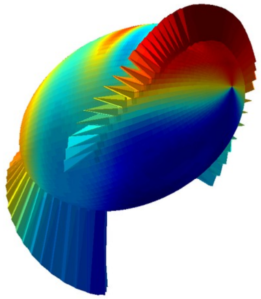
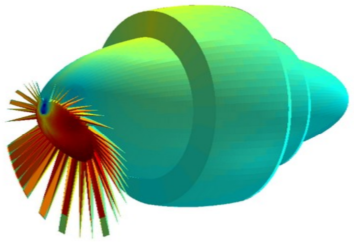
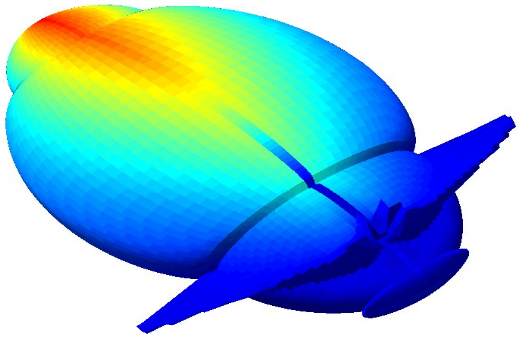
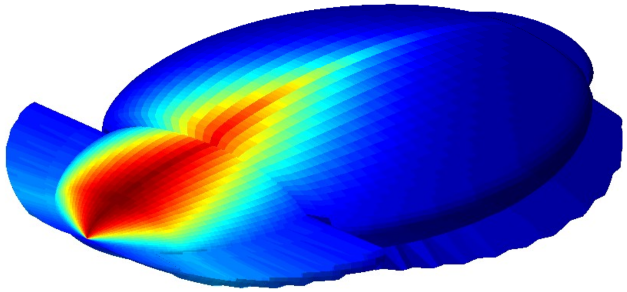
UNIVERSITY OF Cincinnati

College of Engineering & Applied Science

Complex Adaptive Systems Laboratory

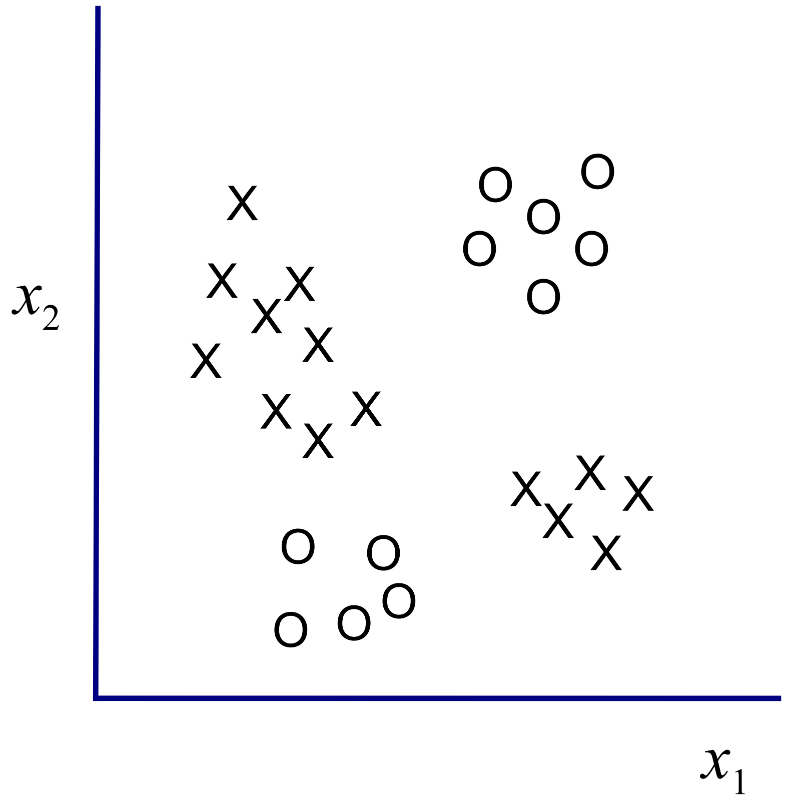



Lecture 7
Multi-Layer Networks & Backpropagation

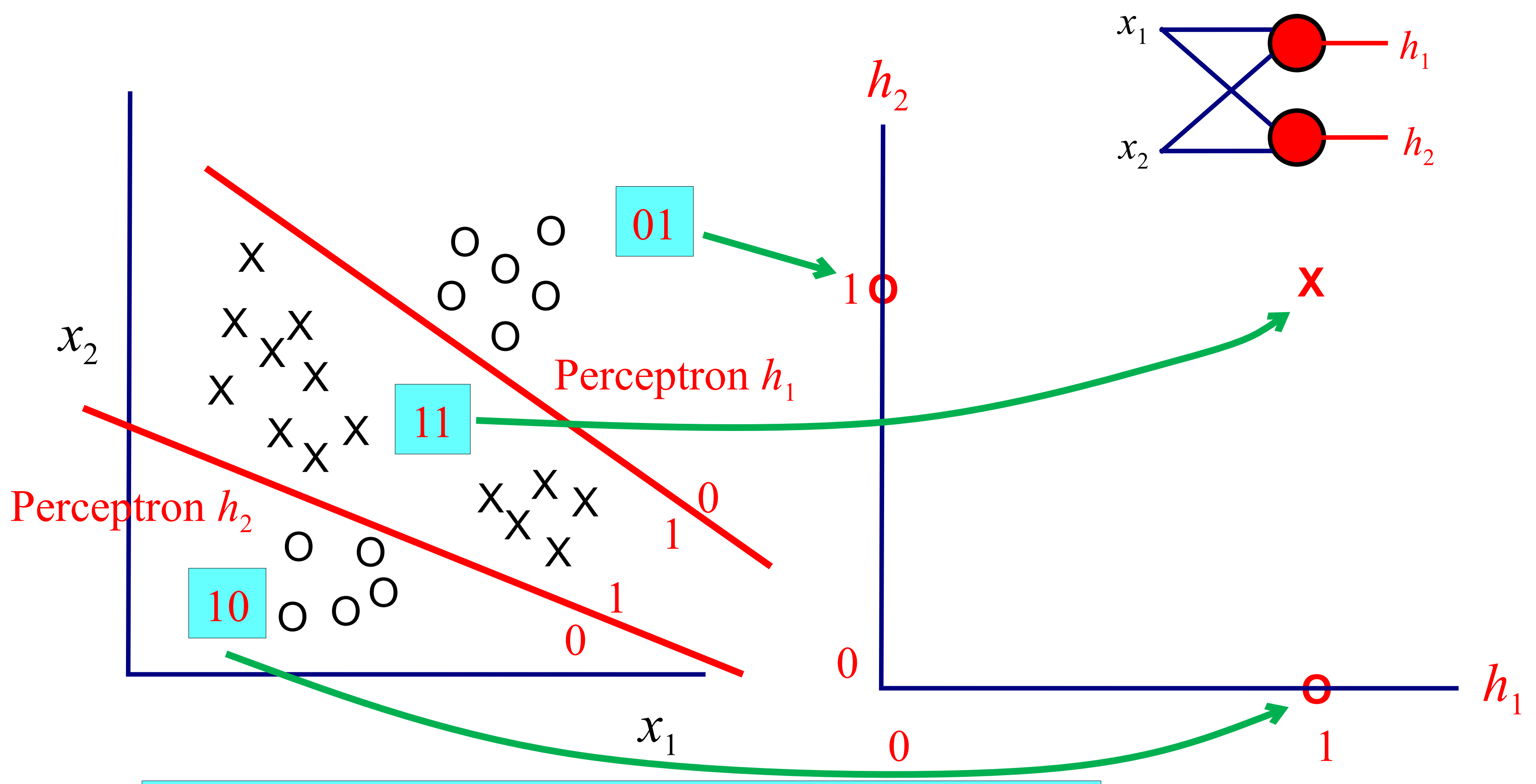



Multilayer Perceptrons

Single perceptrons can only learn linear decision boundaries, so these classes cannot be separated.

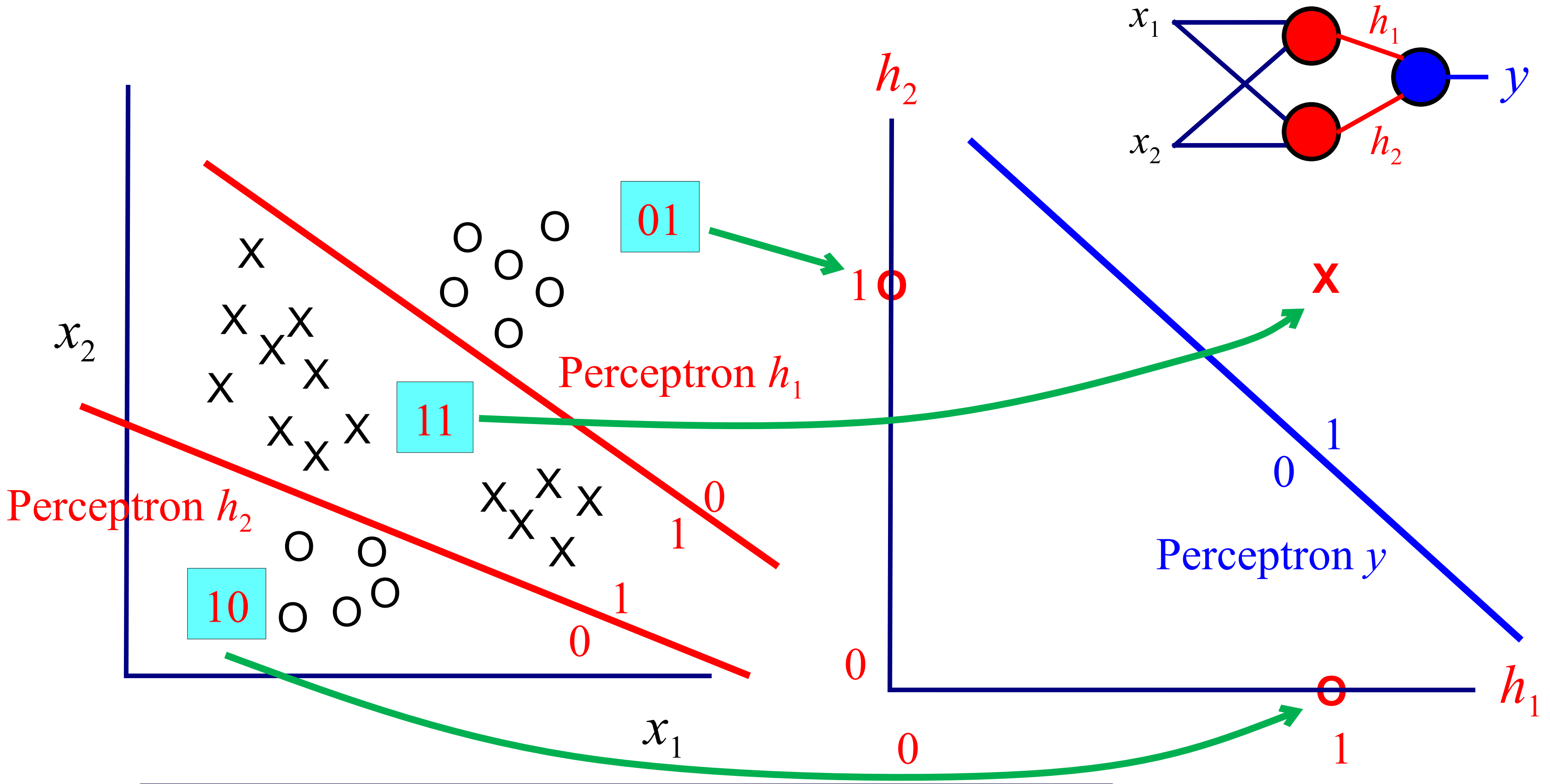


But two perceptrons can draw two linear decision boundaries to re-map the data to a new (p_1, p_2) space



But how can the weights for the network be found?

…. and a third perceptron in the next layer can do the classification.



But how can the weights for the network be found?

The Backpropagation Algorithm

- Based on gradient descent.

- Allows training of any feedforward network

- Can be used to train recurrent networks with minor modification.

  - "Discovered" in various forms by:

Bryson and Ho	1969
Werbos	1974
Parker	1985

The familiar neural network formulation by Rumelhart, Hinton & Williams, 1985

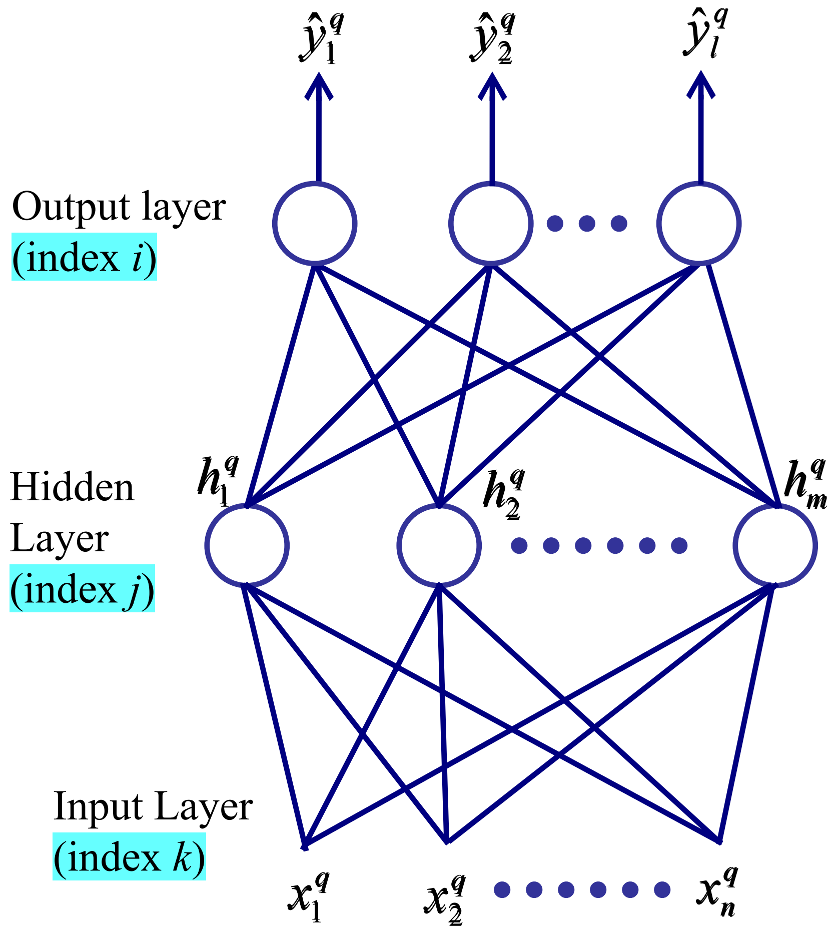
Why is backpropagation important?

Multilayer feedforward networks can classify any data and approximate any function

$\downarrow$

A general training algorithm for such networks is very useful

The Backpropagation Algorithm



q = pattern index $\left[\text{data-point } q = (x^q, y^q)\right]$

n = # of inputs (input space dimension)

m = # of hidden neurons (hidden space dim.)

l = # of outputs (output space dimension)

w_{ij}= weight from hidden neuron j to output i

w_{jk}= weight from input k to hidden neuron j

$$s_j^q = \sum_{k=0}^{n} w_{jk} x_k^q$$

$$h_j^q = f_j\left(s_j^q\right)$$

Hidden Layer

$$s_i^q = \sum_{j=0}^{m} w_{ij} h_j^q$$

$$\hat{y}_i^q = f_i\left(s_i^q\right)$$

Output Layer

$$J^q = \frac{1}{2}\sum_{i=1}^{l}\left(y_i^q - \hat{y}_i^q\right)^2$$

$$= \frac{1}{2}\sum_{i=1}^{l}\left(e_i^q\right)^2$$

Loss function

$$J = \sum_{q=1}^{N} J^q \quad \rightarrow \quad \text{learn by } \Delta\bar{w} = -\eta \nabla J(\bar{w})$$

$\bar{w} \equiv$ vector of all weights

$$\Delta w_{ij} = -\eta_o \frac{\partial J^q}{\partial w_{ij}}$$

$$\Delta w_{jk} = -\eta_h \frac{\partial J^q}{\partial w_{jk}}$$

$$s_j^q = \sum_{k=0}^{n} w_{jk} x_k^q$$

$$h_j^q = f_j\left(s_j^q\right)$$

Hidden Layer

$$s_i^q = \sum_{j=0}^{m} w_{ij} h_j^q$$

$$\hat{y}_i^q = f_i\left(s_i^q\right)$$

Output Layer

$$J^q = \frac{1}{2}\sum_{i=1}^{l}\left(y_i^q - \hat{y}_i^q\right)^2$$

$$= \frac{1}{2}\sum_{i=1}^{l}\left(e_i^q\right)^2$$

Loss function

$$J = \sum_{q=1}^{N} J^q \quad \rightarrow \quad \text{learn by } \Delta\bar{w} = -\eta \nabla J(\bar{w})$$

$\bar{w} \equiv$ vector of all weights

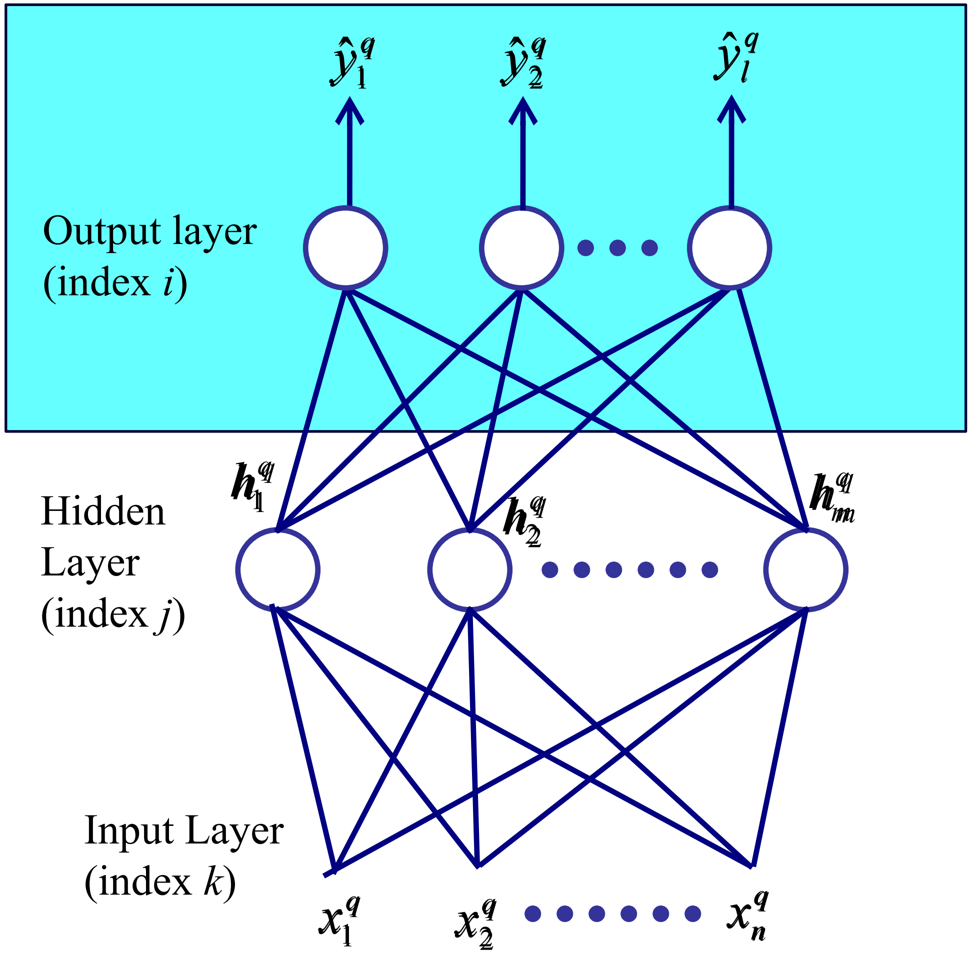
$$\Delta w_{ij} = -\eta_o \frac{\partial J^q}{\partial w_{ij}}$$

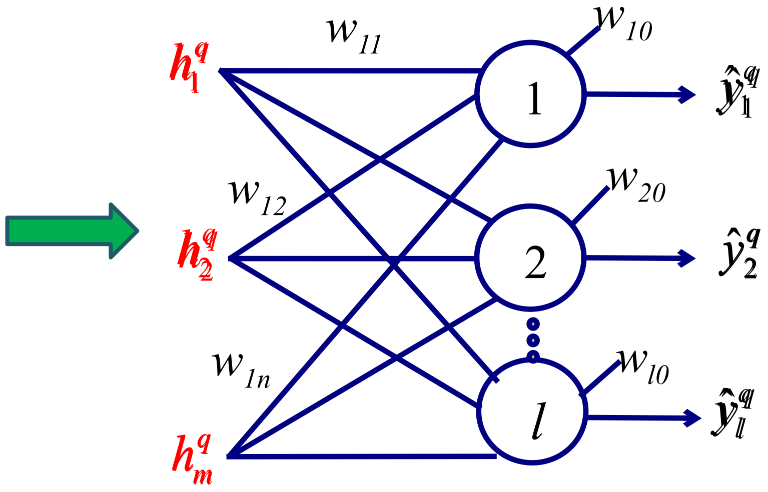
This is easy to calculate using LMS

$$\Delta w_{jk} = -\eta_h \frac{\partial J^q}{\partial w_{jk}}$$

But how to calculate this?

Learning for the Output Layer





$$\frac{\partial J^q}{\partial w_{ij}} = -e_i^q f'(s_i^q) h_j^q$$

$$= -(y_i^q - \hat{y}_i^q) f'(s_i^q) h_j^q$$

LMS

$$\Delta w_{ij} = -\eta \frac{\partial J^q}{\partial w_{ij}} = \eta (y_i^q - \hat{y}_i^q) f'(s_i^q) h_j^q$$

To see this repeat the LMS calculation for $\frac{\partial J^q}{\partial w_{ij}}$:

$$\frac{\partial J^q}{\partial w_{ij}} = \frac{\partial J^q}{\partial e_i^q} \cdot \frac{\partial e_i^q}{\partial \hat{y}_i^q} \cdot \frac{\partial \hat{y}_i^q}{\partial s_i^q} \cdot \frac{\partial s_i^q}{\partial w_{ij}}$$

$$= (e_i^q) \cdot (-1) \cdot (f_i'(s_i^q)) \cdot (h_j^q)$$

$$\equiv - e_i^q \cdot f_i'(s_i^q) \cdot h_j^q$$

where $f'(u) \equiv \left.\frac{df}{du}\right|_u$

If $f(u) = \frac{1}{1+e^{-u}}$

$$f'(u) = \frac{1}{1+e^{-u}} \frac{e^{-u}}{1+e^{-u}} \equiv f(u)(1 - f(u))$$

$$\boxed{\frac{\partial J^q}{\partial w_{ij}} = - \underbrace{(y_i^q - \hat{y}_i^q) f_i'(s_i^q)}_{\equiv \delta_i^q} h_j^q} \quad (1)$$

$$= - \delta_i^q h_j^q$$

$$\therefore \quad w_{ij} = w_{ij} + \eta_o \delta_i^q h_j^q$$

or $\boxed{\Delta w_{ij} = \eta_o \delta_i^q h_j^q}$ (2)

Learning for the Hidden Layer

Calculating $\dfrac{\partial J^q}{\partial w_{jk}}$:

Note that

$$J^q = \frac{1}{2}\sum_{i=1}^{l}\left[y_i^q - \hat{y}_i^q\right]^2 = \frac{1}{2}\sum_{i=1}^{l}\left(e_i^q\right)^2 \qquad (3)$$

$$= \frac{1}{2}\sum_{i=1}^{l}\left\{y_i^q - f_i\left[\sum_j w_{ij} f_j\underbrace{\left(\sum_k w_{jk} x_k^q\right)}_{h_j^q}\right]\right\}^2$$

$$\frac{\partial J^q}{\partial w_{jk}} = \frac{\partial}{\partial w_{jk}}\left[\frac{1}{2}\sum_i \left(e_i^q\right)^2\right]$$

$$= \frac{1}{2}\sum_{i=1}^{l}\frac{\partial}{\partial w_{jk}}\left(e_i^q\right)^2$$

$$= \frac{1}{2}\sum_i \frac{\partial\left(e_i^q\right)^2}{\partial e_i^q}\cdot\frac{\partial e_i^q}{\partial w_{jk}}$$

$$\boxed{\frac{\partial J^q}{\partial w_{jk}} = \sum_i e_i^q\cdot\frac{\partial e_i^q}{\partial w_{jk}}} \qquad (4)$$

$$\frac{\partial e_i^q}{\partial w_{jk}} = \frac{\partial}{\partial w_{jk}}\left(y_i^q - \hat{y}_i^q\right)$$

$$= -\frac{\partial \hat{y}_i^q}{\partial w_{jk}} \quad (\text{since } y_i^q \text{ is fixed})$$

$$= -\frac{\partial \hat{y}_i^q}{\partial s_i^q}\cdot\frac{\partial s_i^q}{\partial w_{jk}} = -f_i'\left(s_i^q\right)\frac{\partial s_i^q}{\partial w_{jk}}$$

$$\boxed{\frac{\partial e_i^q}{\partial w_{jk}} = -f_i'\left(s_i^q\right)\frac{\partial s_i^q}{\partial w_{jk}}} \qquad \textcircled{5}$$

$$\frac{\partial s_i^q}{\partial w_{jk}} = \frac{\partial}{\partial w_{jk}}\sum_{j'=0}^{m} w_{ij'}h_{j'}^q$$

$$= \sum_{j'}\frac{\partial}{\partial w_{jk}} w_{ij'}h_{j'}^q$$

$$\boxed{\frac{\partial s_i^q}{\partial w_{jk}} = \sum_{j'} w_{ij'}\frac{\partial h_{j'}^q}{\partial w_{jk}}} \qquad \textcircled{6}$$

$j' \in$ Hidden Layer

Note : we use j' because j is already used in w_{jk}

$$\frac{\partial s_i^q}{\partial w_{jk}} = \sum_{j'} w_{ij'} \frac{\partial h_{j'}^q}{\partial w_{jk}} \qquad (6)$$

$$\frac{\partial h_{j'}^q}{\partial w_{jk}} = \frac{\partial h_{j'}^q}{\partial s_{j'}^q} \frac{\partial s_{j'}^q}{\partial w_{jk}}$$

Since w_{jk} has no effect on $s_{j'}^q$ if $j \neq j'$

$$= \begin{cases} f_j'(s_j^q) x_k^q & \text{if } j' = j \\ 0 & \text{if } j' \neq j \end{cases}$$

$$\therefore \boxed{\frac{\partial h_{j'}^q}{\partial w_{jk}} = \begin{cases} f_j'(s_j^q) x_k^q & \text{if } j' = j \\ 0 & \text{if } j' \neq j \end{cases}} \qquad (7)$$

$$\textcircled{7} \rightarrow \textcircled{6}$$

$$\boxed{\frac{\partial s_i^q}{\partial w_{jk}} = w_{ij} f_j'(s_j^q) x_k^q} \qquad (8)$$

$$\textcircled{8} \rightarrow \textcircled{5}$$

$$\boxed{\frac{\partial e_i^q}{\partial w_{jk}} = - f_i'(s_i^q) w_{ij} f_j'(s_j^q) x_k^q} \qquad (9)$$

$$\begin{tikzcd} \textcircled{9} \arrow[r] & \textcircled{4} \end{tikzcd}$$

$$\frac{\partial J^q}{\partial w_{jk}} = -\sum_i w_{ij} \overbrace{e_i^q f_i'(s_i^q)}^{\delta_i^q} f_j'(s_j^q) x_k^q$$

$$\boxed{= -\sum_i w_{ij}\delta_i^q f_j'(s_j^q) x_k^q} \qquad \textcircled{10}$$

$$\frac{\partial J^q}{\partial w_{jk}} = -\underbrace{\left[f_j'(s_j^q) \sum_i w_{ij}\delta_i^q \right]}_{\delta_j^q} x_k^q$$

$$\therefore \quad \boxed{\frac{\partial J^q}{\partial w_{jk}} = -\delta_j^q x_k^q}$$

where $\delta_j^q = f_j'(s_j^q) \sum_i w_{ij}\delta_i^q$

and

$$w_{jk} = w_{jk} + \eta_h \delta_j^q x_k^q$$

$$\boxed{\Delta w_{jk} = \eta_h \delta_j^q x_k^q} \qquad \textcircled{11}$$

Gradient Calculation Process

1 $$\frac{\partial J^q}{\partial w_{ij}} = - \underbrace{f_i'(s_i^q)}\underbrace{\left[y_i^q - \hat{y}_i^q\right]}\underbrace{h_j^q} \longleftarrow$$ same as LMS

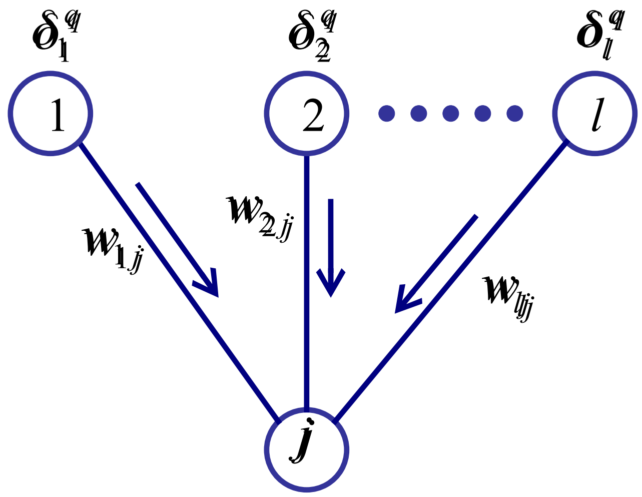
10 $$\frac{\partial J^q}{\partial w_{jk}} = - \overbrace{f_j'(s_j^q)}\overbrace{\sum_i w_{ij}\delta_i^q}\overbrace{x_k^q} \longleftarrow$$ same form as 1

$$\text{(1)}\quad \frac{\partial J^q}{\partial w_{ij}} = -\underbrace{\boxed{f_i'(s_i^q)\left[y_i^q - \hat{y}_i^q\right]}}_{}\, h_j^q, \qquad \boxed{\cdot} = \delta_i^q$$

$$\text{(10)}\quad \frac{\partial J^q}{\partial w_{jk}} = -\boxed{f_j'(s_j^q)\sum_i w_{ij}\delta_i^q}\, x_k^q, \qquad \boxed{\cdot} = \delta_j^q$$

-Signals flow forward
-Errors (δ's) propagate backwards

Note how δ_j^q for $j \in$ hidden layer is calculated recursively



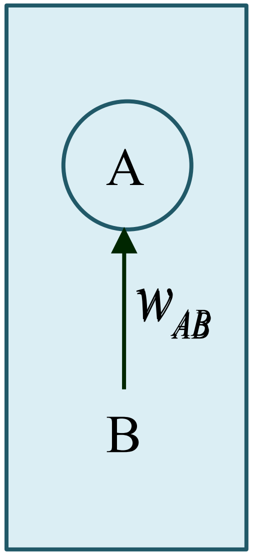
$$\delta_j^q = f_j'(s_j^q)\underbrace{\sum_{i=1}^{l} w_{ij}\delta_i^q}_{}$$

can be calculated at j if δ_i^q are known

Form of the Learning Rule

Compare $\bigcirc\!\!\!\!2$ and $\bigcirc\!\!\!\!\!\!11$. Both have the form

$$\boxed{\Delta w_{AB} = \eta \cdot \delta_A^q \cdot \text{(input on line B)}}$$



if $A \in$ output layer

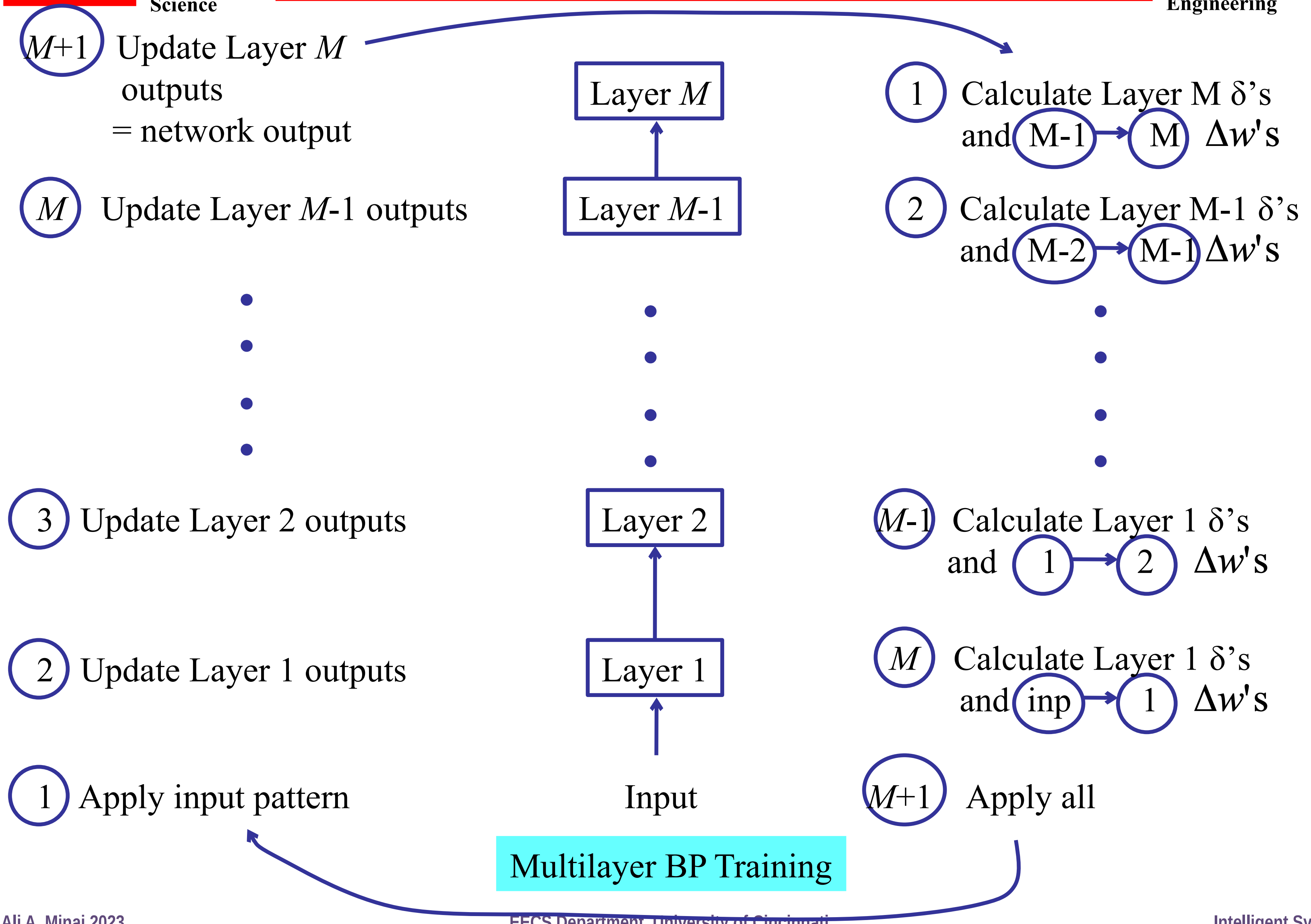
$$\delta_A^q = f_A'(s_A^q)\left[y_A^q - \hat{y}_A^q\right]$$

if $A \in$ hidden layer

$$\delta_A^q = f_A'(s_A^q)\sum_r w_{rA}\delta_r^q$$

where *r* indexes the neurons ***to*** which A connects.

Multilayer BP Training


M+1 Update Layer M outputs = network output
M Update Layer M-1 outputs
3 Update Layer 2 outputs
2 Update Layer 1 outputs
1 Apply input pattern
Layer M
Layer M-1
Layer 2
Layer 1
Input
1 Calculate Layer M δ's and M-1 → M Δw's
2 Calculate Layer M-1 δ's and M-2 → M-1 Δw's
M-1 Calculate Layer 1 δ's and 1 → 2 Δw's
M Calculate Layer 1 δ's and inp → 1 Δw's
M+1 Apply all

A General Formulation

Networks may use:

- Different loss functions.
- Different activation functions (even different between layers or within a layer.)

We can define a general formulation to handle this under the assumption that:

$$J^q = \sum_i J_i^q$$ i.e., total loss is added over the output neurons.

For output layer weights:

$$\frac{\partial J^q}{\partial w_{ij}} = \frac{\partial J^q}{\partial \hat{y}_i^q}\frac{\partial \hat{y}_i^q}{\partial s_i^q}\frac{\partial s_i^q}{\partial w_{ij}} = \underbrace{\boxed{\frac{\partial J_i^q}{\partial \hat{y}_i^q}\frac{\partial \hat{y}_i^q}{\partial s_i^q}}}_{-\delta_i^q}\underbrace{\boxed{\frac{\partial s_i^q}{\partial w_{ij}}}}_{h_j^q}$$

Mean-squared loss function
Linear composition function

For the hidden layer weights:

$$\frac{\partial J^q}{\partial w_{jk}} = \sum_i \frac{\partial J_i^q}{\partial \hat{y}_i^q} \frac{\partial \hat{y}_i^q}{\partial s_i^q} \frac{\partial s_i^q}{\partial h_j^q} \frac{\partial h_j^q}{\partial s_j^q} \frac{\partial s_j^q}{\partial w_{jk}}$$

$$= \left[\sum_i \frac{\partial J_i^q}{\partial \hat{y}_i^q} \frac{\partial \hat{y}_i^q}{\partial s_i^q} \frac{\partial s_i^q}{\partial h_j^q} \right] \left[\frac{\partial h_j^q}{\partial s_j^q} \right] \left[\frac{\partial s_j^q}{\partial w_{jk}} \right]$$

For the hidden layer weights:

$$\frac{\partial J^q}{\partial w_{jk}} = \sum_i \frac{\partial J_i^q}{\partial \hat{y}_i^q} \frac{\partial \hat{y}_i^q}{\partial s_i^q} \frac{\partial s_i^q}{\partial h_j^q} \frac{\partial h_j^q}{\partial s_j^q} \frac{\partial s_j^q}{\partial w_{jk}}$$

$$= \left[\sum_i \underbrace{\boxed{\frac{\partial J_i^q}{\partial \hat{y}_i^q} \frac{\partial \hat{y}_i^q}{\partial s_i^q}}}_{\boxed{-\delta_i^q}} \underbrace{\boxed{\frac{\partial s_i^q}{\partial h_j^q}}}_{\boxed{w_{ij}}} \right] \left[\underbrace{\boxed{\frac{\partial h_j^q}{\partial s_j^q}}}_{\boxed{f_j'(s_j^q)}} \right] \left[\underbrace{\boxed{\frac{\partial s_j^q}{\partial w_{jk}}}}_{\boxed{x_k^q}} \right]$$

$$-\delta_i^q = \boxed{-(y_i^q - \hat{y}_i^q) f_i'(s_i^q)}$$

$$\frac{\partial J^q}{\partial w_{jk}} = -f_j'(s_j^q) \sum_i w_{ij} \delta_i^q \, x_k^q$$

Mean-squared loss function
Linear composition function

Practical Choices

Using Backpropagation for Classification Problems

Backpropagation requires that $\frac{\partial J^q}{\partial w_{jk}}$ exists $\forall w_{ij}$

$\Rightarrow f_i'(s_i^q)$ must exist

$\Rightarrow f(u)$ (activation function) must be differentiable

$$\rightarrow \text{use } f(u) = \frac{1}{1+e^{-u}}$$

$$\text{or } f(u) = \tanh(u)$$

instead of hard threshold.

But classification requires that output neurons have 0/1 or +1/-1 output

?????????

Possible Solutions

- Use 0/1 or +1/-1 as target values y_i^q

- Set operating parameters *H*, *L*, *L*<*H* such that:

$\hat{y}_i \geq H$ is considered a match for $y_i = +1$

$\hat{y}_i \leq L$ is considered a match for $y_i = 0$ or -1

- During training, use 0/1 or +1/-1 as targets but if $\hat{y}_{ii}$ matches the operating targets, make no weight change

Typical values for *H*, *L* are:

H=0.75	*L*=0.25	for 0-1 sigmoid
H=0.75	*L*=-0.75	for +/- sigmoid

Multi-Class Classification

If there are more than 2 classes, how should the output be represented?

Solution:

- If there are *M* classes, have *M* neurons in the output layer.
- During training, use one-hot codes as targets for each class, e.g. if $M = 3$,
  Class 1 = [1 0 0]   Class 2 = [0 1 0]   Class 3 = [0 0 1]
- After training, for a test input *x*, choose the class as the output neuron with the highest output, e.g., Output = [0.1 0.6 0.2] → [0 1 0]
- If "no choice" is allowed, require that an output must be higher than some threshold θ to be considered 1.

A better solution: softmax classifiers

Data Scaling

Given input vectors $\bar{x} = \begin{bmatrix} x_1 & x_2 & . & . & . & x_n \end{bmatrix}^T$

Scale each x_k such that it is 0-mean and has approximately the same range as the other components. Scaling between -1 and +1 is usually a good idea.

Note: if $x_k \geq 0$ in all cases

$$\Delta w_{jk} = \eta \delta_j x_k$$

$\Rightarrow$ All weights for hidden neuron *j* either increase together or decrease together

$\rightarrow$ less discrimination in each training step

$\rightarrow$ slower training

Data Scaling

In binary input situations, use +/- rather than 0/1:

Recall that $\boxed{\text{inp}} \rightarrow \boxed{\text{hidden}}$ weights are modified as:

$$\Delta w_{jk} = \eta \delta_j^q x_k^q$$

$$x_k^q = 0 \longrightarrow \Delta w_{jk} = 0$$

⇒ 0 inputs cause no weight change even if δ_j^q is large

⇒ Learning on these inputs happens only passively (by omission).

Using $\pm$ allows an active weight change at every steps rather than only at the $x = +1$ steps.

$\downarrow$

Potentially faster learning

Choice of Activation Function

Use $f(u) = a \tanh(bu)$

Same reason as scaling inputs 0-mean

$$f(u) = \frac{1}{1+e^{-u}} \Rightarrow 0 < f(u) < 1$$

$\Rightarrow$ all weights to a target neuron increase or decrease together

$$f(u) = a \tanh(bu) \Rightarrow -a < f(u) < a$$

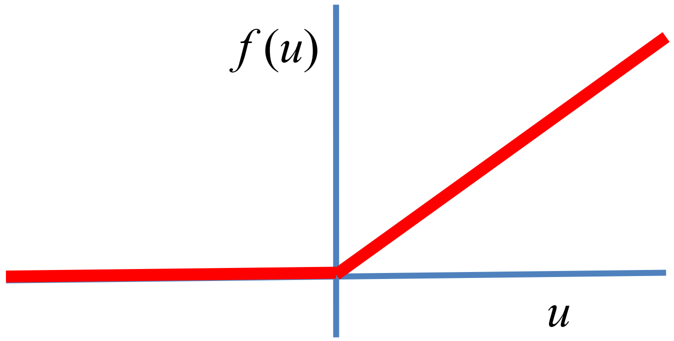
which allows greater discrimination

Le Cun (1989, 1993) suggests

$a = 1.7159 \quad b = 2/3$

This gives $f(1) = 1 \quad f(-1) = -1$

In some cases, a rectified linear unit (ReLU) may work well.



Weight Initialization

Consider

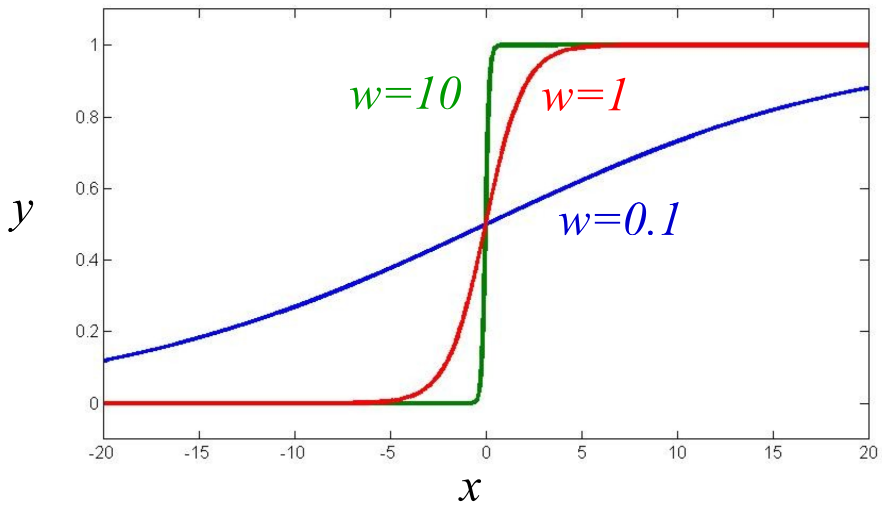
$$y = \frac{1}{1+e^{-wx}}$$

- If w is too small, $f'(wx)$ is a small value for most $x \rightarrow$ slow learning.

- If w is too large, wx is large for most x (except near $x = 0$) and $f'(wx) \approx 0$ (saturation) $\rightarrow$ slow learning.

- For $x \in \Re^n$, the same argument applies w.r.t. values of $||w||$

$\Rightarrow$ The more input lines a neuron has, the smaller its range of weights should be to avoid saturating the sigmoid and causing the gradient to vanish.



Gaussian with mean = 0, standard deviation σ ~

Uniform between (-*a*, +*a*) where *a* ~

Where *n* = number of inputs to the neuron.

Xavier initialization (Glorot & Bengio, 2010):

Uniform between (-*a*, +*a*) where *a* ~

N_s = number of neurons in source layer

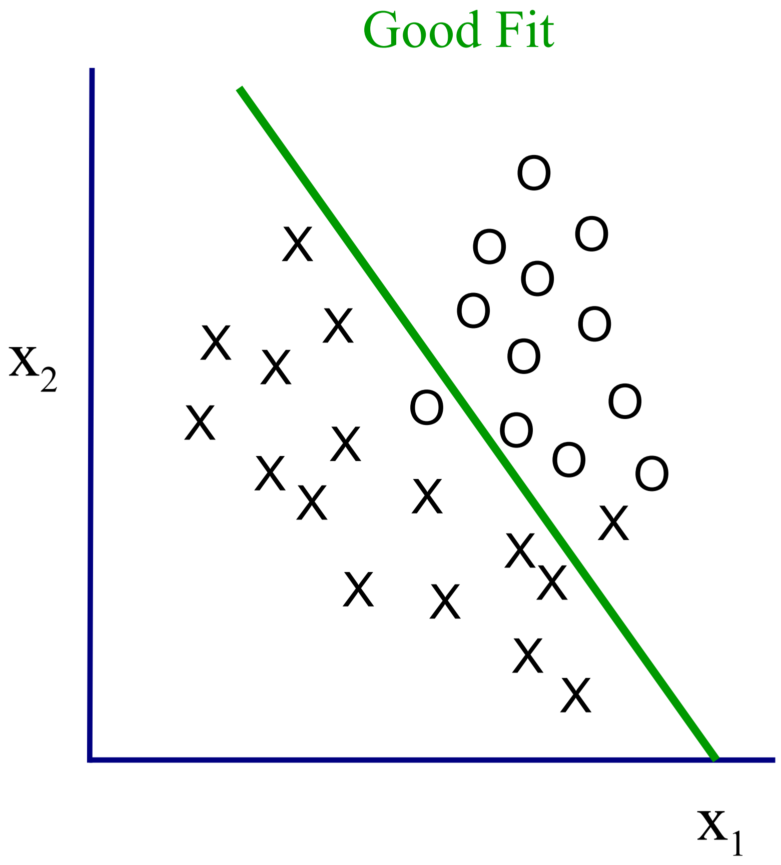
N_t = number of neurons in target layer

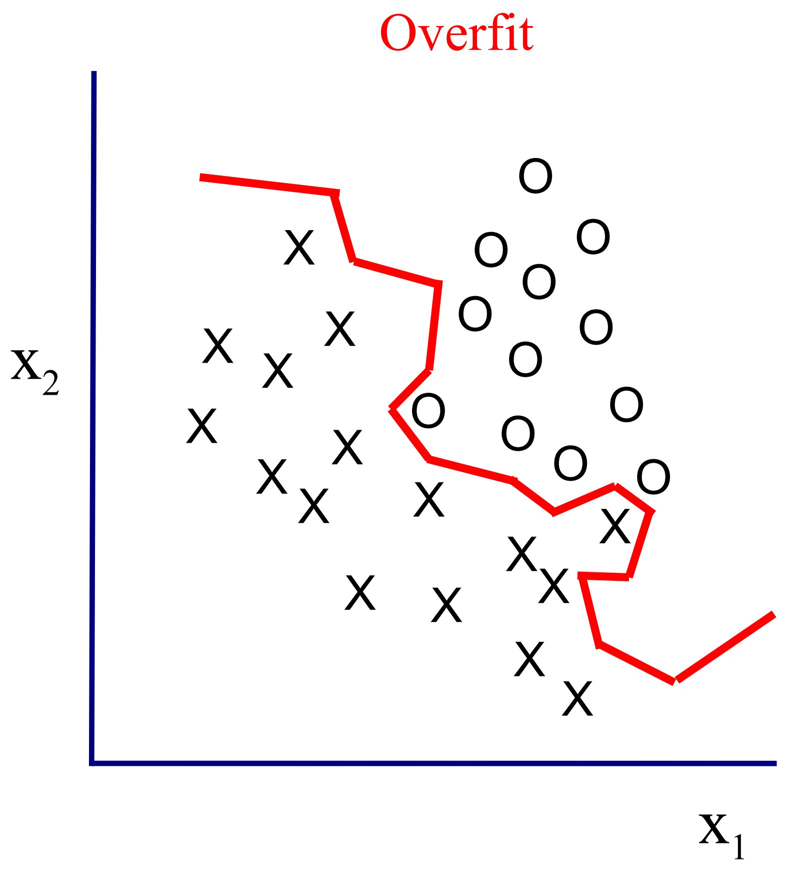
Xavier Glorot, Yoshua Bengio, Proceedings of the Thirteenth International Conference on Artificial Intelligence and Statistics, PMLR 9:249-256, 2010.

- Don't initialize all weights to identical values. This may cause all hidden neurons to learn the same thing and preclude overall learning.
- Initialize weights to be both positive and negative instead of just positive.

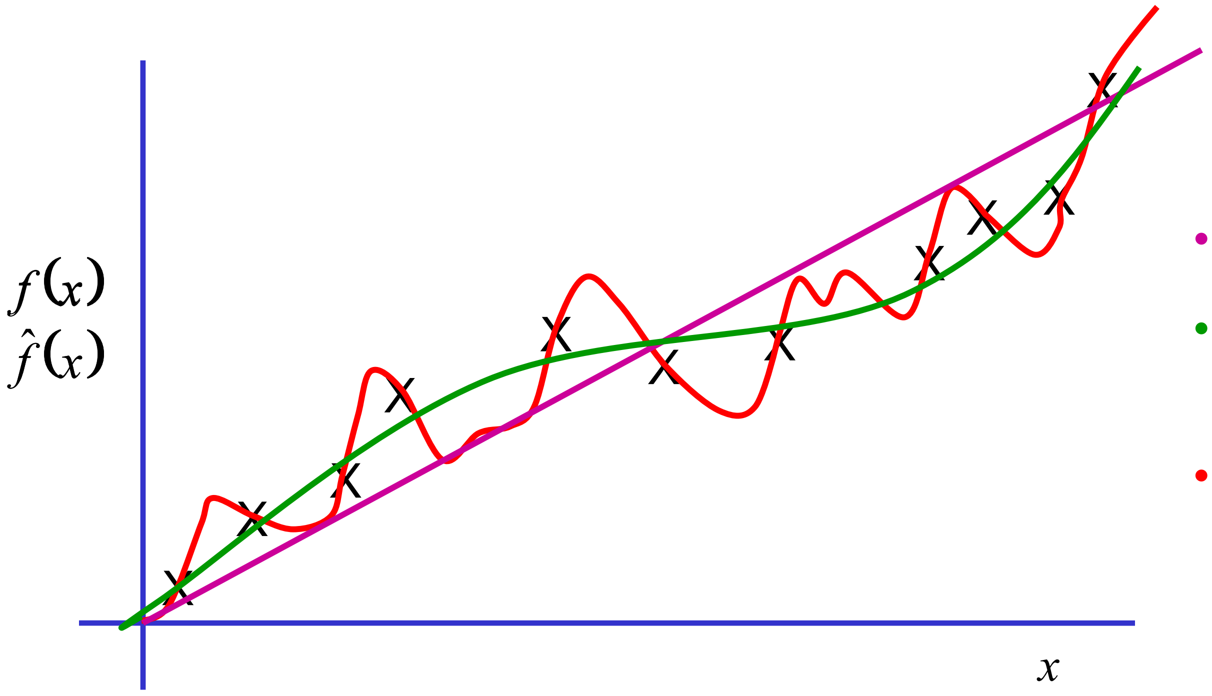
Generalization and Validation

Fitting vs. Overfitting (Classification)





Fitting vs. Overfitting (Approximation)



- Linear Fit (fine)
- Smooth nonlinear fit (best)
- Overfit

Fitting vs. Overfitting

The fine variations seen in overfitting are fitting the “noise” rather than the “signal”

→ → poor generalization

What causes overfitting?

Too many “degrees of freedom” = ways of variation.

In feed-forward networks:

Degrees of freedom ↔ Number of weights and hidden neurons

More weights and neurons arrow more complicated functions and decision boundaries → Overfitting → Poor generalization

use fewer hidden neurons and/or fewer weights/neuron.

Regularization

Modifications to the learning process, network behavior, or network structure that force better generalization.

Some common regularization methods:

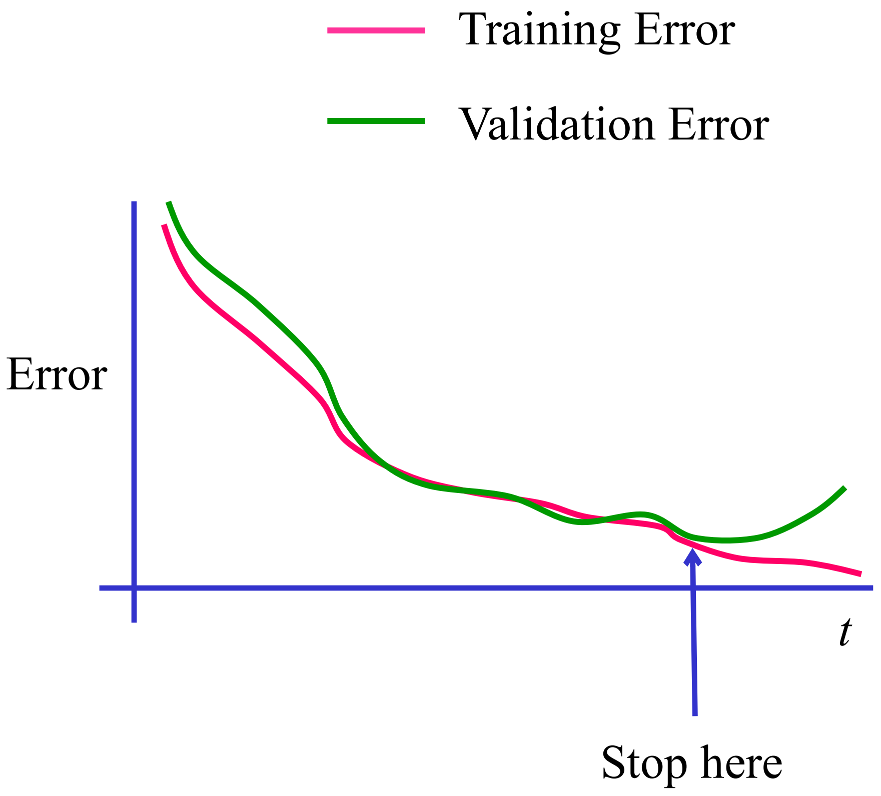
- Early stopping.
- Weight decay.
- Pruning.
- Dropout.
- Sparsity constraints.

Early Stopping

- During training, periodically evaluate network performance on a ***validation set***.

- If training and testing error are both going down, keep training.

- If validation error starts to go up, stop training even if training error is still going down.



- Assumes $\boxed{1}$ "Signal" is learned before "noise".

$\boxed{2}$ Once validation error starts going up, it will never reverse course $\longrightarrow$ Both are heuristics

Weight Decay

Force the total weight of the network to be as low as possible.

For example, add a weight penalty term to the loss function:

$$J^q = \frac{1}{2} \sum_{i \in Outputs} \left(y_i^q - \hat{y}_i^q\right)^2 + \boxed{\lambda \sum_{Layers\ L} \; \sum_{j \in Layer\ L} \; \sum_{i \in Layer\ L+1} \left(w_{ij}\right)^2} \longleftarrow \text{Weight Penalty}$$

Why does this work?

It forces the network to use weights more efficiently in one or both of two ways:

- Make all weights small → smoother function
- Use fewer non-zero weights → fewer degrees-of-freedom

Both → better generalization.

Pruning and Construction

Pruning:

- During training, encourage the network to use as few weights as possible $\rightarrow$ some weights become ≈ 0

- Remove weights of very small size

Notes: $\boxed{1}$ Some algorithms do only the second step

$\boxed{2}$ Pruning algorithms can also remove neurons to become useless

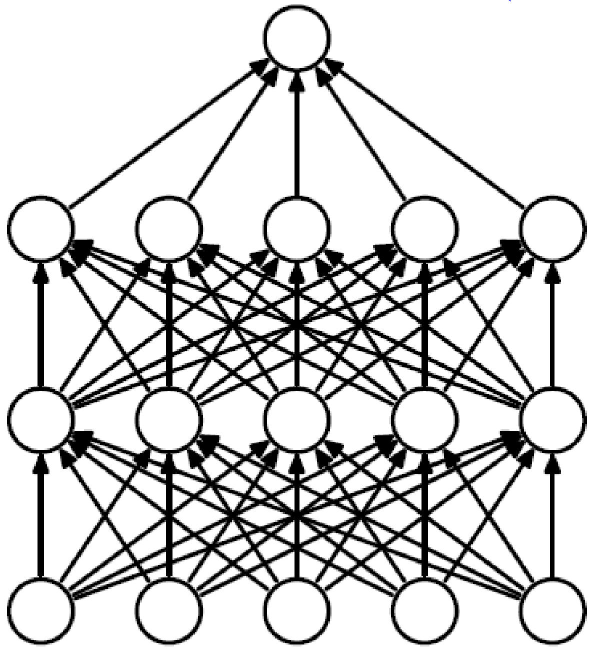
Construction:

Type I: Start with a small network, train as far as possible, add a neuron and keep training. Repeat this process till satisfied.

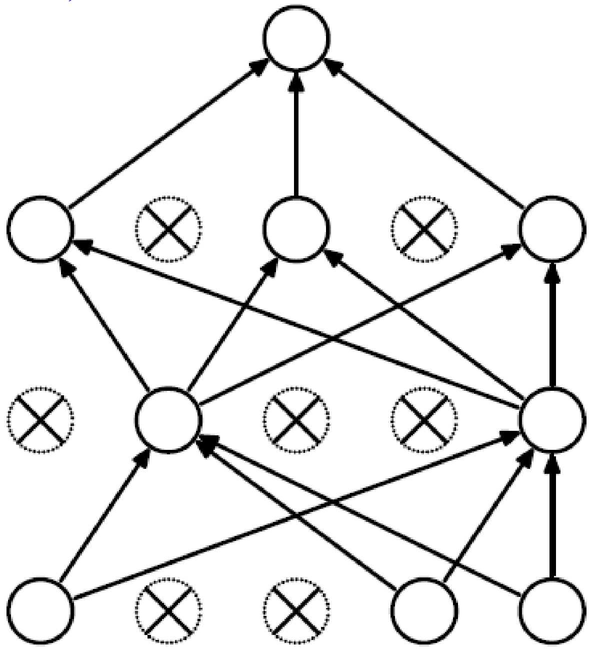
Type II: Start with a large network, train, prune, train, add, ….

Dropout

(Srivastava et al., 2014)



(a) Standard Neural Net



(b) After applying dropout.

Randomly leave out some inputs and hidden neurons during each iteration of the learning update, to prevent them from co-adapting – forcing them to learn somewhat different "views" of the data.

Srivastava et al. (2014) Dropout: A Simple Way to Prevent Neural Networks from Overfitting, *Journal of Machine Learning Research* 15: 1929-1958