IT/PC/B/T/411

Machine Learning

Neural Networks

Dr. Pawan Kumar Singh
Department of Information Technology
Jadavpur University
pawankrsingh.cse@gmail.com
+91-6291555693

Artificial Neural Networks

- Computational models inspired by the human brain:
  - Algorithms that try to mimic the brain.

Imp

Imp

  - Massively parallel, distributed system, made up of simple processing units (neurons)

Imp

V.V.IP

  - Synaptic connection strengths among neurons are used to store the acquired knowledge.
  - Knowledge is acquired by the network from its environment through a learning process

History

- late-1800's - Neural Networks appear as an analogy to biological systems
- 1960's and 70's – Simple neural networks appear
  - Fall out of favor because the perceptron is not effective by itself, and there were no good algorithms for multilayer nets
- 1986 – Backpropagation algorithm appears
  - Neural Networks have a resurgence in popularity
  - More computationally expensive

Applications of ANNs

- ANNs have been widely used in various domains for:
  - Pattern recognition
  - Function approximation
  - Associative memory

digit classificatn

(97% accuracy)

Properties

- Inputs are flexible Imp.
  - any real values
  - Highly correlated or independent
- Target function may be discrete-valued, real-valued, or vectors of discrete or real values
  - Outputs are real numbers between 0 and 1
- Resistant to errors in the training data Imp
- Long training time Imp
- Fast evaluation Imp
- The function produced can be difficult for humans to interpret Imp

When to consider neural networks

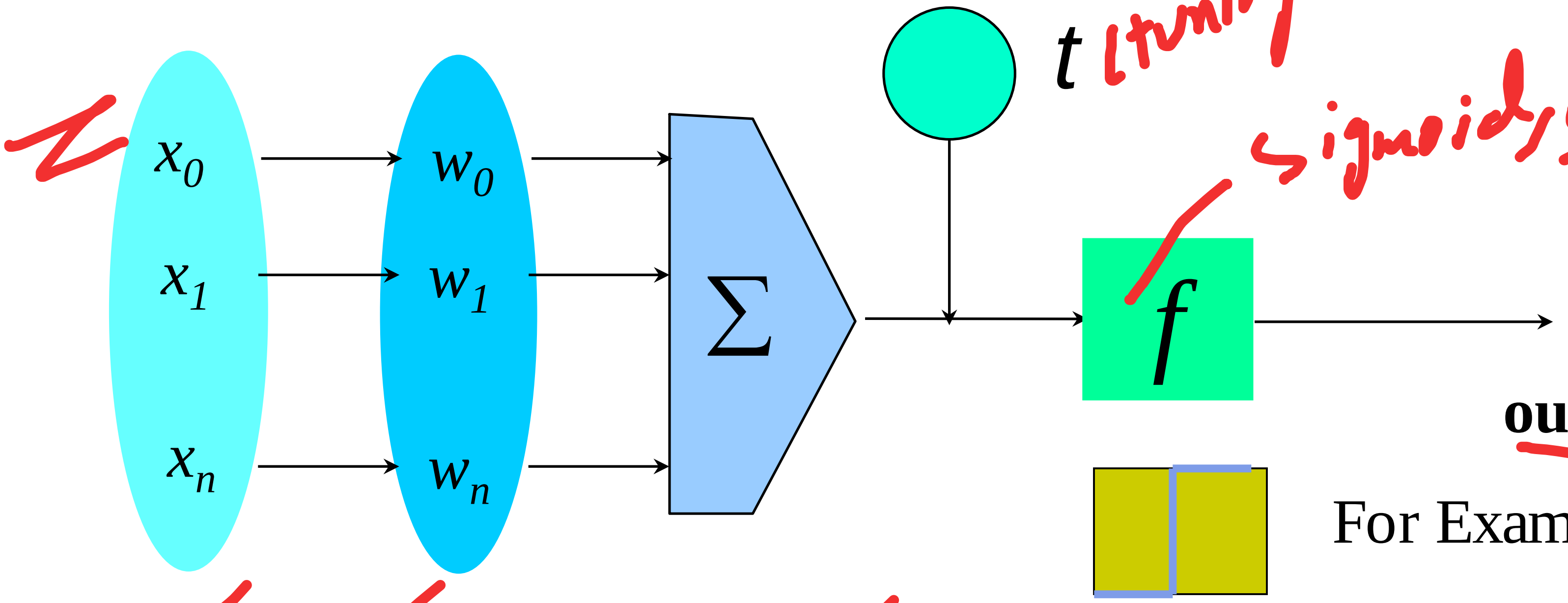
- Input is high-dimensional discrete or raw-valued Imp
- Output is discrete or real-valued Imp
- Output is a vector of values
- Possibly noisy data Imp
- Form of target function is unknown Imp
- Human readability of the result is not important Imp

Examples:

- Speech phoneme recognition Imp
- Image classification Imp Digit Classification
- Financial prediction Imp

Bank Credit Card churn

A Neuron (= a perceptron)



output y

For Example

Input vector **x** weight vector **w** weighted sum Activation function

$$y = \text{sign}\left(\sum_{i=0}^{n} w_i x_i - t\right)$$

- The n-dimensional input vector **x** is mapped into variable y by means of the scalar product and a nonlinear function mapping

dot product

$\sigma(W^T X + b)$

Perceptron

- Basic unit in a neural network
- Linear separator
- Parts
  - N inputs, $x_1 \ldots x_n$
  - Weights for each input, $w_1 \ldots w_n$
  - A bias input x_0 (constant) and associated weight w_0
  - Weighted sum of inputs, $y = w_0x_0 + w_1x_1 + \ldots + w_nx_n$
  - A threshold function or activation function,
    - i.e 1 if y > t, -1 if y <= t

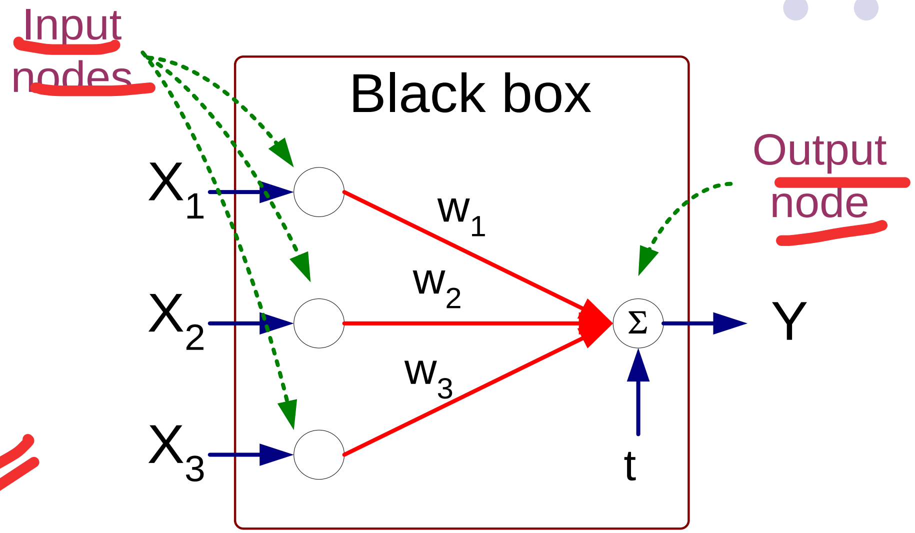
Artificial Neural Networks (ANN)

- Model is an assembly of inter-connected nodes and weighted links

imp

- Output node sums up each of its input value according to the weights of its links

- Compare output node against some threshold t



Perceptron Model

$$Y = I(\sum_i w_i x_i - t)$$

or

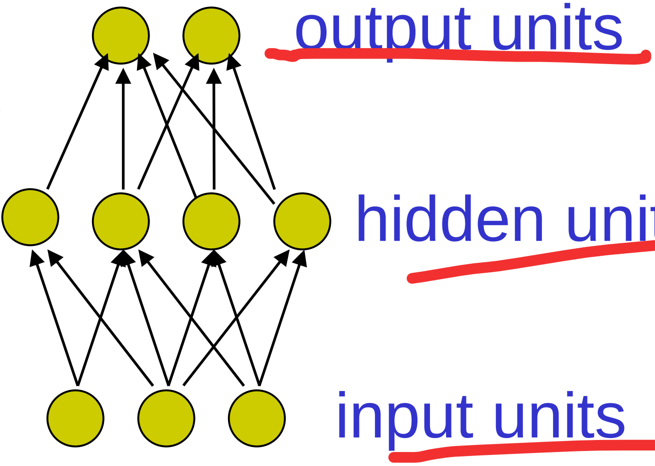
$$Y = sign(\sum_i w_i x_i - t)$$

Types of connectivity

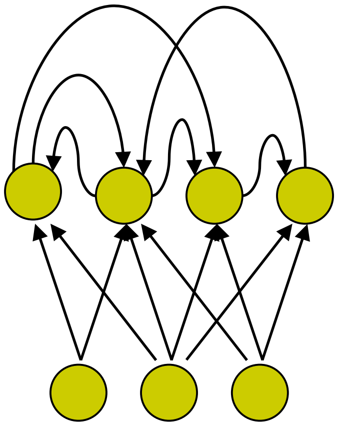
Imp

- Feedforward networks
  - These compute a series of transformations
  - Typically, the first layer is the input and the last layer is the output.
- Recurrent networks
  - These have directed cycles in their connection graph. They can have complicated dynamics.
  - More biologically realistic.

feedForward Networks

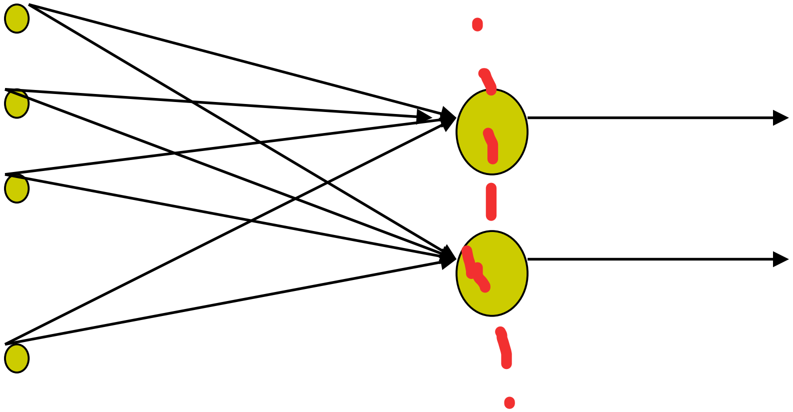


Recurrent networks



Different Network Topologies

- Single layer feed-forward networks
  - Input layer projecting into the output layer



Input layer

Output layer

Single layer network

1 classification
binary y.

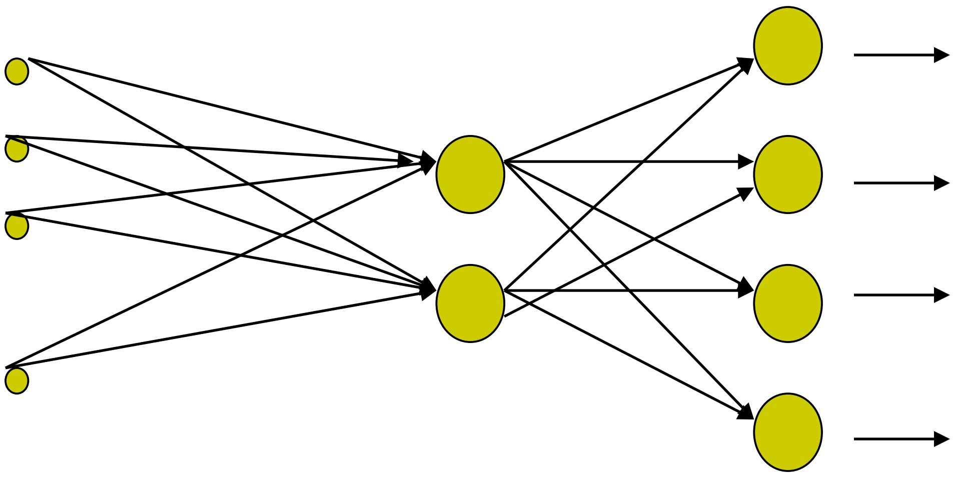
Different Network Topologies

- Multi-layer feed-forward networks

// general use

  - One or more hidden layers. Input projects only from previous layers onto a layer.

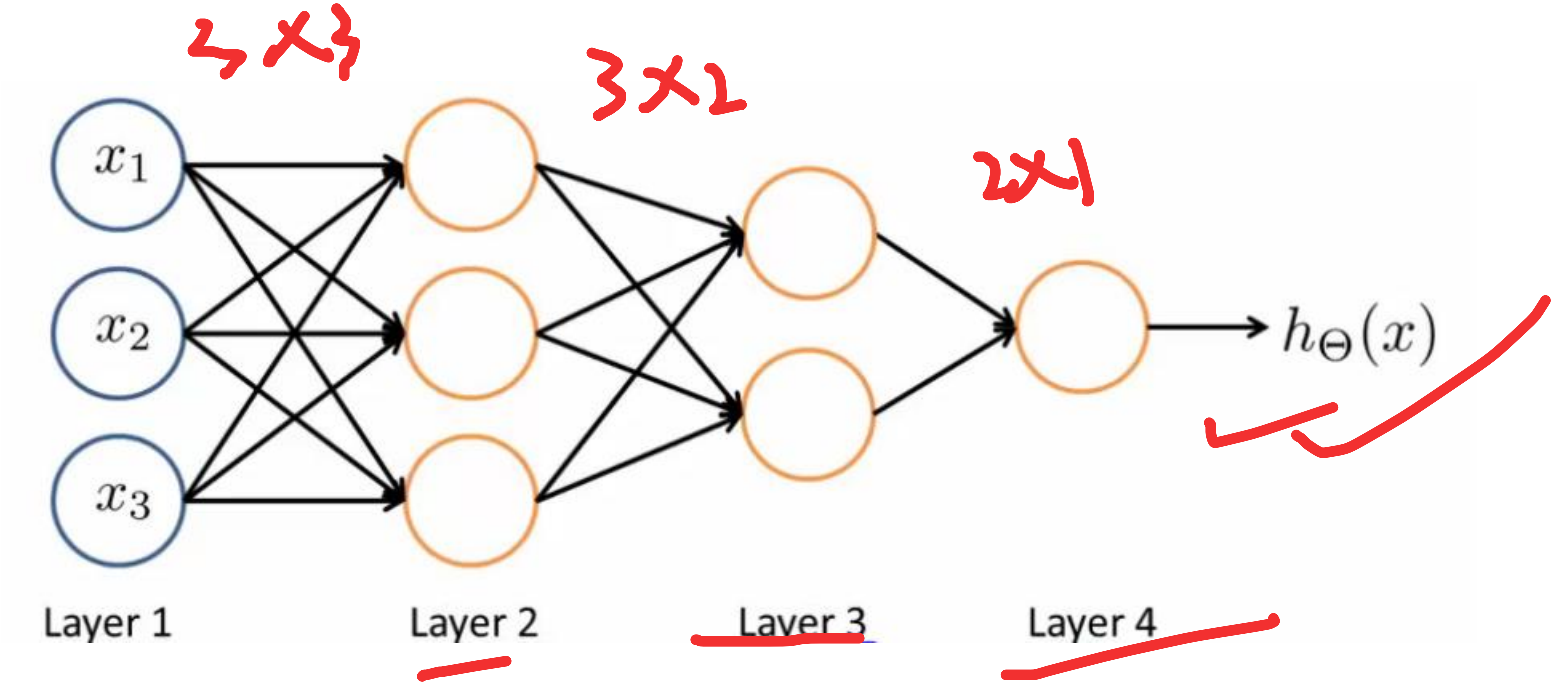
// multi class classification



2-layer or
1-hidden layer
fully connected
network

Different Network Topologies

- Multi-layer feed-forward networks



Input layer

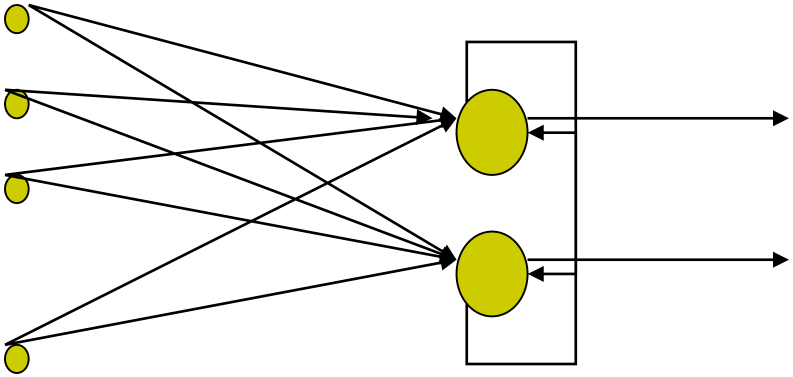
Hidden layers

Output layer

6 + 3 + 6 + 2 + 2 + 1 = 20 training parameters

Different Network Topologies

- Recurrent networks
  - A network with feedback, where some of its inputs are connected to some of its outputs (discrete time).



Input layer

Output layer

Recurrent network

Algorithm for learning ANN

wts

- Initialize the weights (w_0, w_1, …, w_k)

w_0 — 1 / $w_1, w_2 \ldots - w_k$ -|

- Adjust the weights in such a way that the output of ANN is consistent with class labels of training examples
  - Error function: $$E = \sum_i \left[Y_i - f(w_i, X_i)\right]^2$$

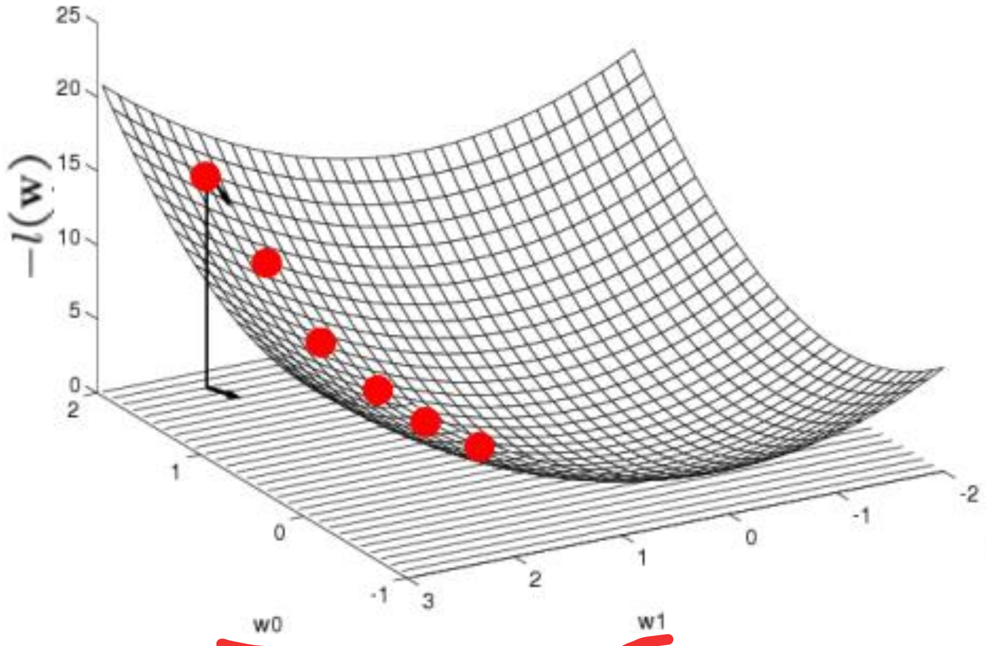
$(y_i - \hat{y}_i)^2$

  - Find the weights w_i's that minimize the above error function
    - e.g., gradient descent, backpropagation algorithm

Optimizing concave/convex function

- Maximum of a concave function = minimum of a convex function ∀

Gradient ascent (concave) / Gradient descent (convex)



Gradient:

$$\nabla_{\mathbf{w}} l(\mathbf{w}) = [\frac{\partial l(\mathbf{w})}{\partial w_0}, \ldots, \frac{\partial l(\mathbf{w})}{\partial w_d}]'$$

Update rule:

Learning rate, η>0

$$\Delta \mathbf{w} = \eta \nabla_{\mathbf{w}} l(\mathbf{w})$$

$$w_i^{(t+1)} \leftarrow w_i^{(t)} + \eta \frac{\partial l(\mathbf{w})}{\partial w_i}\bigg|_t$$

Gradient ascent rule

GRADIENT DESCENT

Suppose we have a scalar function

$$f(w) : \Re \rightarrow \Re$$

We want to find a local minimum.

Assume our current weight is *w*

GRADIENT DESCENT RULE:

$$w \leftarrow w - \eta \frac{\partial}{\partial w} f(w)$$

η is called the LEARNING RATE. A small positive number, e.g. η = 0.05

Gradient Descent in "m" Dimensions

Given $\mathrm{f}(\mathbf{w}) : \Re^m \rightarrow \Re$

$$\nabla \mathrm{f}(\mathrm{w}) = \begin{pmatrix} \frac{\partial}{\partial w_1} \mathrm{f}(\mathrm{w}) \\ \vdots \\ \frac{\partial}{\partial w_m} \mathrm{f}(\mathrm{w}) \end{pmatrix}$$ points in direction of steepest ascent.

$|\nabla \mathrm{f}(\mathrm{w})|$ is the gradient in that direction

GRADIENT DESCENT RULE: $\mathrm{w} \leftarrow \mathrm{w} - \eta \nabla \mathrm{f}(\mathrm{w})$

Equivalently

$$w_j \leftarrow w_j - \eta \frac{\partial}{\partial w_j} \mathrm{f}(\mathrm{w})$$ ….where w_j is the *j* th weight

"just like a linear feedback system"

Linear Perceptrons

They are multivariate linear models:

$$\text{Out}(\boldsymbol{x}) = \boldsymbol{w}^T\boldsymbol{x}$$

And "training" consists of minimizing sum-of-squared residuals by gradient descent.

$$E = \sum_k \left(\text{Out}(x_k) - y_k\right)^2$$

Imp

$$= \sum_k \left(w^T x_k - y_k\right)^2$$

QUESTION: Derive the perceptron training rule.

Linear Perceptron Training Rule

$$E = \sum_{k=1}^{R} (y_k - \mathbf{w}^T \mathbf{x}_k)^2$$

Gradient descent tells us we should update **w** thusly if we wish to minimize *E:*

$$w_j \leftarrow w_j - \eta \frac{\partial E}{\partial w_j}$$ ✓✓

So what's $\frac{\partial E}{\partial w_j}$?

Linear Perceptron Training Rule

$$E = \sum_{k=1}^{R} (y_k - \mathbf{w}^T \mathbf{x}_k)^2$$

Gradient descent tells us we should update **w** thusly if we wish to minimize *E:*

$$w_j \leftarrow w_j - \eta \frac{\partial E}{\partial w_j}$$

So what's $\frac{\partial E}{\partial w_j}$?

$$\frac{\partial E}{\partial w_j} = \sum_{k=1}^{R} \frac{\partial}{\partial w_j} (y_k - \mathbf{w}^T \mathbf{x}_k)^2$$

$$= \sum_{k=1}^{R} 2(y_k - \mathbf{w}^T \mathbf{x}_k) \frac{\partial}{\partial w_j} (y_k - \mathbf{w}^T \mathbf{x}_k)$$

$$= -2 \sum_{k=1}^{R} \delta_k \frac{\partial}{\partial w_j} \mathbf{w}^T \mathbf{x}_k$$

...where...

$$\delta_k = y_k - \mathbf{w}^T \mathbf{x}_k$$

$$= -2 \sum_{k=1}^{R} \delta_k \frac{\partial}{\partial w_j} \sum_{i=1}^{m} w_i x_{ki}$$

$$= -2 \sum_{k=1}^{R} \delta_k x_{kj}$$

Linear Perceptron Training Rule

$$E = \sum_{k=1}^{R} (y_k - \mathbf{w}^T \mathbf{x}_k)^2$$

Gradient descent tells us we should update **w** thusly if we wish to minimize *E:*

[illegible]

$$w_j \leftarrow w_j - \eta \frac{\partial E}{\partial w_j}$$

…where…

$$\frac{\partial E}{\partial w_j} = -2 \sum_{k=1}^{R} \delta_k x_{kj}$$

$$\boxed{w_j \leftarrow w_j + 2\eta \sum_{k=1}^{R} \delta_k x_{kj}}$$

We frequently neglect the 2 (meaning we halve the learning rate)

The “Batch” perceptron algorithm

1) Randomly initialize weights $w_1 \; w_2 \ldots w_m$

2) Get your dataset (append 1’s to the inputs if you don’t want to go through the origin).

3) for i = 1 to R

$$\delta_i := \underline{y_i - \mathbf{w}^{\mathrm{T}} \mathbf{x}_i}$$

4) for j = 1 to m

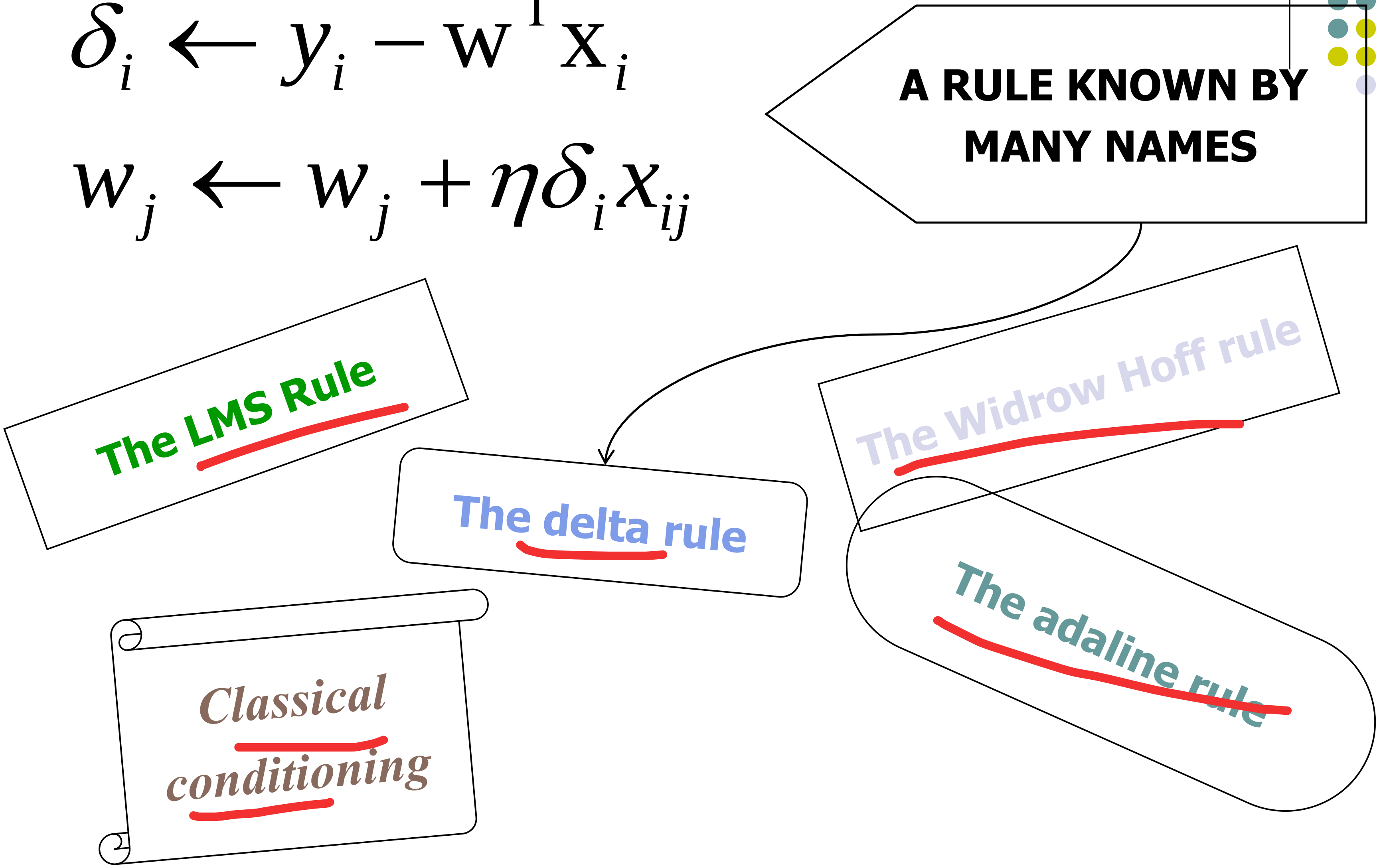
$$w_j \leftarrow w_j + \eta \sum_{i=1}^{R} \delta_i x_{ij}$$

5) if $\underline{\sum \delta_i^2}$ stops improving then stop. Else loop back to 3.

epoch .

$$\delta_i \leftarrow y_i - \mathrm{w}^{\mathrm{T}}\mathrm{x}_i$$

$$w_j \leftarrow w_j + \eta\delta_i x_{ij}$$



A RULE KNOWN BY MANY NAMES

The LMS Rule

The delta rule

The Widrow Hoff rule

The adaline rule

Classical conditioning

Perceptrons and Boolean Functions

- Can learn any disjunction of literals

  e.g. $x_1 \wedge \sim x_2 \wedge \sim x_3 \wedge x_4 \wedge x_5$

- Can learn majority function

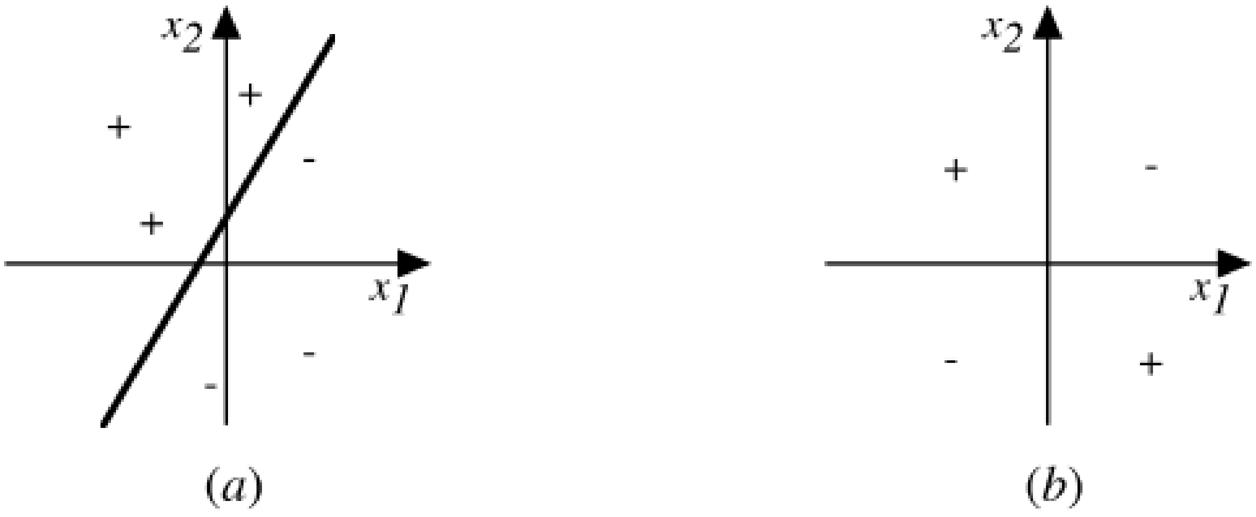
  $$f(x_1, x_2 \ldots x_n) = \begin{cases} 1 \text{ if } n/2 \; x_i\text{'s or more are} = 1 \\ 0 \text{ if less than } n/2 \; x_i\text{'s are} = 1 \end{cases}$$

- What about the exclusive or function?

  $f(x_1,x_2) = x_1 \forall x_2 =$

  $(x_1 \wedge \sim x_2) \vee (\sim x_1 \wedge x_2)$

Decision surface of a perceptron

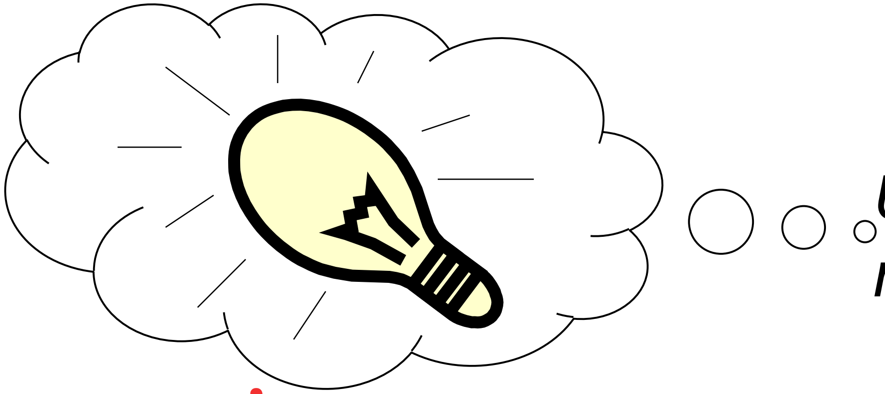


- Decision surface is a hyperplane
  - Can capture linearly separable classes
- Non-linearly separable
  - Use a network of them

Multilayer Networks

The class of functions representable by perceptrons is limited

$$\text{Out}(\text{x}) = g\left(\mathbf{w}^{\text{T}}\mathbf{x}\right) = g\left(\sum_j w_j x_j\right)$$



Use a wider representation !

(How)

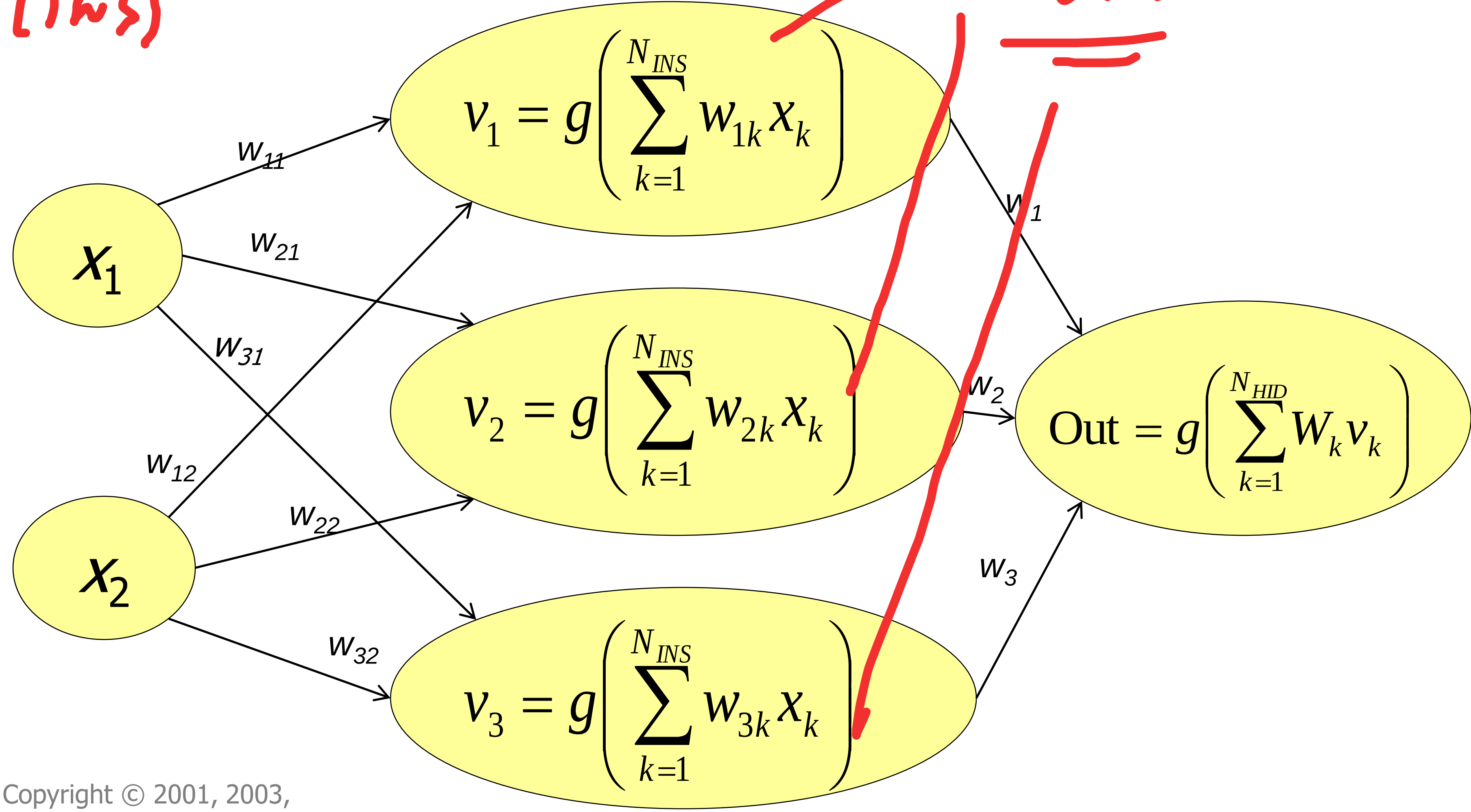
$$\text{Out}(\text{x}) = g\left(\sum_j W_j g\left(\sum_k w_{jk} x_{jk}\right)\right)$$

This is a nonlinear function
Of a linear combination
Of non linear functions
Of linear combinations of inputs

A 1-HIDDEN LAYER NET

$N_{INPUTS} = 2$ (INS)

$N_{HIDDEN} = 3$ (HID no-bias



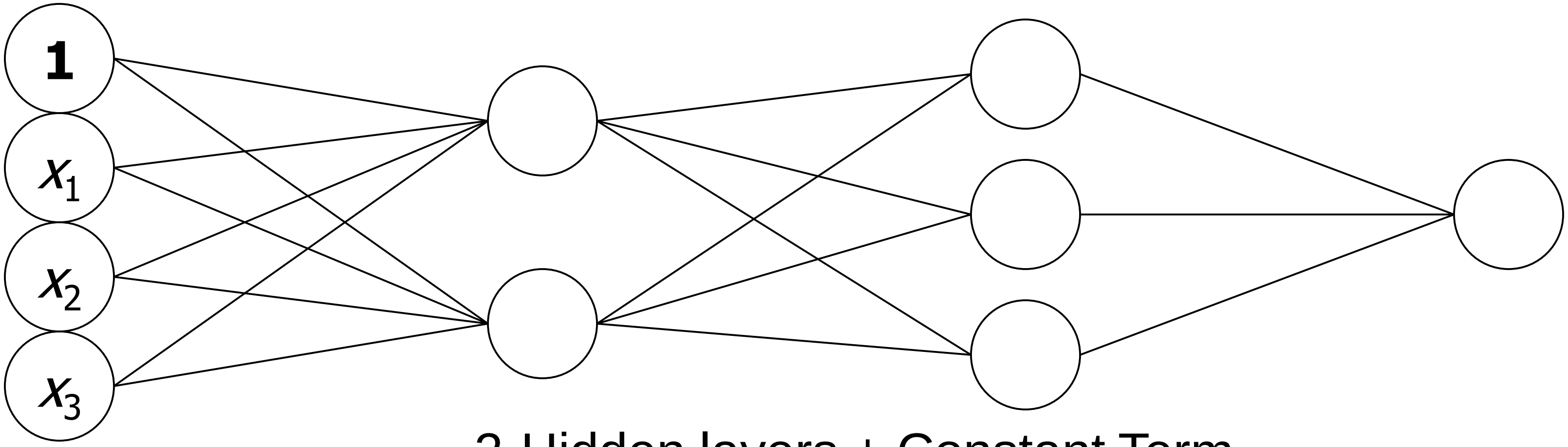
$$v_1 = g\left(\sum_{k=1}^{N_{INS}} w_{1k} x_k\right)$$

$$v_2 = g\left(\sum_{k=1}^{N_{INS}} w_{2k} x_k\right)$$

$$v_3 = g\left(\sum_{k=1}^{N_{INS}} w_{3k} x_k\right)$$

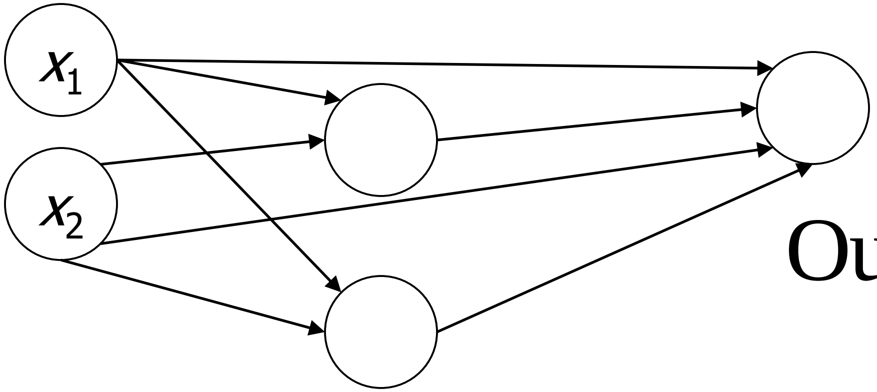
$$\text{Out} = g\left(\sum_{k=1}^{N_{HID}} W_k v_k\right)$$

OTHER NEURAL NETS



2-Hidden layers + Constant Term

"JUMP" CONNECTIONS



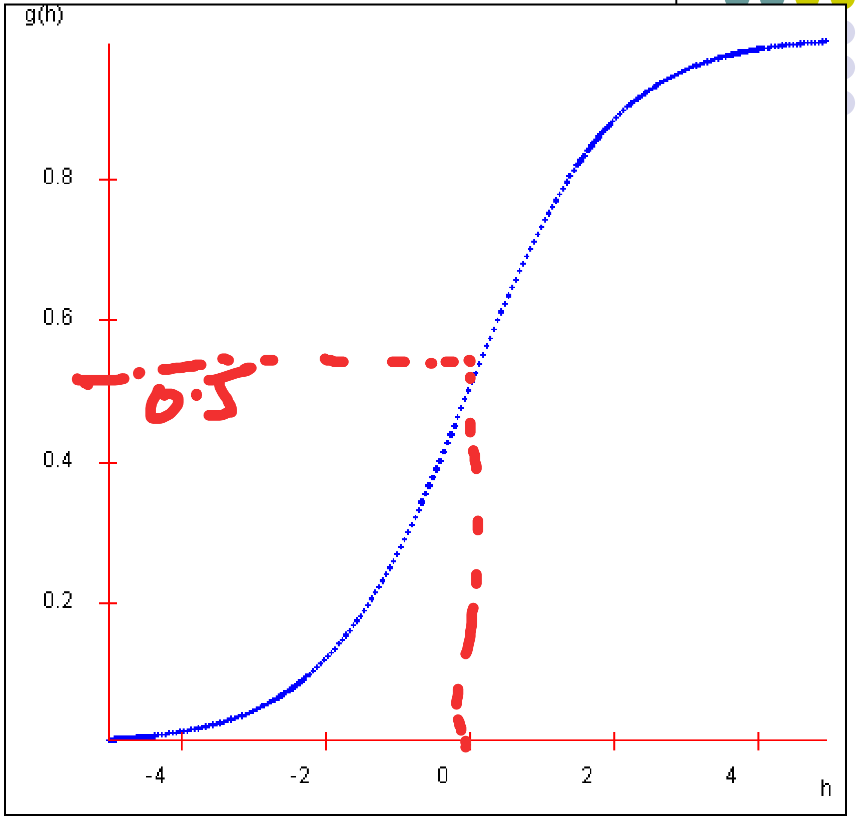
$$\text{Out} = g\left(\sum_{k=1}^{N_{INS}} w_{0k} x_k + \sum_{k=1}^{N_{HID}} W_k v_k \right)$$

Multi-layer Networks

- Linear units inappropriate
  - No more expressive than a single layer
- Introduce non-linearity
  - Threshold not differentiable
- Use sigmoid function

The Sigmoid

$$g(h) = \frac{1}{1+\exp(-h)}$$



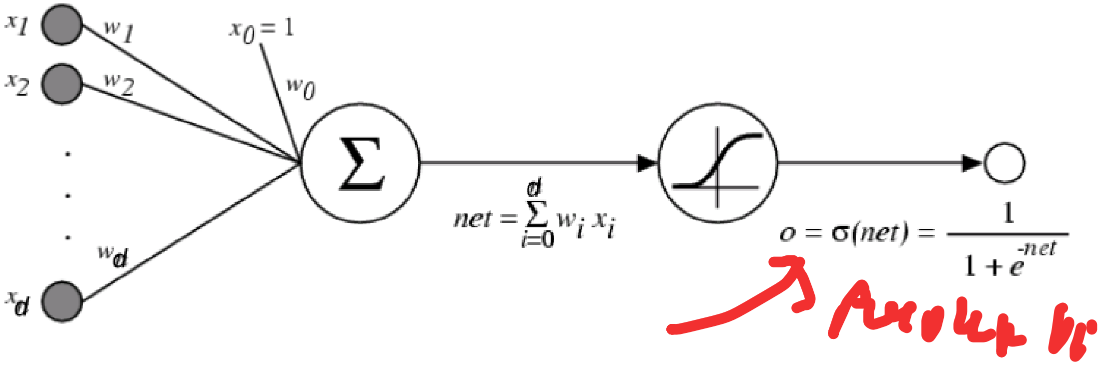
Now we choose ***w*** to minimize

$$\sum_{i=1}^{R}\left[y_i - \mathrm{Out}(\mathrm{x}_i)\right]^2 = \sum_{i=1}^{R}\left[y_i - \underline{g(\mathrm{w}^{\mathrm{T}}\mathrm{x}_i)}\right]^2$$

mse

sigmoid

Sigmoid Unit


x_1
w_1
$x_0 = 1$
w_0
x_2
w_2
w_d
x_d
Σ
$net = \sum_{i=0}^{d} w_i x_i$
$o = \sigma(net) = \frac{1}{1 + e^{-net}}$

Output Unit

$\sigma(x)$ is the sigmoid function/activation function (also linear, threshold)

$$\frac{1}{1+e^{-x}}$$

Nice property: $\frac{d\sigma(x)}{dx} = \sigma(x)(1 - \sigma(x))$ Differentiable

We can derive gradient decent rules to train

- One sigmoid unit
- *Multilayer networks* of sigmoid units → Backpropagation

Backpropagation

$$\mathrm{Out}(\mathrm{x}) = g\left(\sum_j W_j g\left(\sum_k w_{jk} x_k\right)\right)$$

Find a set of weights $\{W_j\},\{w_{jk}\}$

to minimize

$$\sum_i \left(y_i - \mathrm{Out}\left(\mathrm{x}_i\right)\right)^2$$

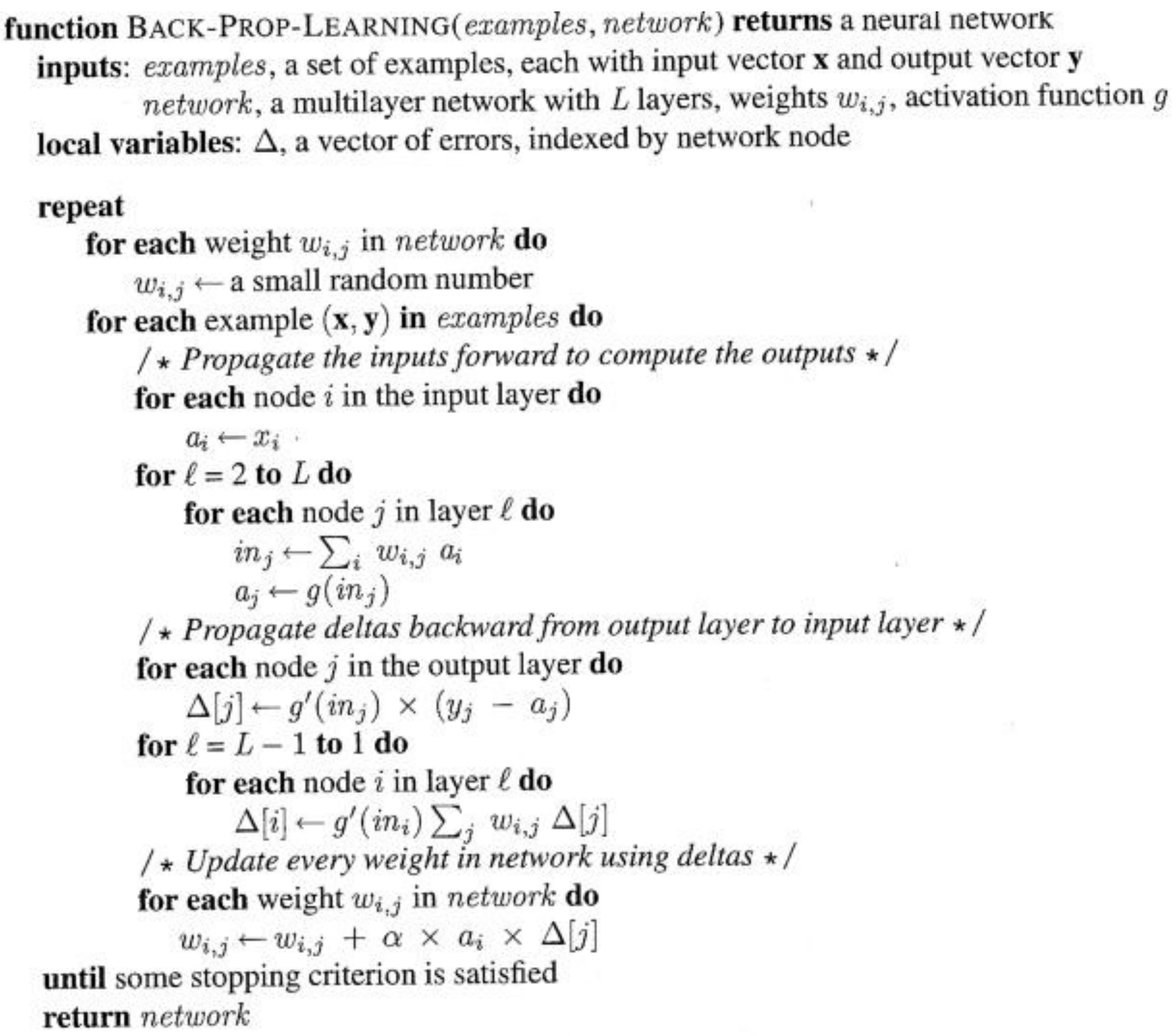
by gradient descent.

That’s it!

That’s the backpropagation algorithm.

Backpropagation

- Iteratively process a set of training tuples & compare the network's prediction with the actual known target value
- For each training tuple, the weights are modified to **minimize the mean squared error** between the network's prediction and the actual target value
- Modifications are made in the "**backwards**" direction: from the output layer, through each hidden layer down to the first hidden layer, hence "**backpropagation**"
- Steps
  - Initialize weights (to small random #s) and biases in the network
  - Propagate the inputs forward (by applying activation function)
  - Backpropagate the error (by updating weights and biases)
  - Terminating condition (when error is very small, etc.)



```
function BACK-PROP-LEARNING(examples, network) returns a neural network
  inputs: examples, a set of examples, each with input vector $\mathbf{x}$ and output vector $\mathbf{y}$
          network, a multilayer network with $L$ layers, weights $w_{i,j}$, activation function $g$
  local variables: $\Delta$, a vector of errors, indexed by network node

  repeat
      for each weight $w_{i,j}$ in network do
          $w_{i,j} \leftarrow$ a small random number
      for each example $(\mathbf{x}, \mathbf{y})$ in examples do
          /* Propagate the inputs forward to compute the outputs */
          for each node $i$ in the input layer do
              $a_i \leftarrow x_i$
          for $\ell = 2$ to $L$ do
              for each node $j$ in layer $\ell$ do
                  $in_j \leftarrow \sum_i w_{i,j}\, a_i$
                  $a_j \leftarrow g(in_j)$
          /* Propagate deltas backward from output layer to input layer */
          for each node $j$ in the output layer do
              $\Delta[j] \leftarrow g'(in_j) \times (y_j - a_j)$
          for $\ell = L - 1$ to $1$ do
              for each node $i$ in layer $\ell$ do
                  $\Delta[i] \leftarrow g'(in_i) \sum_j w_{i,j}\, \Delta[j]$
          /* Update every weight in network using deltas */
          for each weight $w_{i,j}$ in network do
              $w_{i,j} \leftarrow w_{i,j} + \alpha \times a_i \times \Delta[j]$
  until some stopping criterion is satisfied
  return network
```

How A Multi-Layer Neural Network Works?

- The **inputs** to the network correspond to the attributes measured for each training tuple
- Inputs are fed simultaneously into the units making up the **input layer**
- They are then weighted and fed simultaneously to a **hidden layer**
- The number of hidden layers is arbitrary, although usually only one
- The weighted outputs of the last hidden layer are input to units making up the **output layer**, which emits the network's prediction
- The network is **feed-forward** in that none of the weights cycles back to an input unit or to an output unit of a previous layer
- From a statistical point of view, networks perform **nonlinear regression**: Given enough hidden units and enough training samples, they can closely approximate any function

Defining a Network Topology

- First decide the **network topology:** # of units in the *input layer*, # of *hidden layers* (if > 1), # of units in *each hidden layer*, and # of units in the *output layer*
- Normalizing the input values for each attribute measured in the training tuples to [0.0—1.0]
- One **input** unit per domain value, each initialized to 0
- **Output**, if for classification and more than two classes, one output unit per class is used
- Once a network has been trained and its accuracy is **unacceptable**, repeat the training process with a *different network topology* or a *different set of initial weights*

Backpropagation and Interpretability

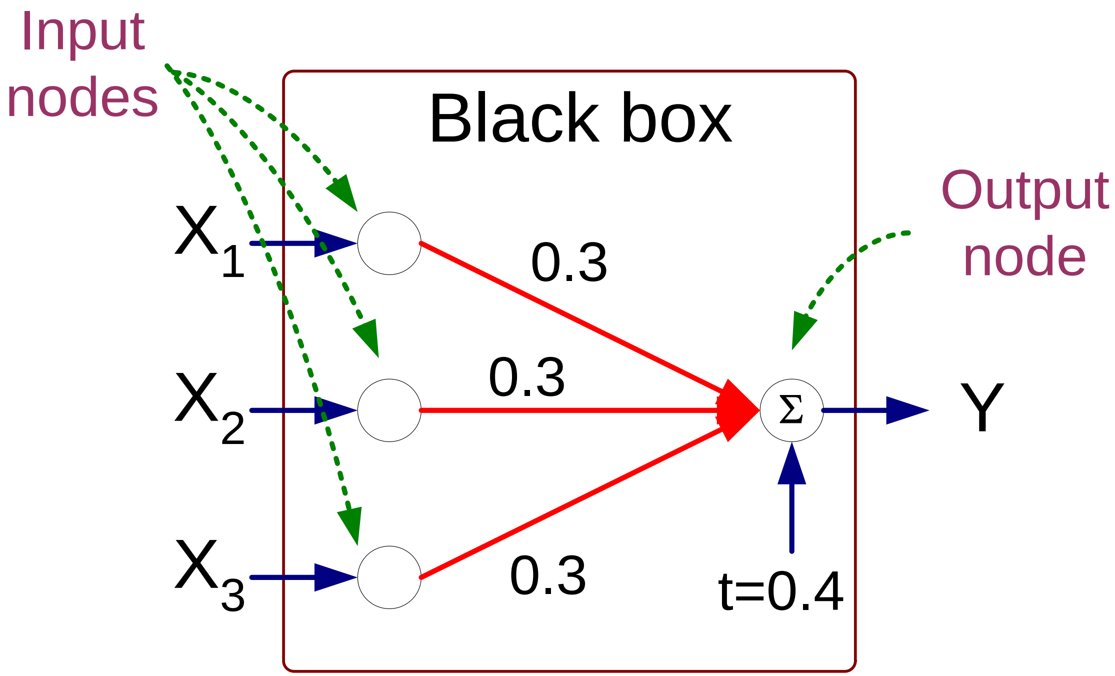
- Efficiency of backpropagation: Each **epoch** (one interation through the training set) takes O(|D| * *w*), with |D| tuples and *w* weights, but # of epochs can be exponential to n, the number of inputs, in the worst case
- **Rule extraction from networks:** network pruning
  - Simplify the network structure by removing weighted links that have the least effect on the trained network
  - Then perform link, unit, or activation value clustering
  - The set of input and activation values are studied to derive rules describing the relationship between the input and hidden unit layers
- **Sensitivity analysis:** assess the impact that a given input variable has on a network output. The knowledge gained from this analysis can be represented in rules

Neural Network as a Classifier

- Weakness
  - Long training time
  - Require a number of parameters typically best determined empirically, e.g., the network topology or “structure.”
  - Poor interpretability: Difficult to interpret the symbolic meaning behind the learned weights and of “hidden units” in the network
- Strength
  - High tolerance to noisy data
  - Ability to classify untrained patterns
  - Well-suited for continuous-valued inputs and outputs
  - Successful on a wide array of real-world data
  - Algorithms are inherently parallel
  - Techniques have recently been developed for the extraction of rules from trained neural networks

Artificial Neural Networks (ANN)

X_1	X_2	X_3	Y
1	0	0	0
1	0	1	1
1	1	0	1
1	1	1	1
0	0	1	0
0	1	0	0
0	1	1	1
0	0	0	0



$$Y = I(0.3X_1 + 0.3X_2 + 0.3X_3 - 0.4 > 0)$$

$$\text{where } I(z) = \begin{cases} 1 & \text{if } z \text{ is true} \\ 0 & \text{otherwise} \end{cases}$$

Linear Perceptrons

They are multivariate linear models:

$$\text{Out}(\boldsymbol{x}) = \boldsymbol{w}^{\mathrm{T}}\boldsymbol{x}$$

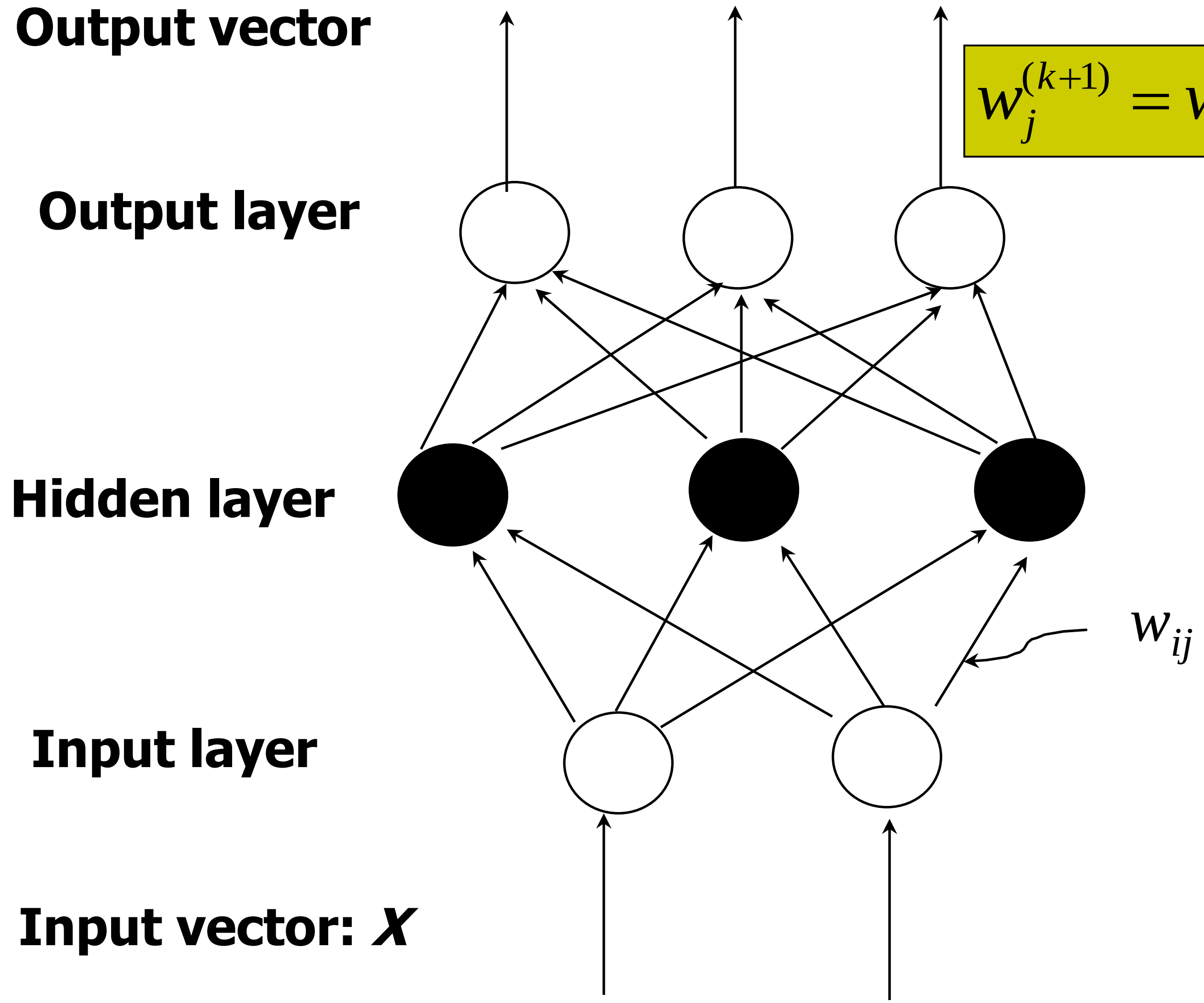
And “training” consists of minimizing sum-of-squared residuals by gradient descent.

$$\begin{aligned}\mathrm{E} &= \sum_{k} (\mathrm{Out}(\mathrm{x}_{\mathrm{k}}) - y_{\mathrm{k}})^2 \\ &= \sum_{k} \left(\mathrm{w}^{\mathrm{T}}\mathrm{x}_{\mathrm{k}} - y_{\mathrm{k}}\right)^2\end{aligned}$$

QUESTION: Derive the perceptron training rule.

A Multi-Layer Feed-Forward Neural Network

$$w_j^{(k+1)} = w_j^{(k)} + \lambda(y_i - \hat{y}_i^{(k)})x_{ij}$$



Output vector

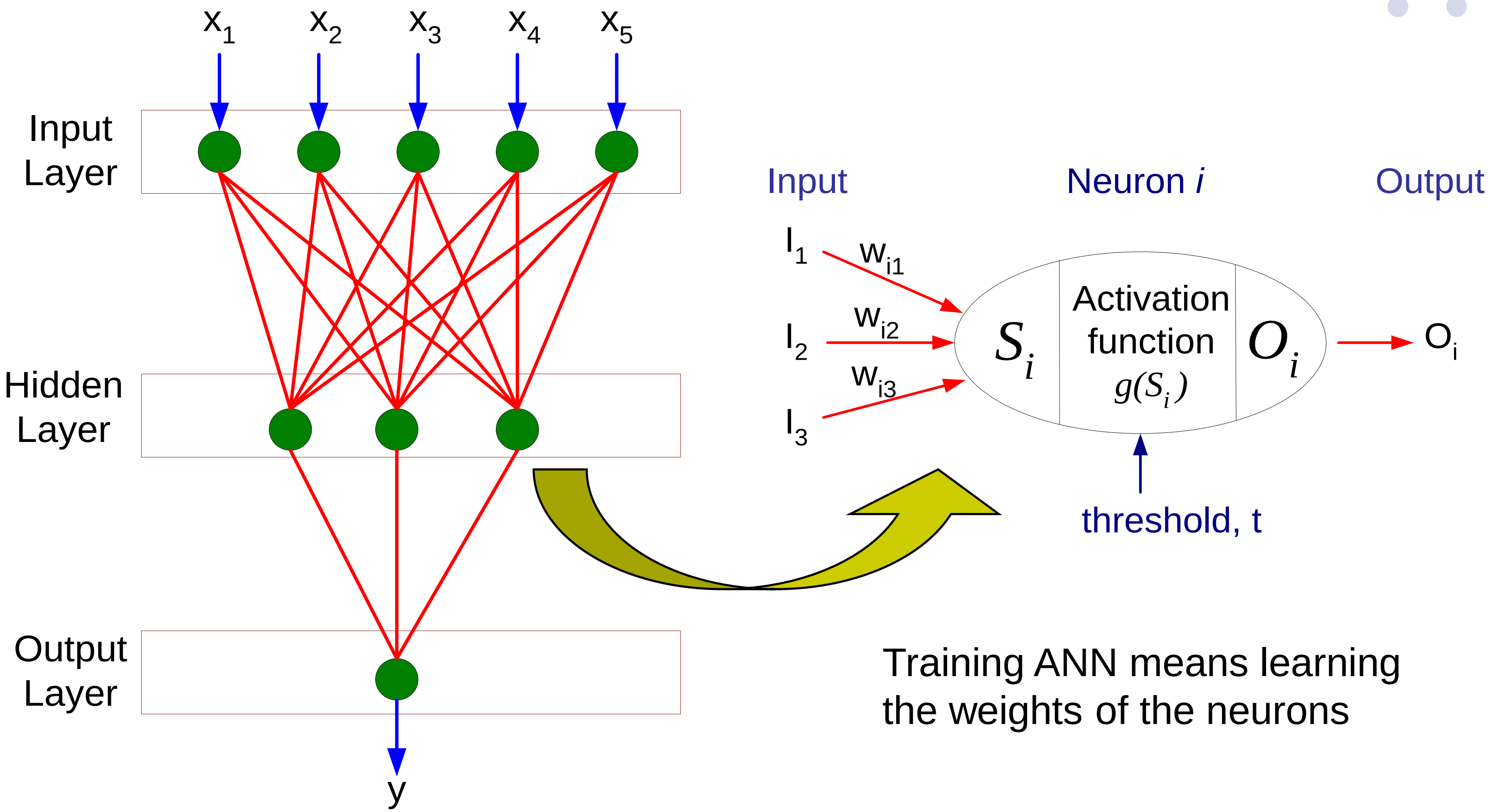
Output layer

Hidden layer

Input layer

Input vector: ***X***

General Structure of ANN



Training ANN means learning the weights of the neurons