\partial E}{\partial w_{ij}} = \frac{\partial}{\partial w_{ij}} \frac{1}{2} \sum_{k=1}^{c} (y_k - t_k)^2 = \sum_{k=1}^{c} (y_k - t_k) \frac{\partial y_k}{\partial w_{ij}}$$



We’ll derive it later, for now let’s just see the result…

$$\Delta w_{ij} \propto -\frac{\partial E}{\partial w_{ij}}$$



For output units…

$$\delta_j = g'(net_j)(a_j - t_j)$$

For hidden units…

$$\delta_j = g'(net_j) \sum_k w_{jk} \delta_k$$

$$\Delta w_{ij} \propto -\delta_j a_i$$

Note that these equations apply for any weight connecting unit *i* to unit *j*.

Generalized Delta Rule

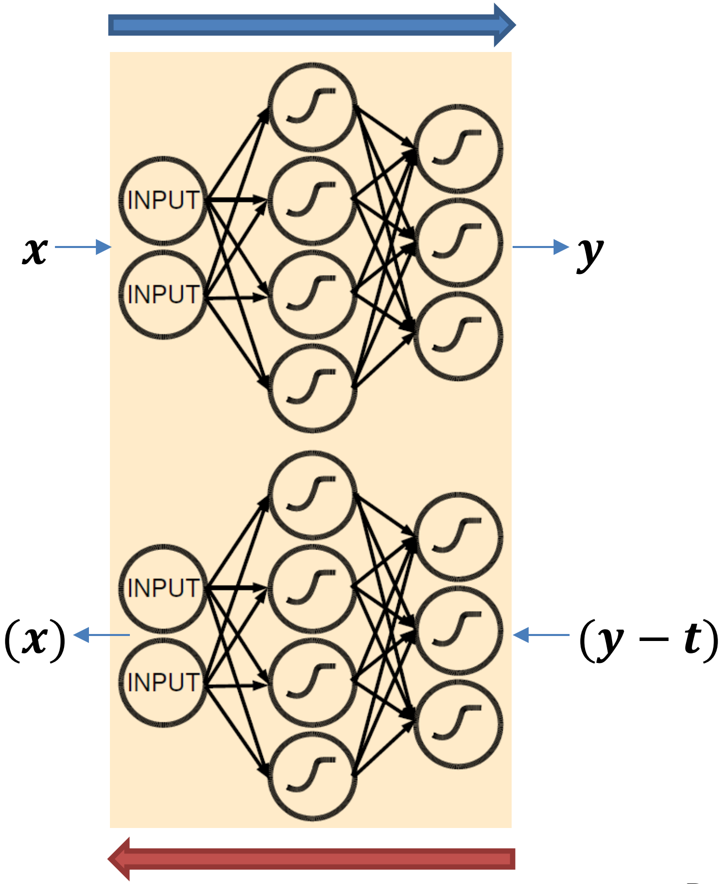
- Also known as **error backpropagation** or “**backprop**”
- Two step process:
  - **Prediction**: the inputs are provided and information flows to calculate the output activations (forward pass)
  - **Fitting**: errors are calculated at the output layer and information flows in reverse to calculate the weight updates (backward pass)
- Issue: biological plausibility
  - Real neurons do not pass information backward across synapses…

Output unit: $\delta_j = g'(net_j)(a_j - t_j)$

Hidden unit: $\delta_j = g'(net_j)\sum_k w_{jk}\delta_k$

$a_i = g(net_i) \qquad net_i = w_{i0} + \sum_j w_{ij}a_i$

Forward Pass (**W**)



Backward Pass (Δ**W**)

$\Delta w_{ij} \propto -\delta_j a_i$

Parallel Distributed Processing (Rumelhart and McClelland, 1986)

Backprop Derivation

- The Chain Rule is the Key

$$\frac{df}{dx} = \frac{df}{du}\frac{du}{dx}$$

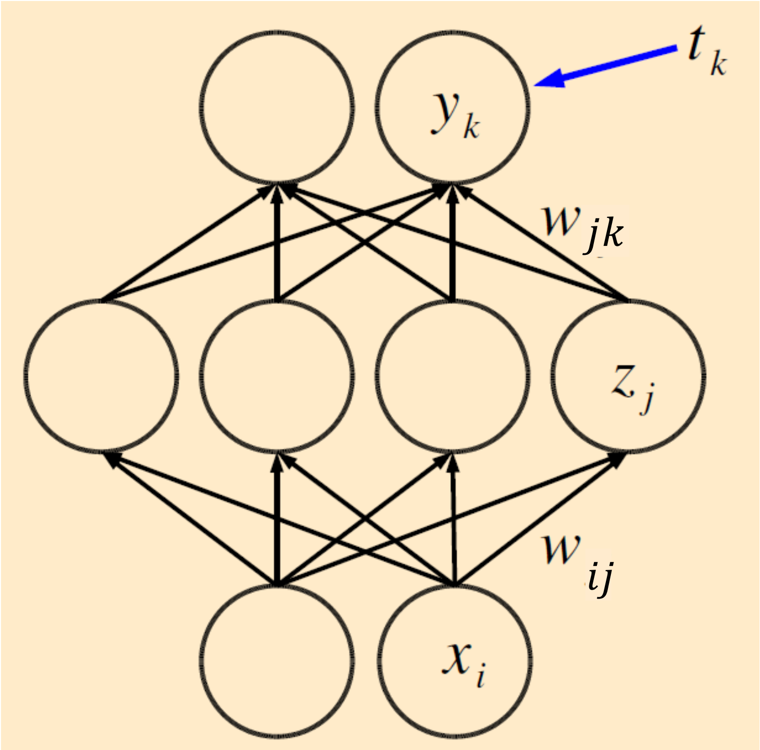
$$f(x) = f\big(g_1(x), g_2(x), \dots, g_n(x)\big)$$

$$\frac{\partial f}{\partial x} = \sum_{i=1}^{n} \frac{\partial f}{\partial g_i}\frac{\partial g_i}{\partial x}$$

$$E = \sum_{n=1}^{N} E^n$$

$$E^n = \frac{1}{2}\sum_{k=1}^{c}(y_k^n - t_k^n)^2$$

$$\boxed{\delta_j \equiv \frac{\partial E^n}{\partial net_j}}$$



C output units

M hidden units

D input units

Delta for an Output Unit

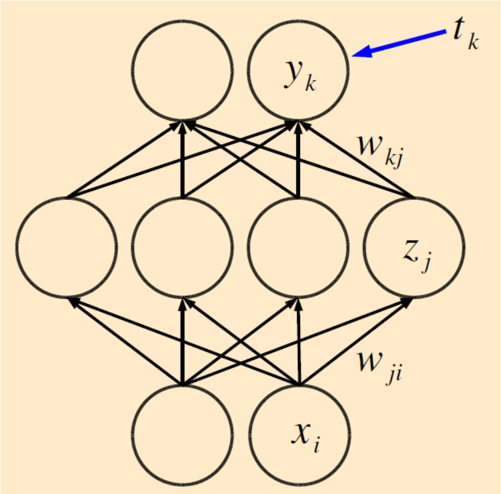
$$\delta_k = \frac{\partial E^n}{\partial net_k} = \frac{\partial}{\partial net_k}\left(\frac{1}{2}\sum_{c=1}^{C}(y_c^n - t_c^n)^2\right)$$

$$= \frac{1}{2}\frac{\partial}{\partial net_k}(y_k^n - t_k^n)^2$$

$$= (y_k^n - t_k^n)\frac{\partial}{\partial net_k}(y_k^n - t_k^n)$$

$$= (y_k^n - t_k^n)\frac{\partial y_j^n}{\partial net_k}$$

$$= (y_k^n - t_k^n)\frac{\partial g(net_k)}{\partial net_k} = (y_k^n - t_k^n)g'(net_k)$$



Rederived Delta Rule

$$-\frac{\partial E^n}{\partial w_{jk}} = -\frac{\partial E^n}{\partial net_k}\frac{\partial net_k}{\partial w_{jk}}$$

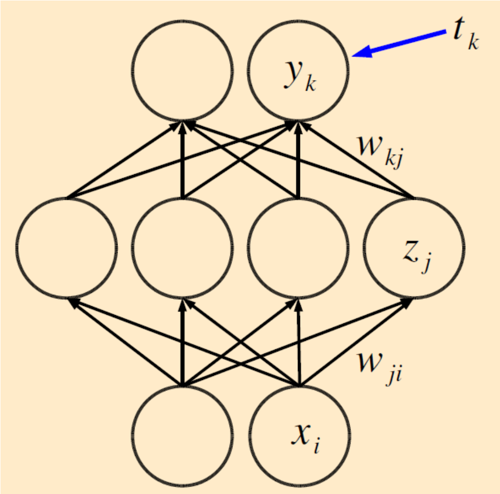
$$= -\delta_k \frac{\partial net_k}{\partial w_{jk}}$$

$$= -\delta_k \frac{\partial}{\partial w_{jk}}\left(\sum_{m=1}^{M} w_{mk} z_m^n\right)$$

$$= -\delta_k z_j^n$$

$$= -(y_k^n - t_k^n)g'(net_k)z_j^n$$

$$= -g'(net_k)(y_k^n - t_k^n)z_j^n$$



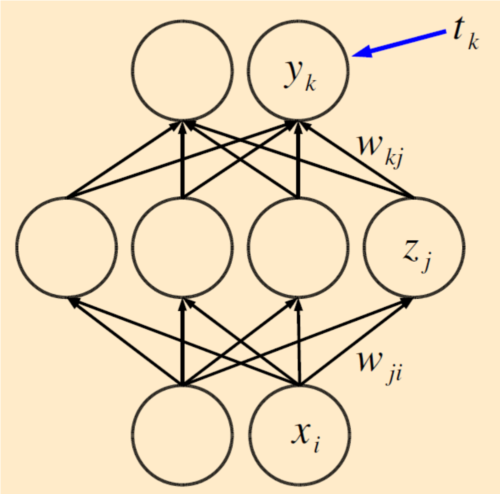
Gradient for a Hidden Unit

$$-\frac{\partial E^n}{\partial w_{ij}} = -\frac{\partial E^n}{\partial net_j}\frac{\partial net_j}{\partial w_{ij}}$$

$$= -\delta_j \frac{\partial}{\partial w_{ij}}\left(\sum_{d=1}^{D} w_{dj} x_d^n\right)$$

$$= -\delta_j x_d^n$$

OK, but what is the delta here?



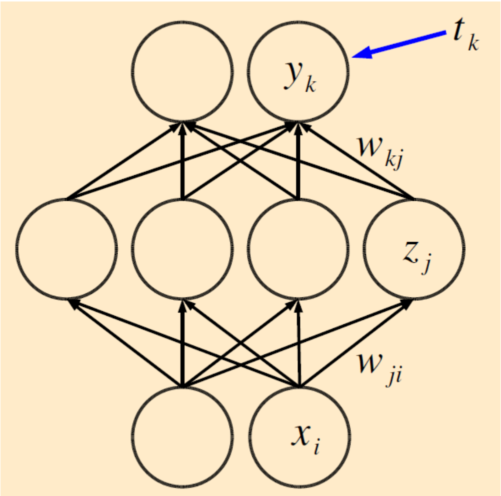
Delta for a Hidden Unit

$$\delta_j = \frac{\partial E^n}{\partial net_j} = \frac{\partial E^n}{\partial z_j^n}\frac{\partial z_j^n}{\partial net_j} = \frac{\partial E^n}{\partial z_j^n} g'(net_j)$$

$$= g'(net_j) \sum_{c=1}^{C} \frac{\partial E^n}{\partial net_c}\frac{\partial net_c}{\partial z_j^n}$$

$$= g'(net_j) \sum_{c=1}^{C} \delta_c \frac{\partial}{\partial z_j^n}\left(\sum_{m=1}^{M} w_{mc} z_m^n\right)$$

$$= g'(net_j) \sum_{c=1}^{C} \delta_c w_{jc}$$



So, now we can complete the gradient from before…

$$-\frac{\partial E^n}{\partial w_{ij}} = -\delta_j x_d^n$$

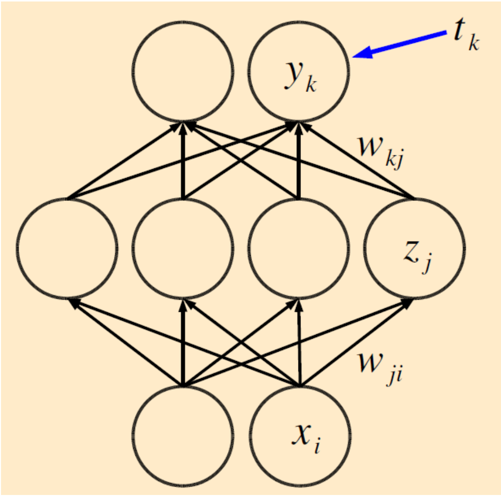
Again, it's "complex" but not incomputable!

The chain rule makes it all possible…

$$\frac{\partial E}{\partial w_{ij}} = \frac{\partial}{\partial w_{ij}} \frac{1}{2} \sum_{k=1}^{c} (y_k - t_k)^2 = \sum_{k=1}^{c} (y_k - t_k) \frac{\partial y_k}{\partial w_{ij}}$$

Chaining back the **deltas** through **weighted sums** works throughout the network!

$$\Delta w_{ij} \propto -\frac{\partial E}{\partial w_{ij}}$$



For output units…

$$\delta_j = g'(net_j)(a_j - t_j)$$

For hidden units…

$$\delta_j = g'(net_j) \sum_k w_{jk} \delta_k$$

$$\Delta w_{ij} \propto -\delta_j a_i$$

Note that these equations apply for any weight connecting unit *i* to unit *j*.

Generalized Delta Rule

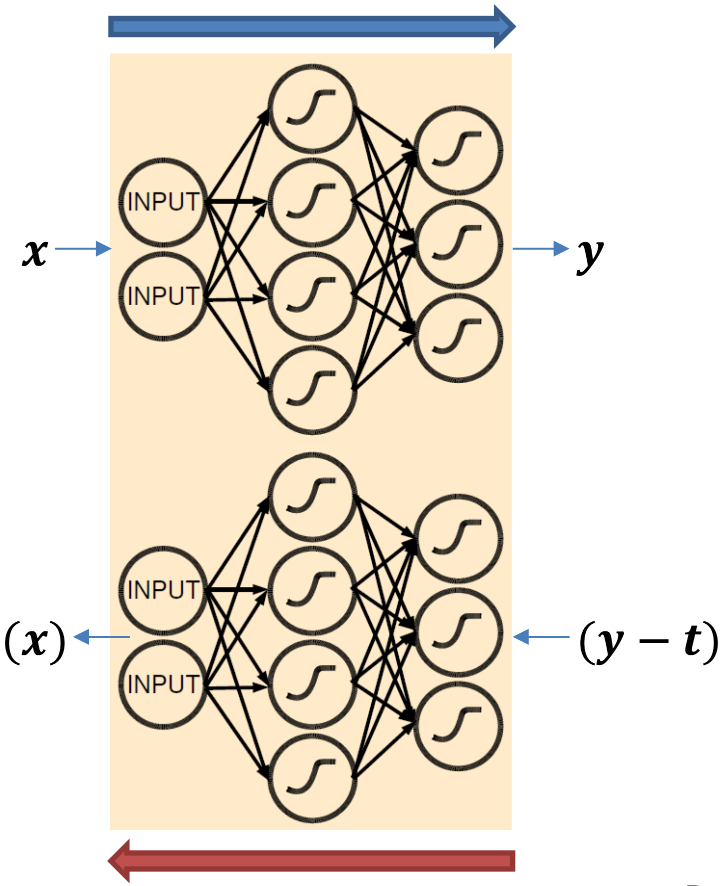
- Also known as **error backpropagation** or "**backprop**"
- Two step process:
  - **Prediction**: the inputs are provided and information flows to calculate the output activations (forward pass)
  - **Fitting**: errors are calculated at the output layer and information flows in reverse to calculate the weight updates (backward pass)
- Issue: biological plausibility
  - Real neurons do not pass information backward across synapses…

Output unit: $\delta_j = g'(net_j)(a_j - t_j)$

Hidden unit: $\delta_j = g'(net_j)\sum_k w_{jk}\delta_k$

$a_i = g(net_i) \qquad net_i = w_{i0} + \sum_j w_{ij}a_i$

Forward Pass (**W**)



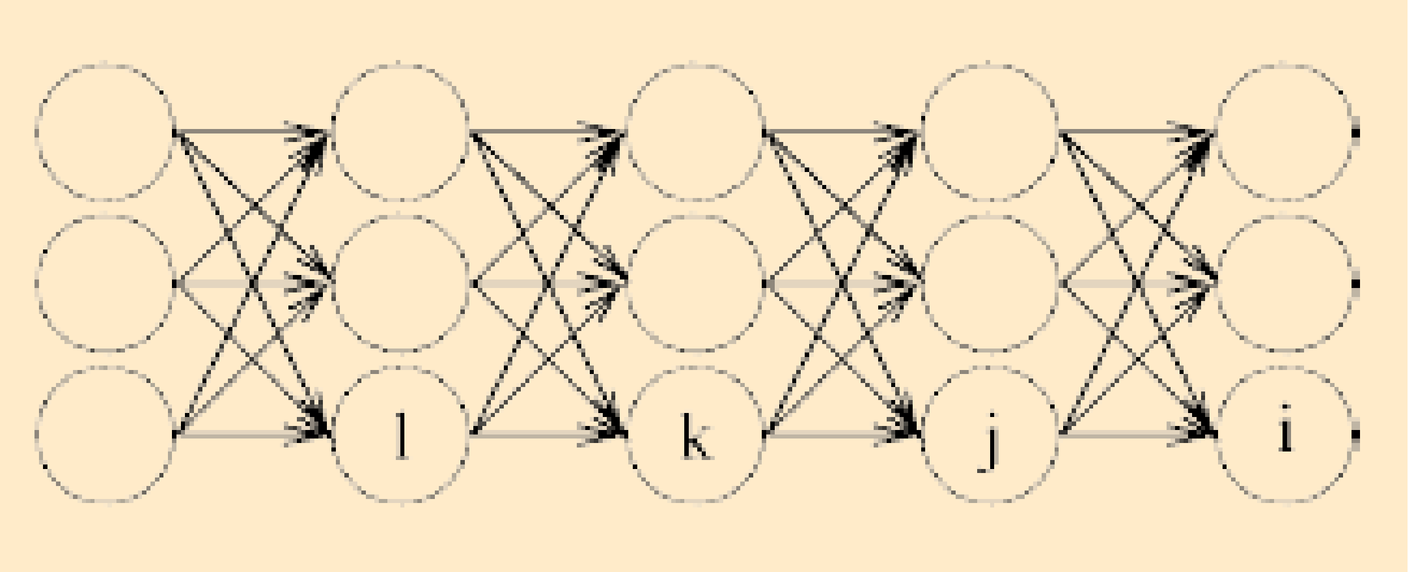
Backward Pass (Δ**W**)

$\Delta w_{ij} \propto -\delta_j a_i$

Parallel Distributed Processing (Rumelhart and McClelland, 1986)

Deep Learning?

- Generalized Delta Rule or Error Backpropagation
  - This algorithm scales to an arbitrary number of hidden layers
  - Maybe…



$$\delta_j = g'(net_j)\sum_i w_{ij}\delta_i$$

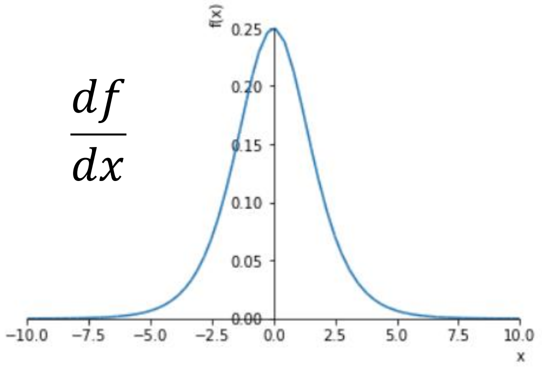
$$\delta_k = g'(net_k)\sum_j w_{jk}\delta_j$$

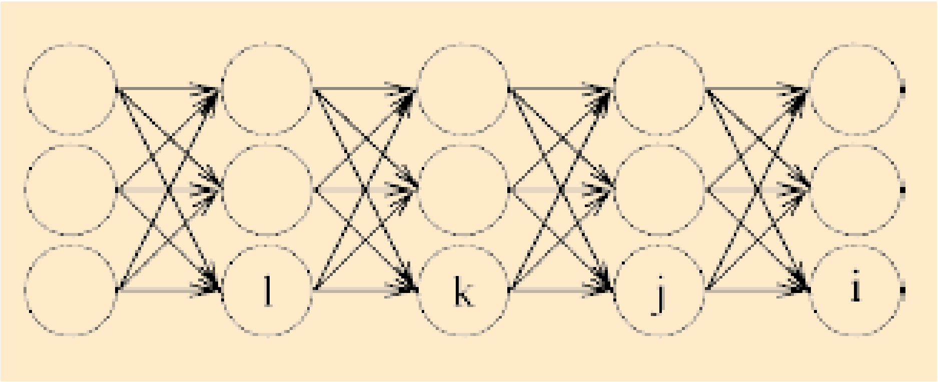
$$\delta_l = g'(net_l)\sum_k w_{kl}\delta_k$$

The Vanishing Gradient Problem

- Getting caught in the flat, planar regions of the error surface is problematic
- We **need** non-linear activation functions to make use of multiple layers, but typical non-linear activation functions (sigmoid, tanh) cause the gradient to **vanish**…

$$\frac{df}{dx}$$





$$\delta_j = g'(net_j)\sum_i w_{ij}\delta_i$$

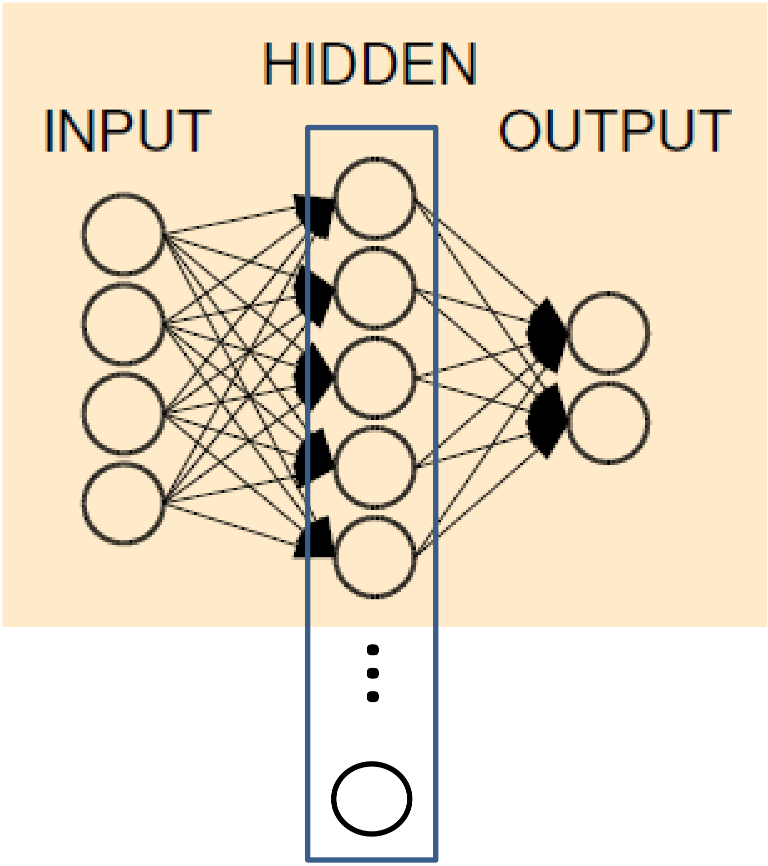
$$\delta_k = g'(net_k)\sum_j w_{jk}\delta_j$$

$$\delta_l = g'(net_l)\sum_k w_{kl}\delta_k$$

- We will see some workarounds to this *particular* issue soon, but even still, gradient optimization is tricky…

So is wider better?

- Given the practical problems with the gradient disappearing the “wider is better” trend existed for over a decade without any clear resolution...
- Potential problems
  - (Major issue) Complex decision boundaries are probably better described by multilayered partitions instead of a single, very complex partition: might even be easier to learn this way if we could…
  - (Minor issue) Wider networks allow for poorer parallel training and performance since we can’t process more than two batches of patterns through a feed-forward network at any one time



Encodings Matter

- Let's reencode the XOR problem vectors…
  - Let each value (0 or 1) be replaced with a corresponding vector
    - zero=[0 1]
    - one=[1 0]
  - For each pattern:
    - compute the *outer product* of the value vector for column 1 and the value vector for column 2
    - Flatten into a 4 element vector
  - Combine vectors into new input pattern matrix…
- Even Hebbian learning can solve this!

$$\begin{bmatrix} 0 & 0 \\ 0 & 1 \\ 1 & 0 \\ 1 & 1 \end{bmatrix} = \begin{bmatrix} 0 \\ 1 \\ 1 \\ 0 \end{bmatrix}$$

First pattern is [0 0], so use two vectors: [0 1] and [0 1]

$$\begin{array}{c|cc} \times & 0 & 1 \\ \hline 0 & \multicolumn{2}{c}{\multirow{2}{*}{$\begin{bmatrix} 0 & 0 \\ 0 & 1 \end{bmatrix}$}} \\ 1 & & \end{array}$$

Flatten into a four element vector…

$$\begin{bmatrix} 0 & 0 & 0 & 1 \end{bmatrix}$$

Do the same for all four:

$$\begin{bmatrix} 0 & 0 & 0 & 1 \\ 0 & 0 & 1 & 0 \\ 0 & 1 & 0 & 0 \\ 1 & 0 & 0 & 0 \end{bmatrix} = \begin{bmatrix} 0 \\ 1 \\ 1 \\ 0 \end{bmatrix}$$

Higher-dimensional Projections

- **Projecting data into higher dimensions** makes *decision boundaries* (or functional shapes for regressions) *easier to obtain*…
- Generally, the more units used, the more complex this projection *may* be
- However, this *kind* of solution is limited in the sense that it has to be performed in this specific way (no **nested, repeated, functional, or relational** solutions…)