Artificial Intelligence for Medicine II

Spring 2025

Lecture 52: Supervised Learning
ARTIFICIAL NEURAL NETWORKS

(Many slides adapted from Bing Liu, Han, Kamber & Pei; Tan, Steinbach, Kumar   and the web)

Artificail Neural Networks (ANN)

- Two basic motivations for ANN research:
  - to model brain function
  - to solve engineering (and business) problems
- So far as modeling the brain goes, it is worth remembering:

"… metaphors for the brain are usually based on the most complex device currently available: in the seventeenth century the brain was compared to a hydraulic system, and in the early twentieth century to a telephone switchboard. Now, of course, we compare the brain to a digital computer."

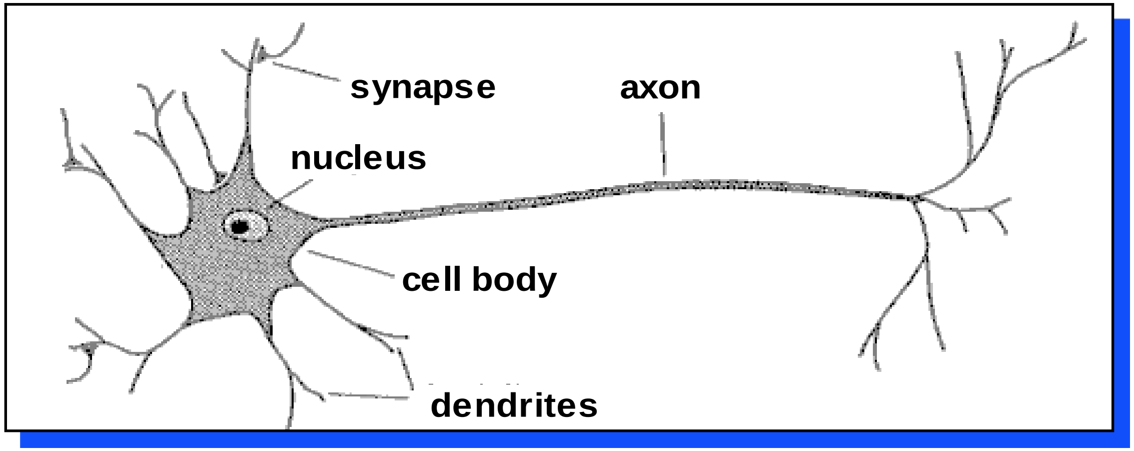
Biological inspirations

- The brain uses massively parallel computation
  - The human brain contains about 10 billion ($\sim 10^{11}$ neurons) nerve cells (neurons)
  - Each neuron is connected to the others through 10000 synapses ($\sim 10^{4}$ connections per neuron)

- Neurons respond slowly
  - 10^{-3} s compared to 10^{-9} s for electrical circuits

- Properties of the brain
  - It can learn, reorganize itself from experience
  - It adapts to the environment
  - It is robust and fault tolerant

Biological neuron



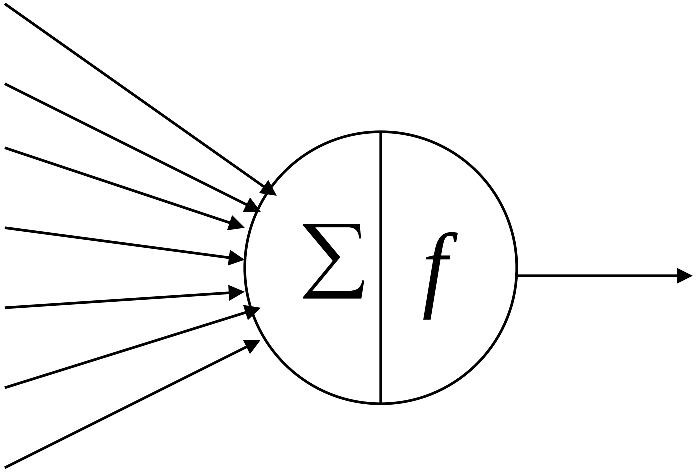
- A neuron has
  - A branching input (dendrites)
  - A branching output (the axon)
- The information circulates from the dendrites to the axon via the cell body
- Axon connects to dendrites via synapses
  - Synapses vary in strength
  - Synapses may be excitatory or inhibitory

Artificial Neural Networks

- Historically, ANN theories were first developed by neurophysiologists. For engineers (and others), the attractions of ANN processing include:
  - inherent parallelism
  - speed (avoiding the von Neumann bottleneck)
  - distributed "holographic" storage of information
  - robustness
  - generalization
  - learning by example rather than having to understand the underlying problem (a double-edged sword!)
- Remember that every ANN is a mathematical model. There is usually a good statistical explanation of ANN behaviour

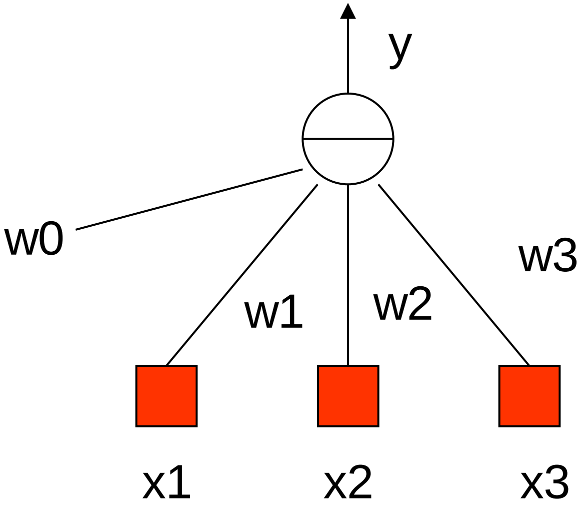
What is a neuron?

- **a (biological) neuron is a** node **that has** many inputs **and *one*** output
- **inputs come from other neurons or sensory organs**
- **the inputs are weighted**
- weights **can be both positive and negative**
- **inputs are** summed **at the node to produce an** activation **value**
- **if the activation is greater than some** threshold**, the neuron** fires



What is a neuron ? - 1

- Definition : Non linear, parameterized function with restricted output range



$$y = f\left(w_0 + \sum_{i=1}^{n-1} w_i x_i \right)$$

What is a neuron?

- In order to simulate neurons on a computer, we need a mathematical model of this node
  - **node *i* has *n*** inputs ***x_j***
  - **each** connection **has an associated** weight ***w_{ij}***
  - **the** net input **to node *i* is the sum of the products of the connection inputs and their weights:**

$$net_i = \sum_{j=1}^{n} w_{ij} x_j$$

  - **The** output **of node *i* is determined by applying a** non-linear transfer function ***f*** **to the net input:**

$$x_i = f(net_i)$$

What is a neuron?

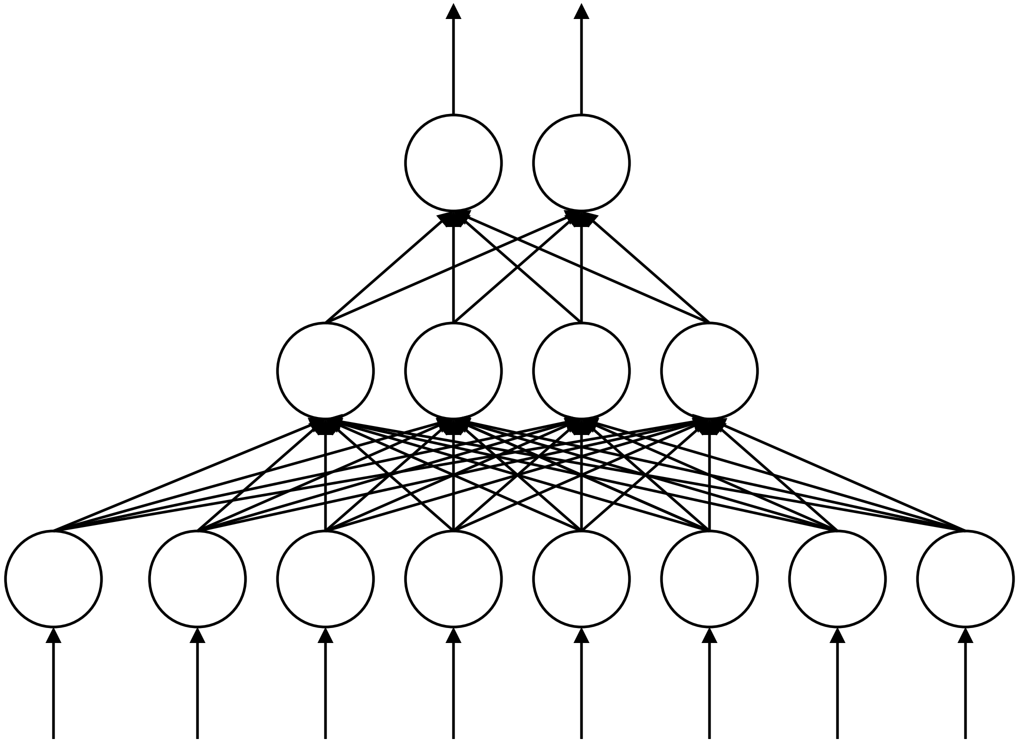
- A common choice for the transfer (activation) function is the sigmoid:

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

- The sigmoid has similar non-linear properties to the transfer function of real neurons:
  - bounded below by 0
  - saturates when input becomes large
  - bounded above by 1

What is a neural network?

- Now that we have a model for an artificial neuron, we can imagine connecting many of then together to form an Artificial Neural Network:



Output layer

Hidden layer

Input layer

Learning

- The procedure that consists in estimating the parameters of neurons so that the whole network can perform a specific task

- The Learning process (supervised)
  - Present the network a number of inputs and their corresponding outputs
  - See how closely the actual outputs match the desired ones
  - Modify the parameters (weights) to better approximate the desired outputs

Applications of ANNs

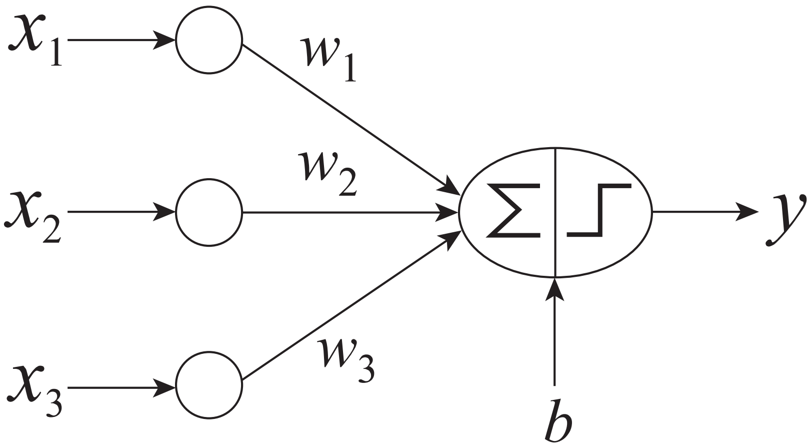
- Predicting financial time series
- Diagnosing medical conditions
- Identifying clusters in customer databases
- Identifying fraudulent credit card transactions
- Hand-written character recognition (cheques)
- Predicting the failure rate of machinery
- Natural Language Processing
- and many more….

Artificial Neural Networks (ANN)

- **Basic Idea:** A complex non-linear function can be learned as a composition of simple processing units

- Simplest ANN: **Perceptron** (single neuron)

Basic Architecture of Perceptron



$$y = \begin{cases} 1, & \text{if } \mathbf{w}^T\mathbf{x} + b > 0. \\ -1, & \text{otherwise.} \end{cases}$$

$$\tilde{\mathbf{w}} = (\mathbf{w}^T \;\; b)^T \qquad \tilde{\mathbf{x}} = (\mathbf{x}^T \;\; 1)^T$$

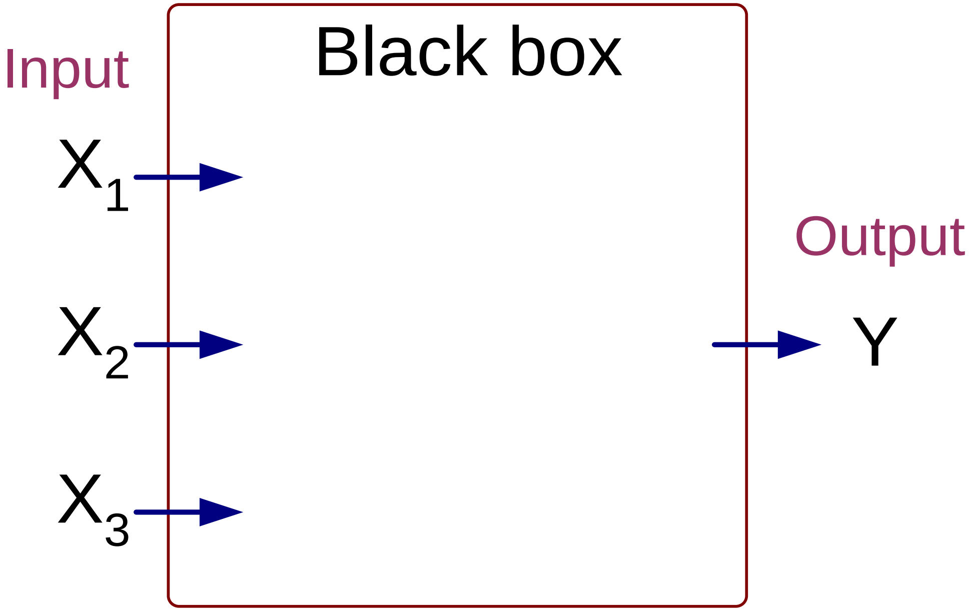
$$\hat{y} = sign(\tilde{\mathbf{w}}^T\tilde{\mathbf{x}})$$

Activation Function

- Learns linear decision boundaries
- Related to logistic regression (activation function is sign instead of sigmoid)

Perceptron Example

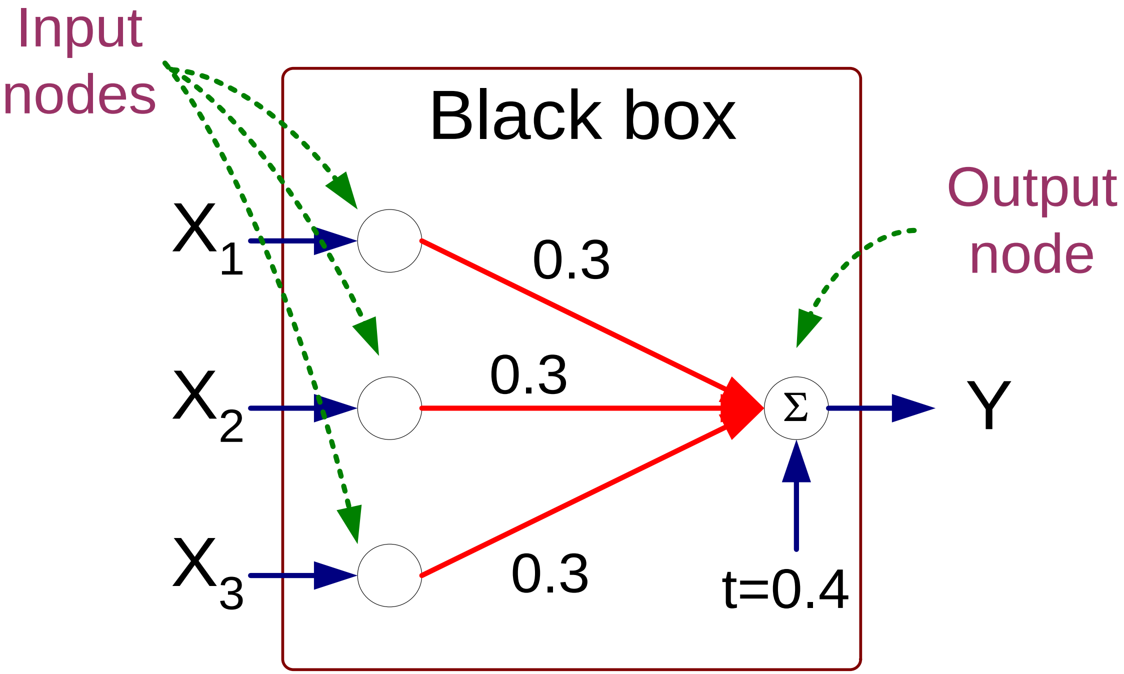
X_1	X_2	X_3	Y
1	0	0	-1
1	0	1	1
1	1	0	1
1	1	1	1
0	0	1	-1
0	1	0	-1
0	1	1	1
0	0	0	-1



Output Y is 1 if at least two of the three inputs are equal to 1.

Perceptron Example

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



$$Y = sign(0.3X_1 + 0.3X_2 + 0.3X_3 - 0.4)$$

$$\text{where } sign(x) = \begin{cases} 1 & \text{if } x \geq 0 \\ -1 & \text{if } x < 0 \end{cases}$$

Perceptron Learning Rule

- Initialize the weights (w_0, w_1, …, w_d)
- Repeat
  - For each training example (x_i, y_i)
    - Compute $\widehat{y}_i$
    - Update the weights:

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

- Until stopping condition is met

- k: iteration number; λ: learning rate

Perceptron Learning Rule

- Weight update formula:

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

- Intuition:
  - Update weight based on error: e = $(y_i - \hat{y}_i)$
    - If y = $\hat{y}$, e=0: no update needed
    - If y > $\hat{y}$, e=2: weight must be increased (assuming x_{ij} is positive) so that $\hat{y}$ will increase
    - If y < $\hat{y}$, e=-2: weight must be decreased (assuming x_{ij} is positive) so that $\hat{y}$ will decrease

Example of Perceptron Learning

$$\lambda = 0.1$$

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

	w_0	w_1	w_2	w_3
0	0	0	0	0
1	-0.2	-0.2	0	0
2	0	0	0	0.2
3	0	0	0	0.2
4	0	0	0	0.2
5	-0.2	0	0	0
6	-0.2	0	0	0
7	0	0	0.2	0.2
8	-0.2	0	0.2	0.2

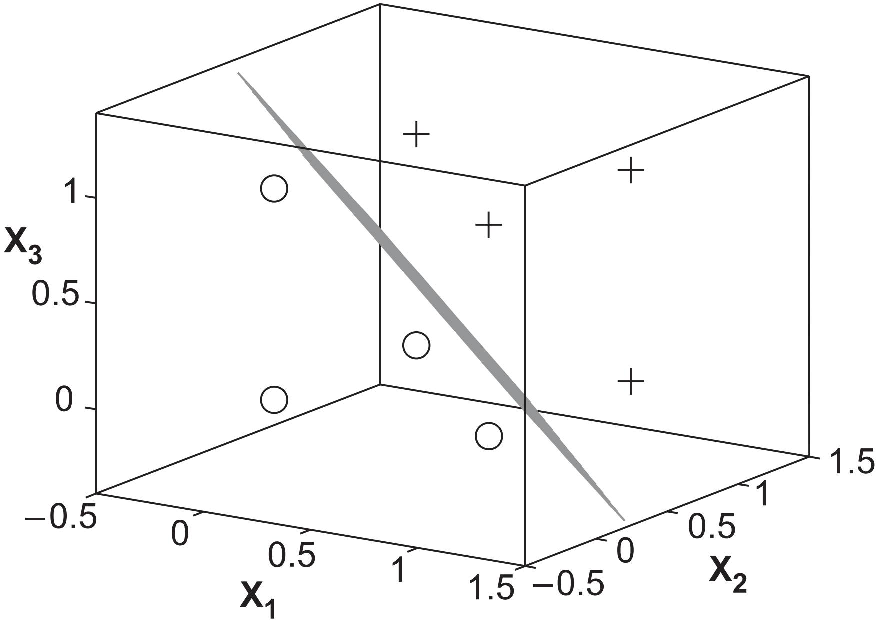
Weight updates over first epoch

Epoch	w_0	w_1	w_2	w_3
0	0	0	0	0
1	-0.2	0	0.2	0.2
2	-0.2	0	0.4	0.2
3	-0.4	0	0.4	0.2
4	-0.4	0.2	0.4	0.4
5	-0.6	0.2	0.4	0.2
6	-0.6	0.4	0.4	0.2

Weight updates over
all epochs

Perceptron Learning

- Since y is a linear combination of input variables, decision boundary is linear

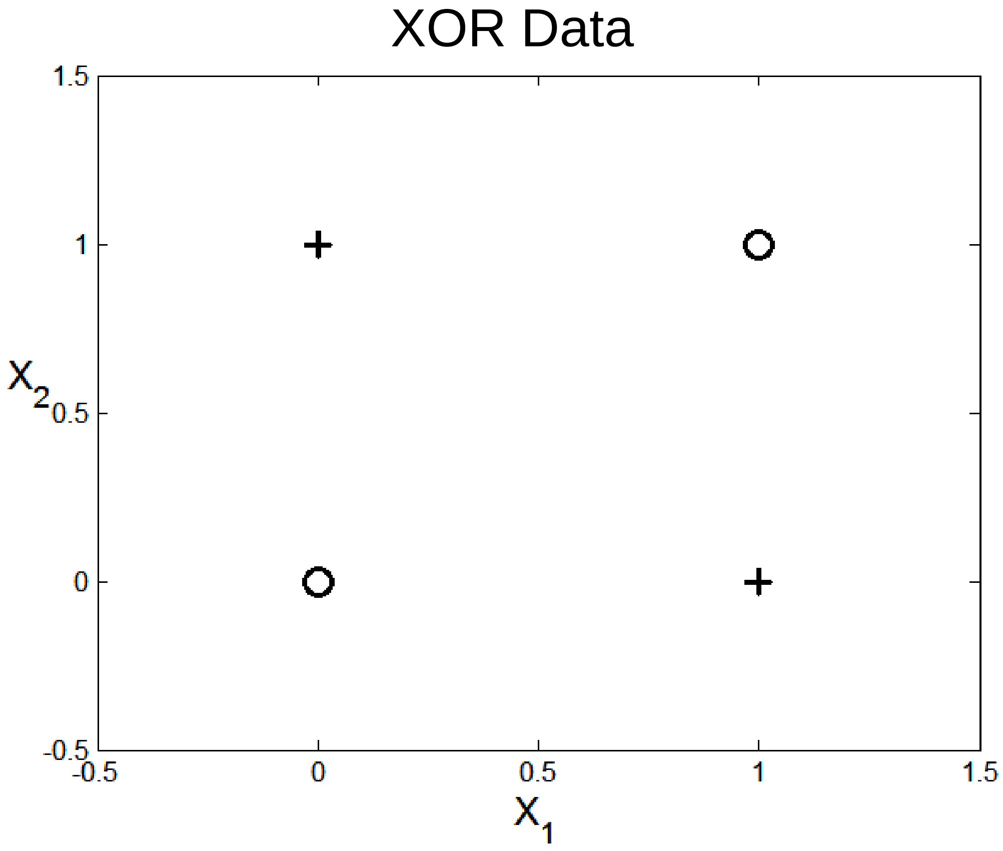


Nonlinearly Separable Data

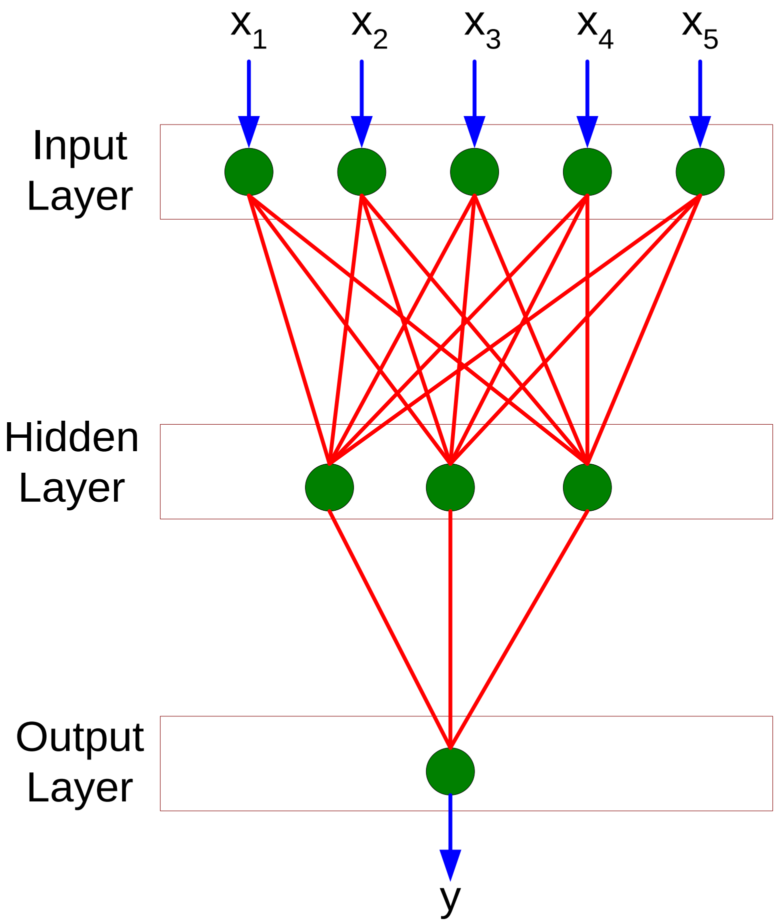
For nonlinearly separable problems, perceptron learning algorithm will fail because no linear hyperplane can separate the data perfectly

$$y = x_1 \oplus x_2$$

x_1	x_2	y
0	0	-1
1	0	1
0	1	1
1	1	-1



Multi-layer Neural Network



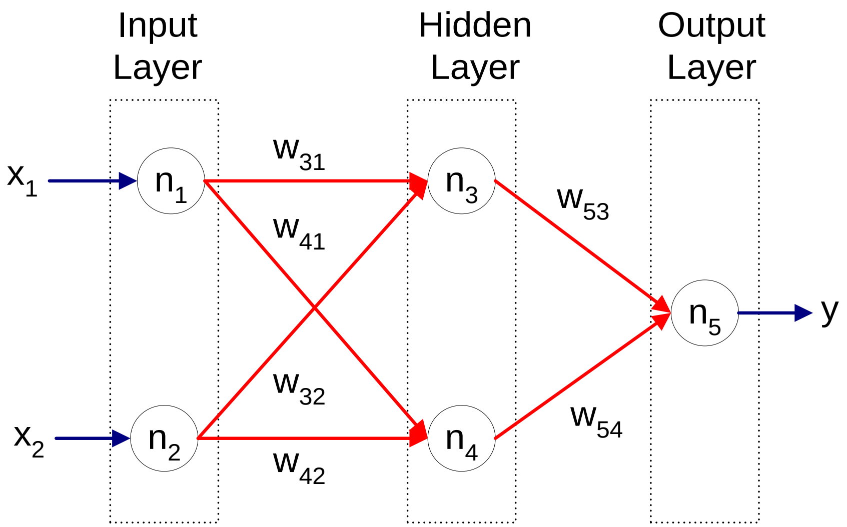
- More than one *hidden layer* of computing nodes

- Every node in a hidden layer operates on activations from preceding layer and transmits activations forward to nodes of next layer

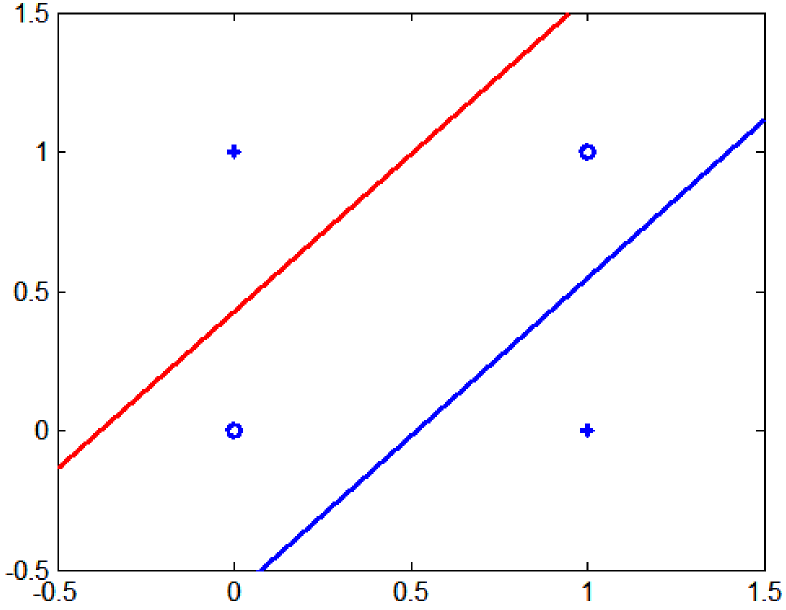
- Also referred to as "feedforward neural networks"

Multi-layer Neural Network

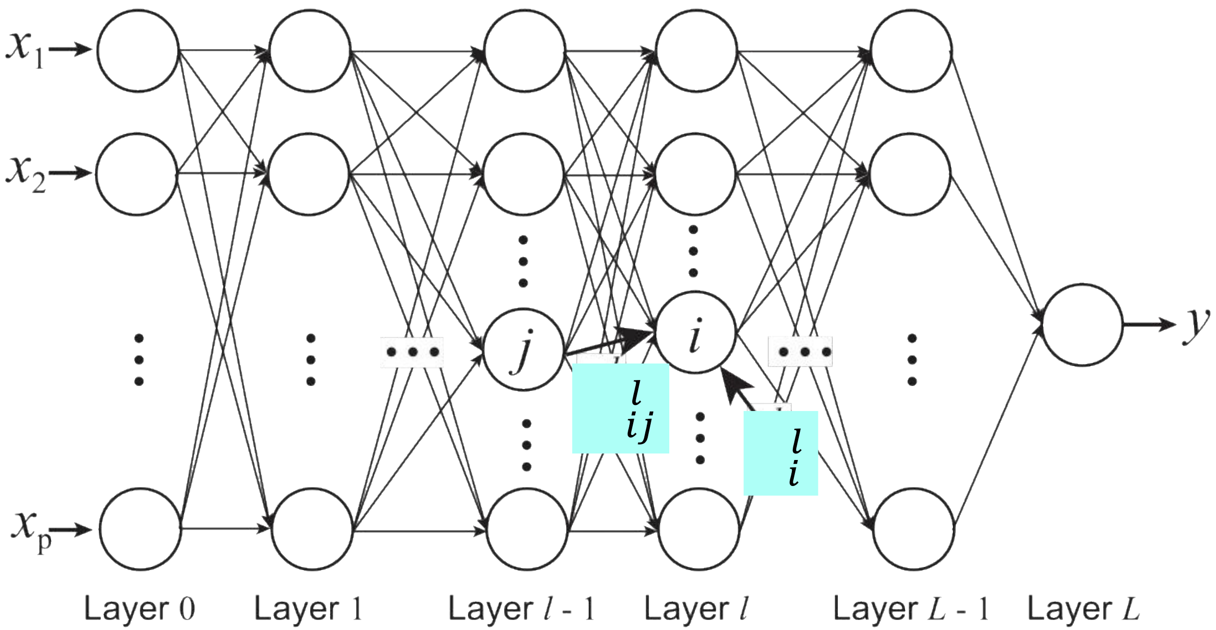
- Multi-layer neural networks with at least one hidden layer can solve any type of classification task involving nonlinear decision surfaces



XOR Data



Multi-Layer Network Architecture with Many Hidden Layers



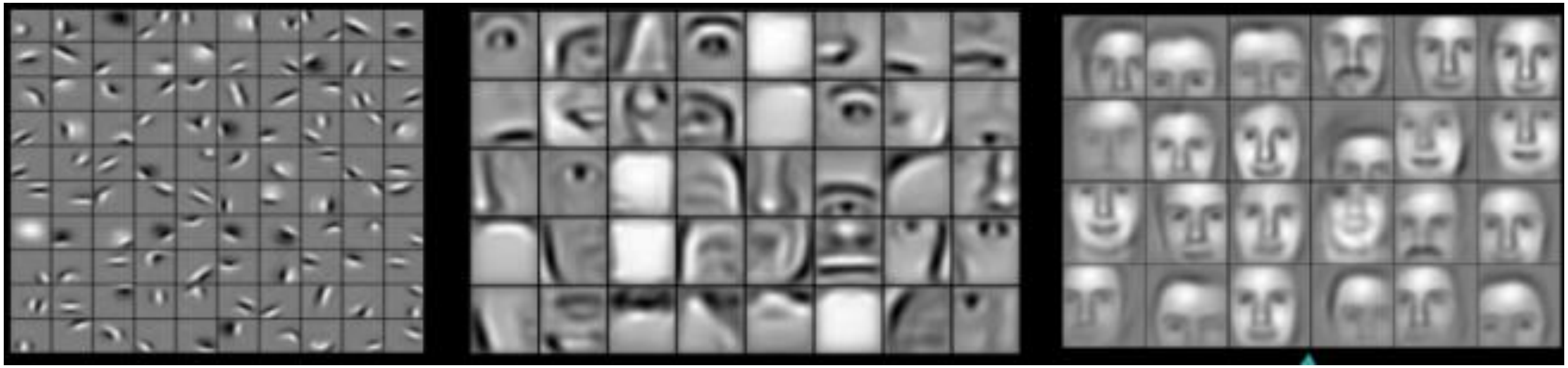
$$a_i^l = f(z_i^l) = f\Big(\underbrace{\sum_j w_{ij}^l a_j^{l-1} + b_i^l}_{\text{Linear Predictor}}\Big)$$

Activation value at node i at layer l

Activation Function

Why Multiple Hidden Layers?

- Activations at hidden layers can be viewed as features extracted as functions of inputs
- Every hidden layer represents a level of abstraction
  - *Complex features are compositions of simpler features*



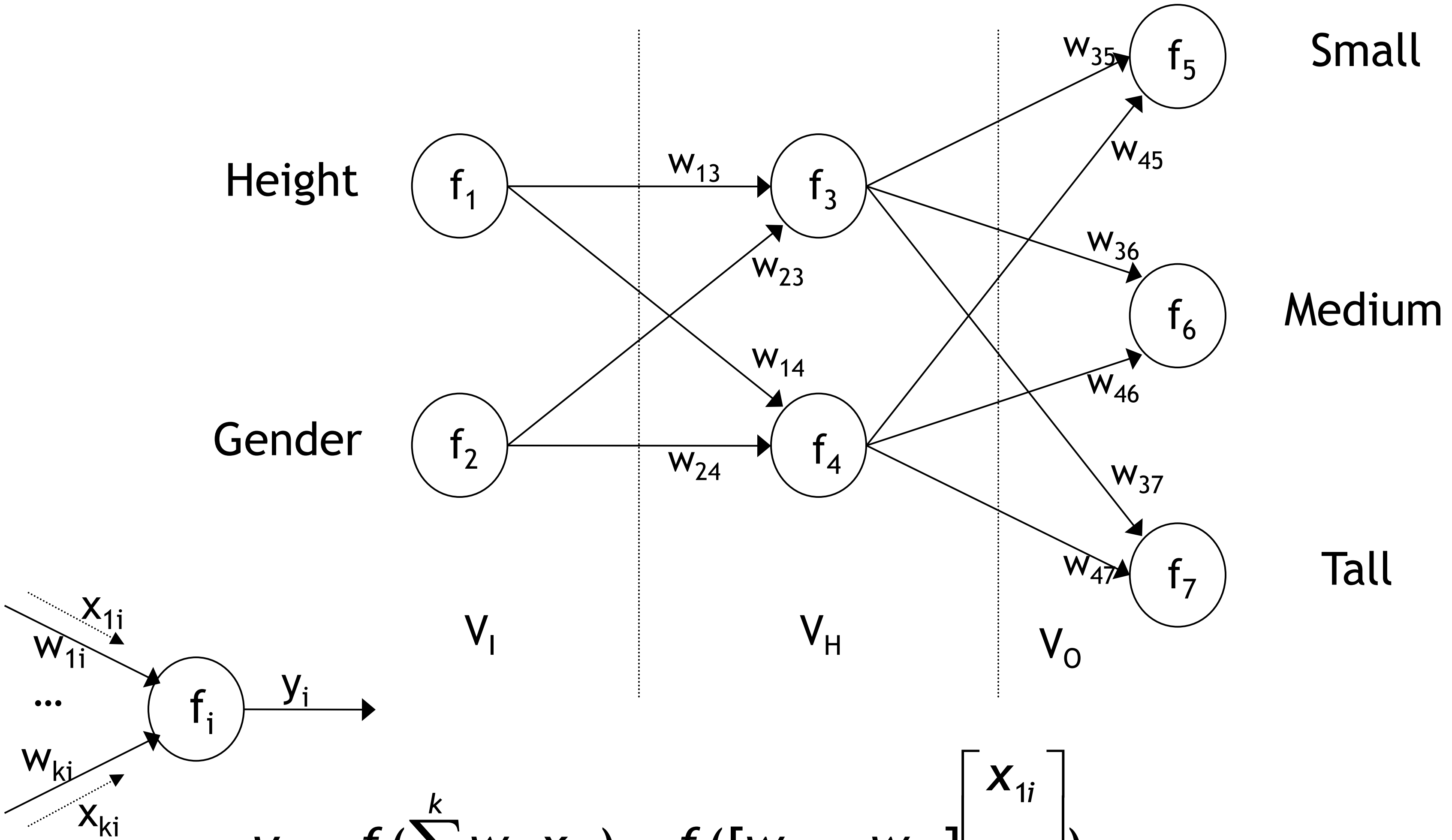
- Number of layers is known as **depth** of ANN
  - *Deeper networks express complex hierarchy of features*

Neural Networks in Detail

Definition:

- NN is a directed graph $F=<V,E>$ with vertices $V=\{1,2,\ldots,n\}$ and edges $E=\{<i,j> \mid 1 \leq i,j \leq n\}$, with the following restrictions:
  - V is partitioned into a set of input nodes, V_I, hidden nodes, V_H, and output nodes, V_O.
  - The vertices are also partitioned into layers. There can be more than one hidden layers.
  - Any edge $<i,j>$ must have node i in layer h-1 and node j in layer h. (Note: In some advanced NNs, it is possible to have the edges connect nodes at arbitrary layers.)
  - Edge $<i,j>$ is labeled with a numeric value w_{ij}.
  - Node i is labeled with a function f_i.

NN Example



$$y_i = f_i(\sum_{j=1}^{k} w_{ji} x_{ji}) = f_i([w_{1i} \dots w_{ki}] \begin{bmatrix} x_{1i} \\ \cdots \\ x_{ki} \end{bmatrix})$$

NN Activation Functions

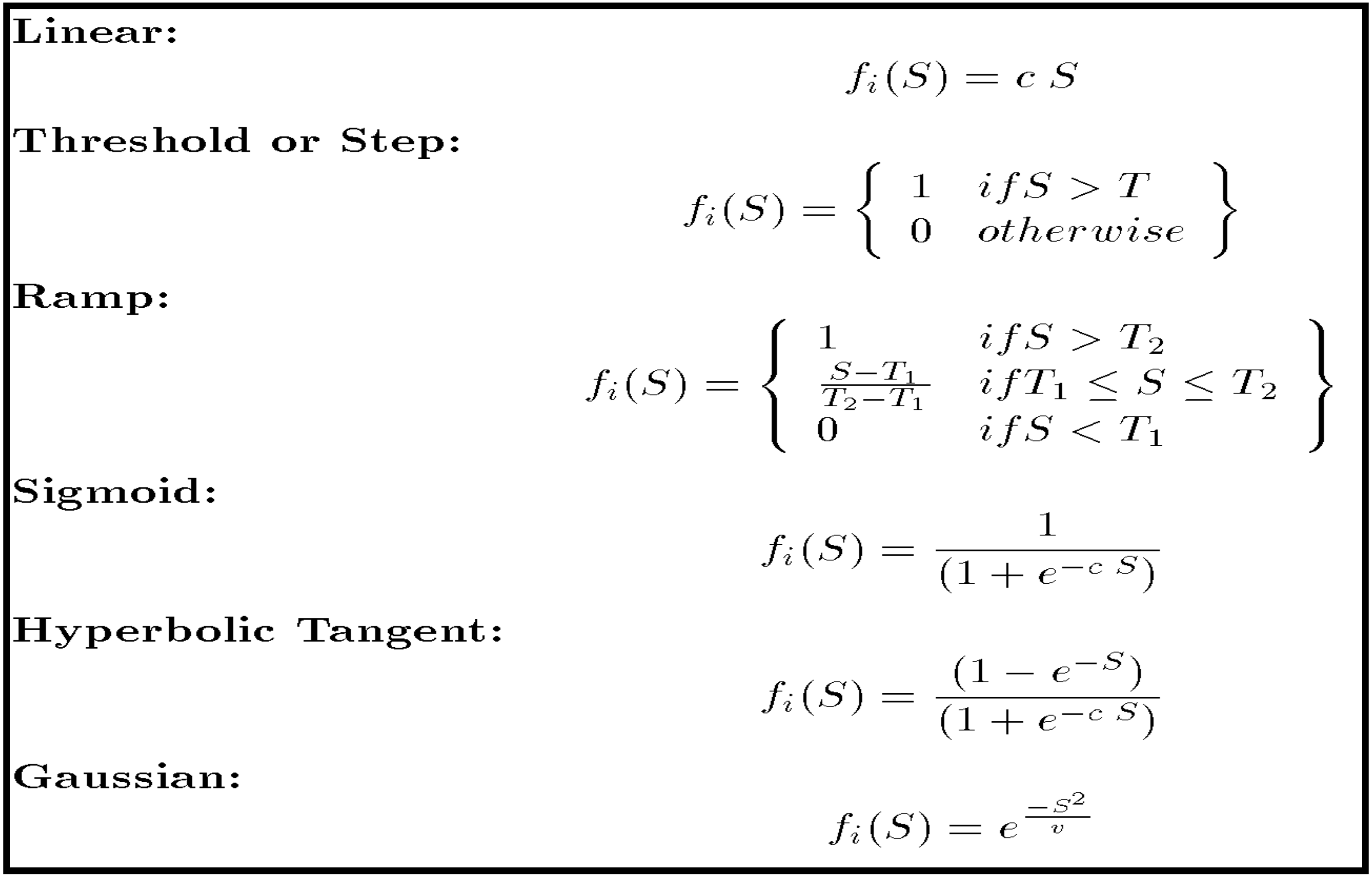
- Sometimes called *processing element function*, *firing rule*.
- Functions associated with nodes in graph, using sets of inputs and weights. Usually in the form of sum of products:

$$S = w_{oi} + (\sum_{h=1}^{k} (w_{hi} x_{hi}))$$

$$w_{oi} \text{ is bias input (optional)}$$

- Output may be in range [-1,1] (*bipolar*) or [0,1] (*unipolar*)

NN Activation (Transfer)Functions



Linear:

$$f_i(S) = c\ S$$

Threshold or Step:

$$f_i(S) = \left\{ \begin{array}{ll} 1 & if S > T \\ 0 & otherwise \end{array} \right\}$$

Ramp:

$$f_i(S) = \left\{ \begin{array}{ll} 1 & if S > T_2 \\ \frac{S-T_1}{T_2-T_1} & if T_1 \leq S \leq T_2 \\ 0 & if S < T_1 \end{array} \right\}$$

Sigmoid:

$$f_i(S) = \frac{1}{(1 + e^{-c\ S})}$$

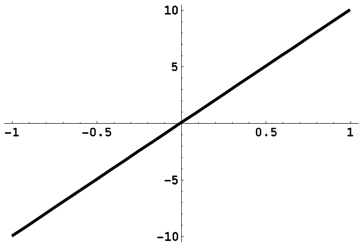
Hyperbolic Tangent:

$$f_i(S) = \frac{(1 - e^{-S})}{(1 + e^{-c\ S})}$$

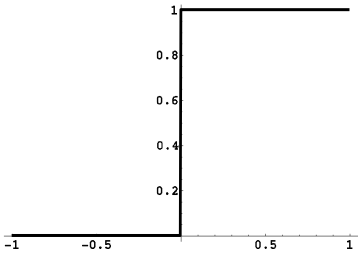
Gaussian:

$$f_i(S) = e^{\frac{-S^2}{v}}$$

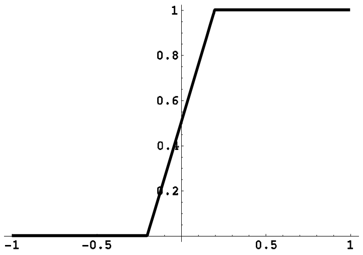
NN Activation Functions (cont.)



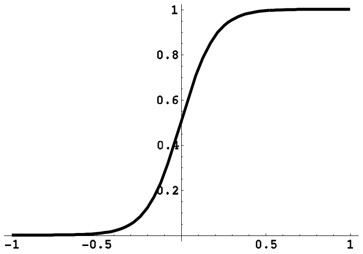
Linear



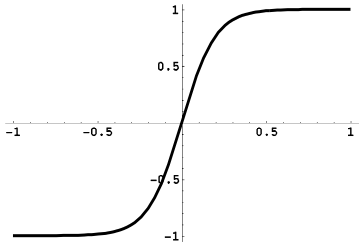
Step/Threshold



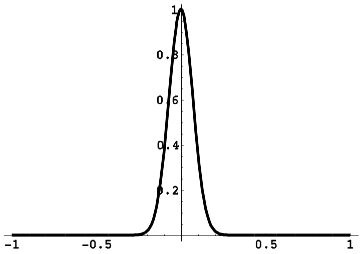
Ramp



Sigmoid



Hyperbolic tangent



Gaussian

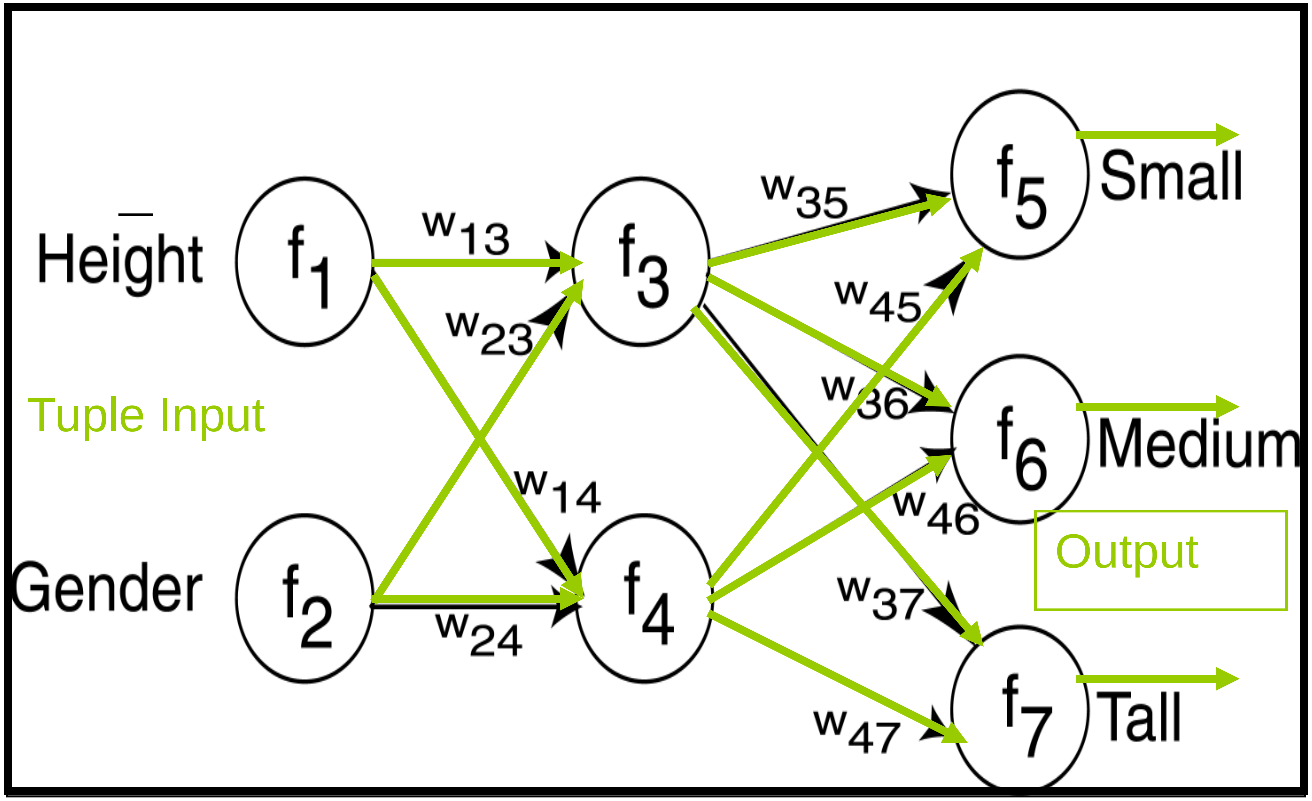
NN Model

- A *neural network model* is a computational model consisting of three parts:
  - Neural Network graph
  - Learning algorithm that indicates how learning takes place.
  - Recall techniques that determine how information is obtained from the network.
- We will look at propagation as a recall technique.

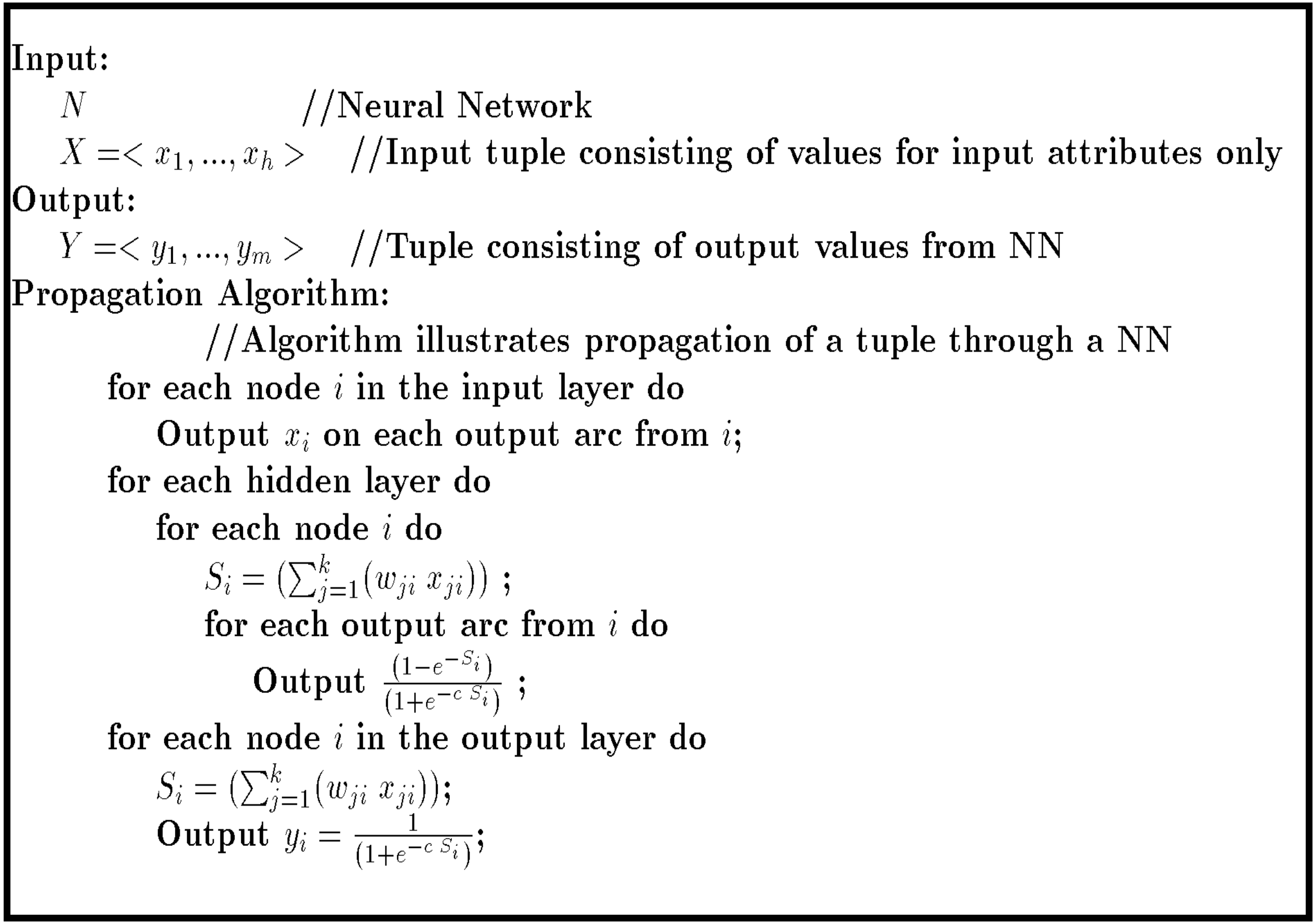
Classification Using Neural Networks

- Typical NN structure for classification:
  - One output node per class
  - Output value is class membership function value
- Supervised learning
  - For each tuple in training set, propagate it through NN.
  - Adjust weights on edges to improve future classification.
- Algorithms:
  - Propagation (recall),
  - Backpropagation (learning) ,
  - Gradient Descent (technique to modify the weights in the graph)

Propagation

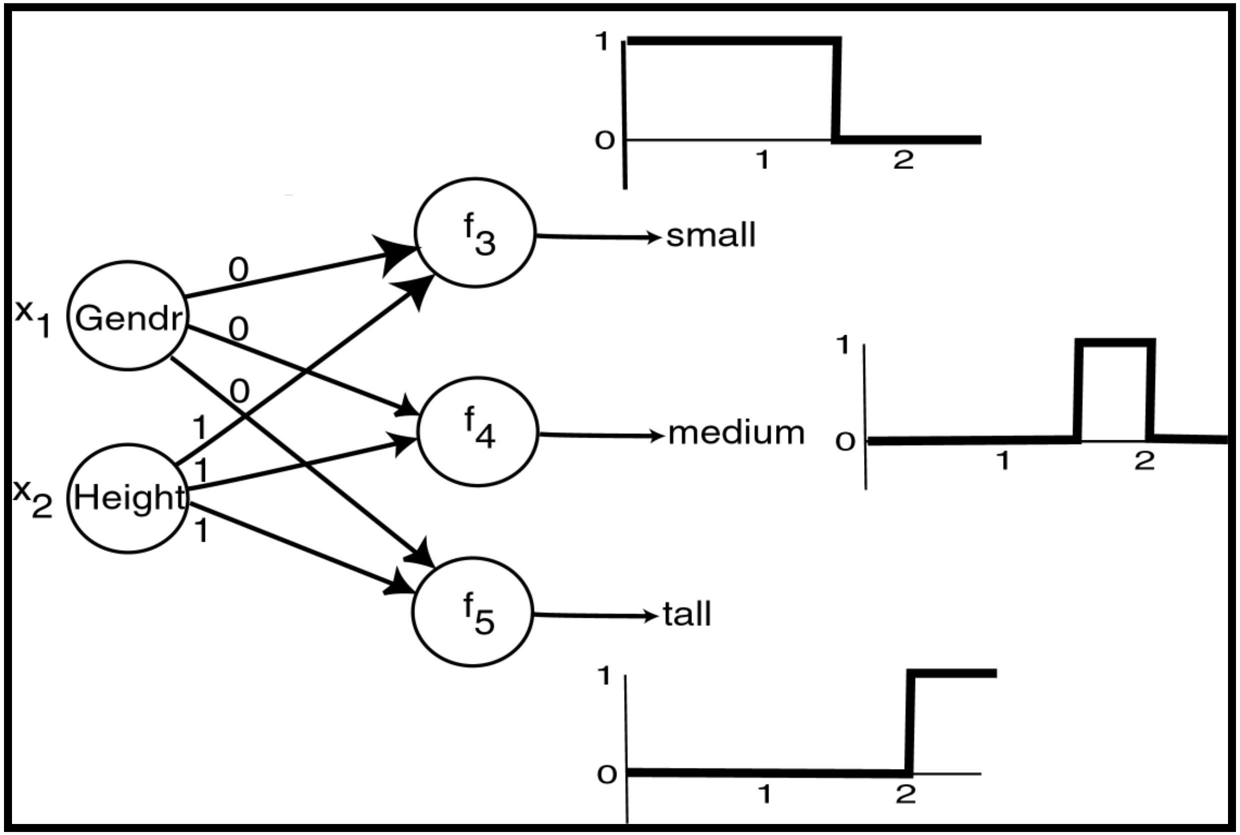


NN Propagation Algorithm



```
Input:
    $N$                    //Neural Network
    $X = < x_1, ..., x_h >$    //Input tuple consisting of values for input attributes only
Output:
    $Y = < y_1, ..., y_m >$    //Tuple consisting of output values from NN
Propagation Algorithm:
                //Algorithm illustrates propagation of a tuple through a NN
        for each node $i$ in the input layer do
            Output $x_i$ on each output arc from $i$;
        for each hidden layer do
            for each node $i$ do
                $S_i = (\sum_{j=1}^{k}(w_{ji}\, x_{ji}))$ ;
                for each output arc from $i$ do
                    Output $\frac{(1-e^{-S_i})}{(1+e^{-c\, S_i})}$ ;
        for each node $i$ in the output layer do
            $S_i = (\sum_{j=1}^{k}(w_{ji}\, x_{ji}))$;
            Output $y_i = \frac{1}{(1+e^{-c\, S_i})}$;
```

Example Propagation

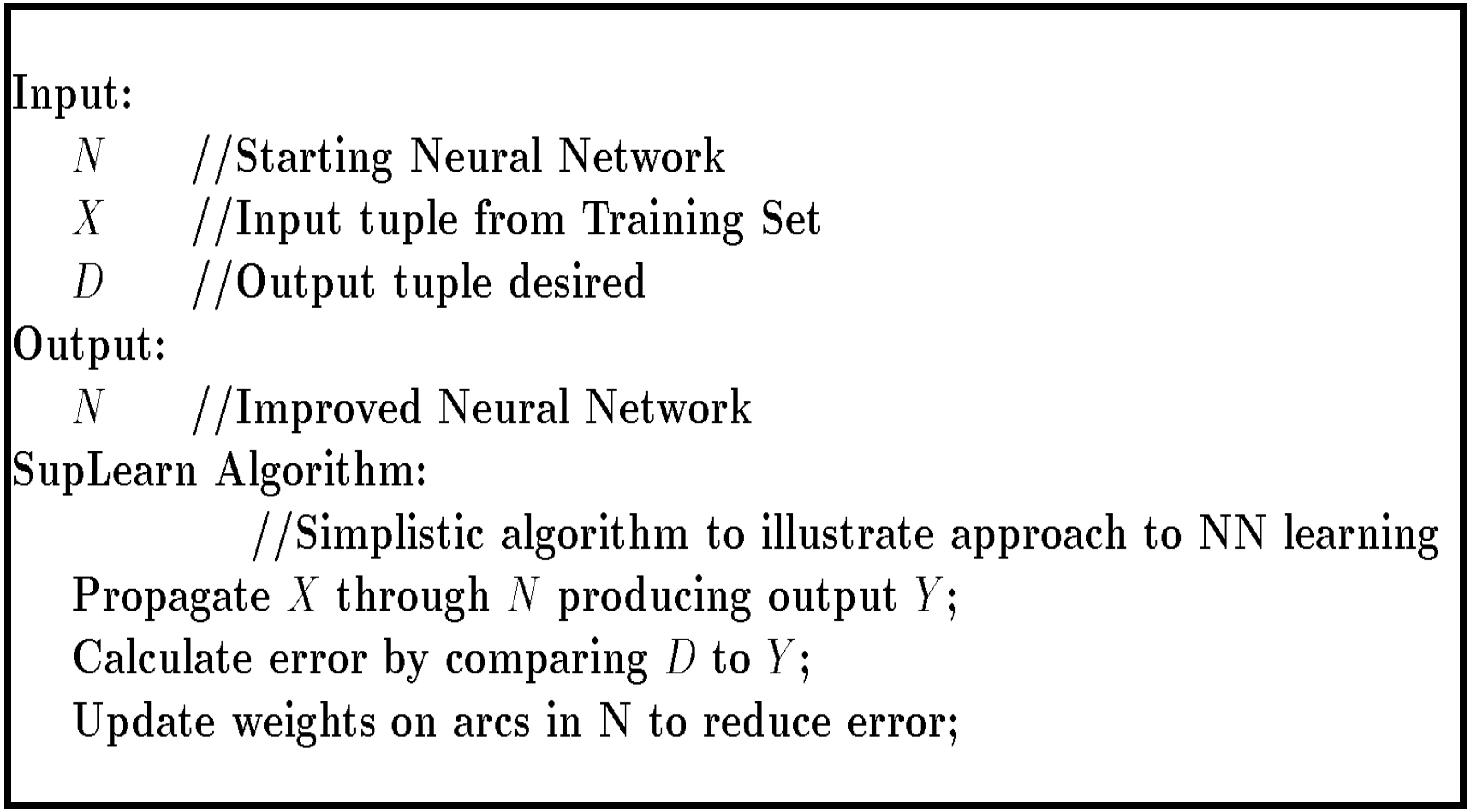


© Prentie Hall

NN Learning

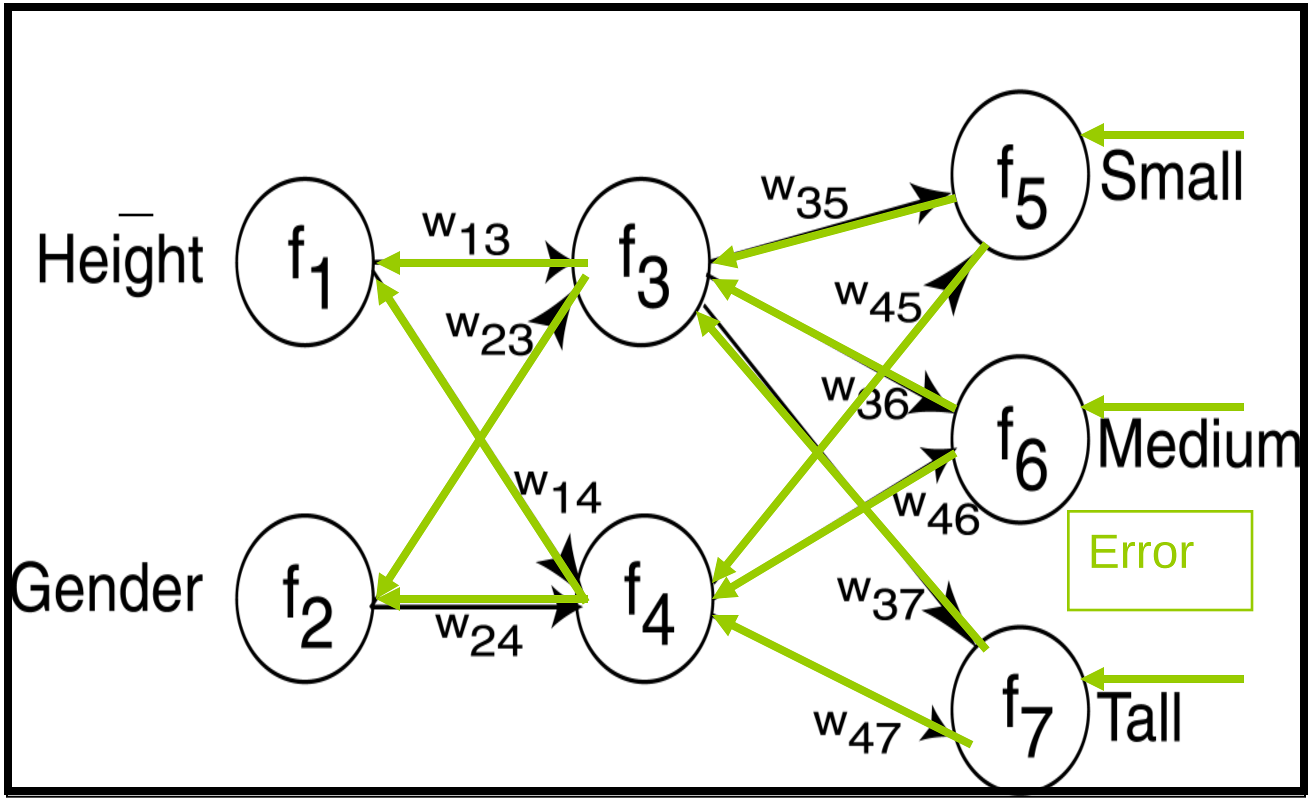
- Propagate input values through graph.
- Compare output to desired output.
- Adjust weights in graph accordingly.
  - Backpropagation – adjust the weights of each edge, starting backward from the edges that connect the output-hidden layer.

NN Supervised Learning



```
Input:
    $N$       //Starting Neural Network
    $X$       //Input tuple from Training Set
    $D$       //Output tuple desired
Output:
    $N$       //Improved Neural Network
SupLearn Algorithm:
                //Simplistic algorithm to illustrate approach to NN learning
    Propagate $X$ through $N$ producing output $Y$;
    Calculate error by comparing $D$ to $Y$;
    Update weights on arcs in N to reduce error;
```

Backpropagation



NN Supervised Learning

- Training the nets begins with assigning each arc a **small random positive weight**.
- Feed training examples into the net, one by one. (Each training example is marked with the desired output.)
- Compute the actual output for each example by feeding each attribute value into the relevant node, multiplying by the appropriate arc weights, summing weighted inputs, and applying transfer functions to obtain outputs at each level.
- Compare actual output at final output level to desired output for each example. (This is called '*supervised*' learning as the learning method can compare actual outputs with desired outputs.)
- Adjust weights by small fraction so as to minimize the error. Error may be the simple squared difference between actual and desired output, or some other more complex error function.
- Continue feeding examples into net and adjusting weights (usually feeding the training set multiple times).

Supervised Learning

- Possible error values assuming output from node i is y_i but should be d_i:
- The mean squared error (MSE) can be used to find the error.

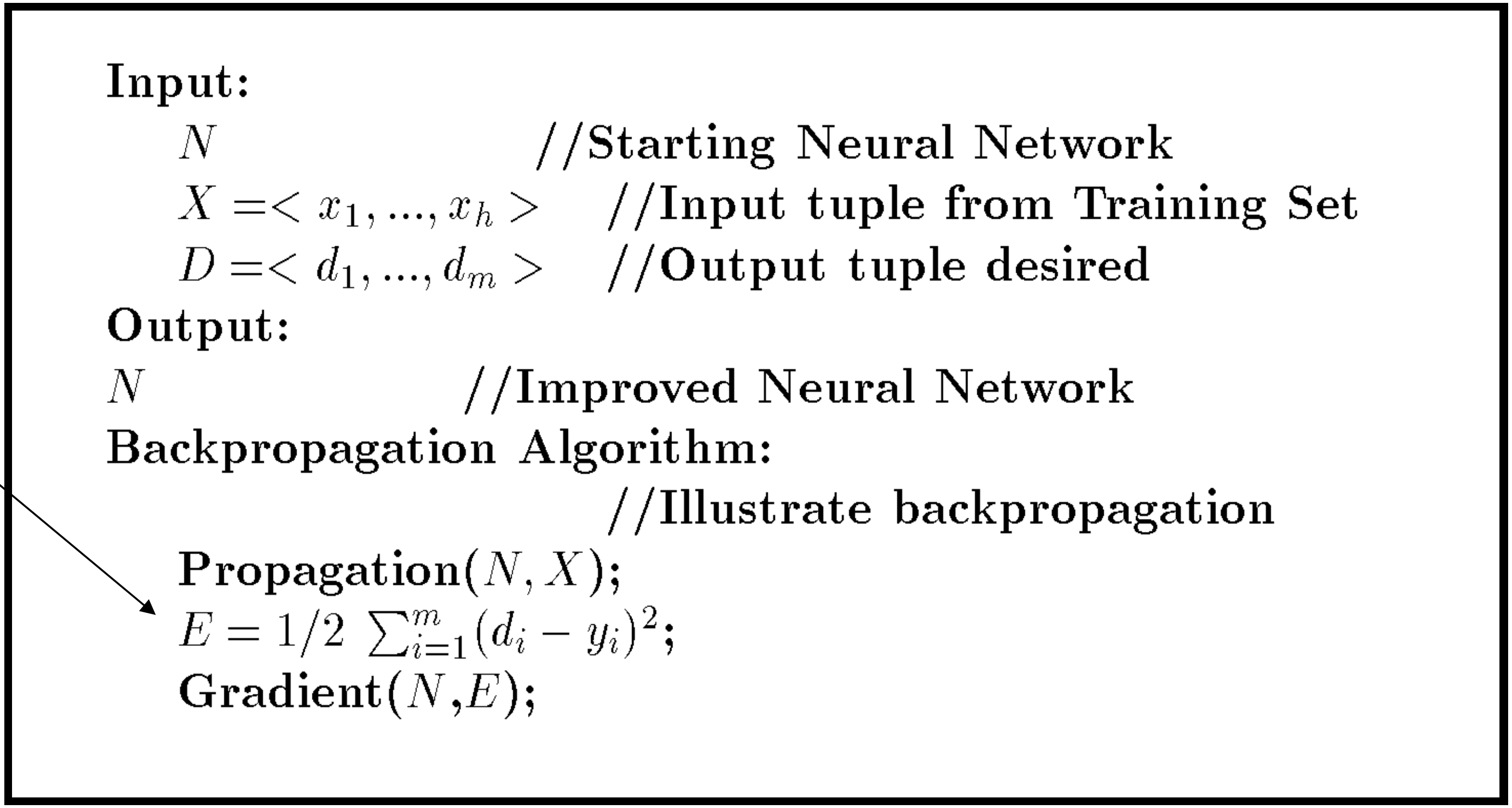
$$\mid y_i - d_i \mid$$

$$\frac{(y_i - d_i)^2}{2}$$

$$\sum_{i=1}^{m} \frac{(y_i - d_i)^2}{m}$$

- Change weights on arcs based on estimated error

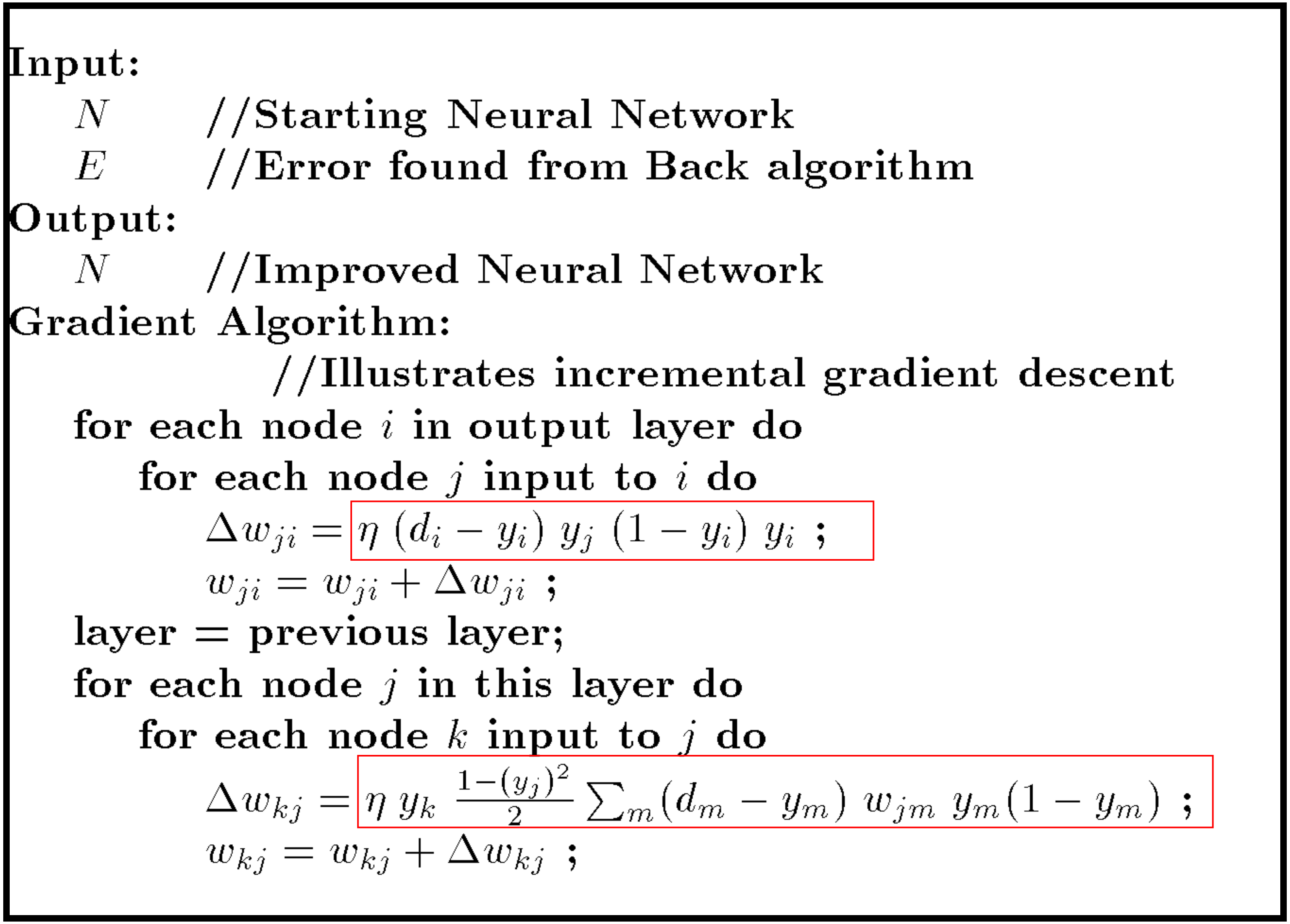
Backpropagation Algorithm



```
Input:
    $N$                              //Starting Neural Network
    $X =< x_1, ..., x_h >$           //Input tuple from Training Set
    $D =< d_1, ..., d_m >$           //Output tuple desired
Output:
$N$                                  //Improved Neural Network
Backpropagation Algorithm:
                                     //Illustrate backpropagation
    Propagation($N, X$);
    $E = 1/2 \sum_{i=1}^{m} (d_i - y_i)^2$;
    Gradient($N$,$E$);
```

MSE

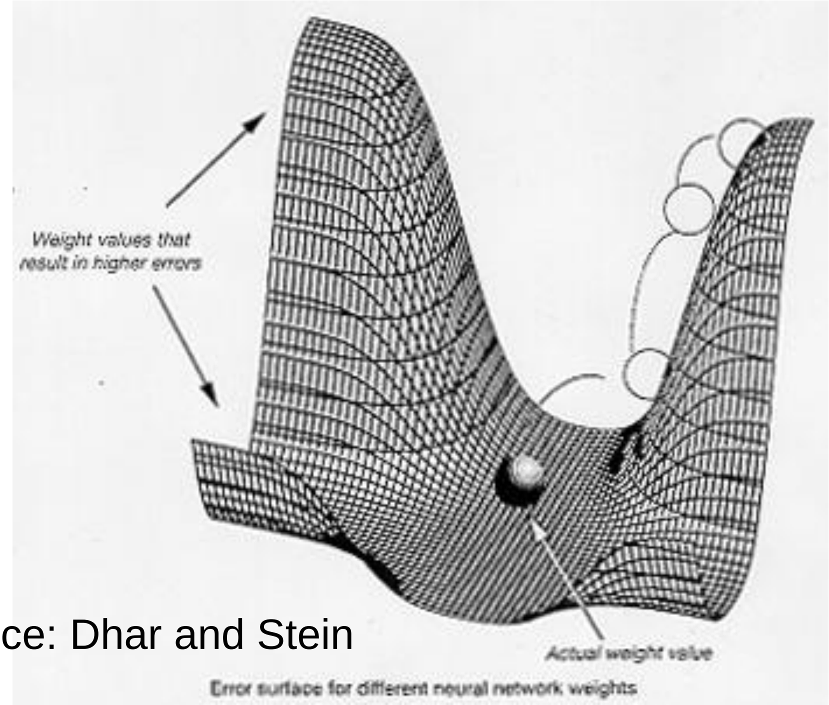
Gradient Descent Algorithm



```
Input:
    $N$        //Starting Neural Network
    $E$        //Error found from Back algorithm
Output:
    $N$        //Improved Neural Network
Gradient Algorithm:
                //Illustrates incremental gradient descent
    for each node $i$ in output layer do
        for each node $j$ input to $i$ do
            $\Delta w_{ji} = \boxed{\eta\ (d_i - y_i)\ y_j\ (1 - y_i)\ y_i\ ;}$
            $w_{ji} = w_{ji} + \Delta w_{ji}\ ;$
    layer = previous layer;
    for each node $j$ in this layer do
        for each node $k$ input to $j$ do
            $\Delta w_{kj} = \boxed{\eta\ y_k\ \frac{1-(y_j)^2}{2} \sum_m (d_m - y_m)\ w_{jm}\ y_m (1 - y_m)\ ;}$
            $w_{kj} = w_{kj} + \Delta w_{kj}\ ;$
```

Back-propagation

- **Back-propagation** is the process of **adjusting weights** to minimize the output error. The technique used is **gradient descent**: differentiate the transfer functions and, for each weight, decide whether to add or subtract a little in order to reduce the total error.



Error surface for different neural network weights

Source: Dhar and Stein

Assume we can adjust two weights (w_1 and w_2). The surface on the left is an example of how the error may vary as a result of different combinations of these weights.

Notice that the direction (positive or negative) and steepness of the slopes can be obtained using partial derivatives of the error with respect to w_1 and w_2

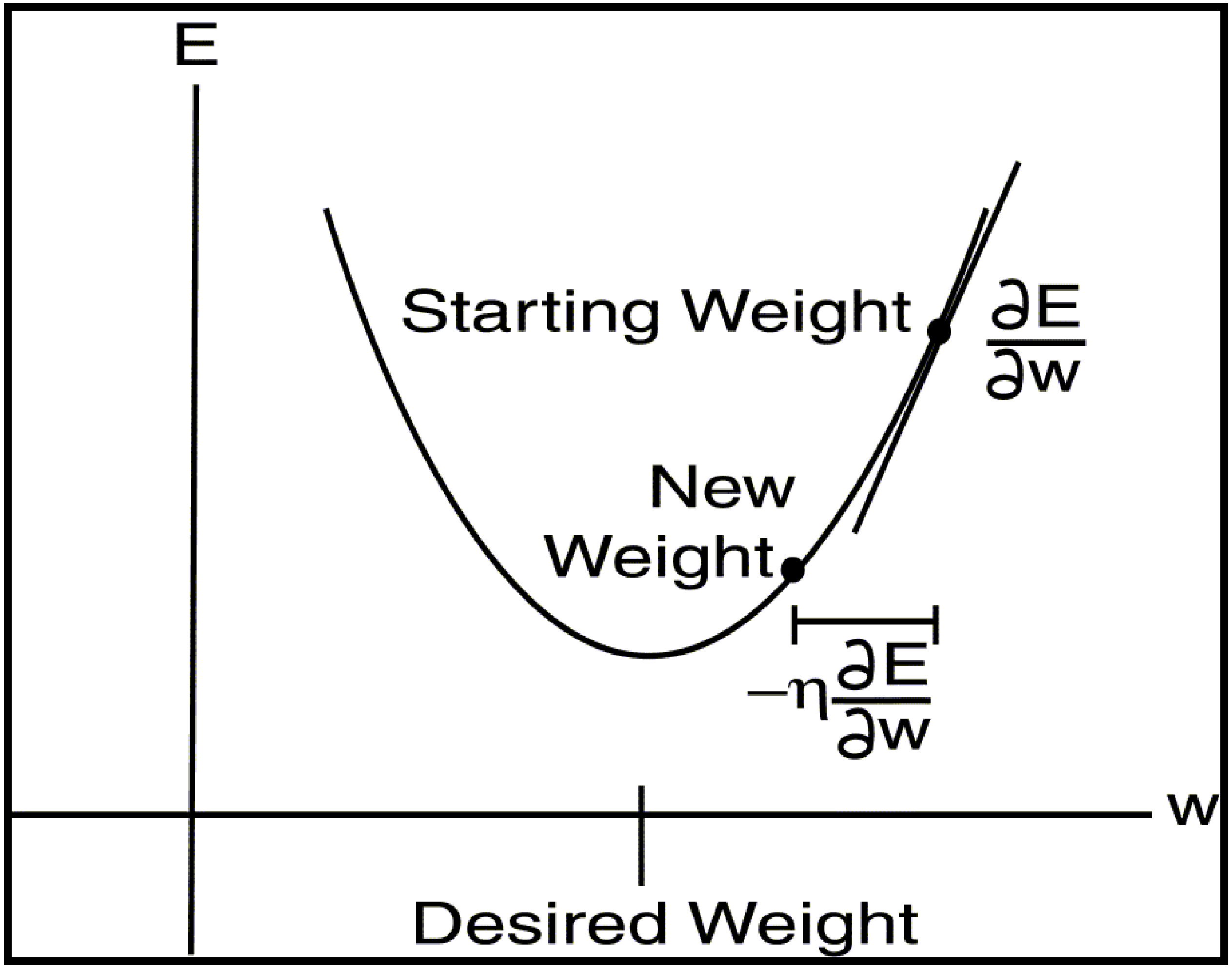
Gradient Descent

- The basic idea of gradient descent is to find the set of weights that minimizes MSE.
- The derivative of E gives the slope (or gradient) of the error fuction for one weight.

$$\Delta w_{ji} = -\eta \frac{\partial E}{\partial w_{ji}}$$

- We wish to find the weight where this slope is zero.
- The derivate finds the direction that reduces the error the most.
- The delta is added to the weight.
- η is refered to as the learing rate (or parameter). This value determines how fast the algorithm learns

Gradient Descent


E
Starting Weight
$\frac{\partial E}{\partial w}$
New Weight
$-\eta\frac{\partial E}{\partial w}$
w
Desired Weight

Output Layer Learning

$$\Delta w_{ji} = -\eta \, \frac{\partial E}{\partial w_{ji}} = -\eta \, \frac{\partial E}{\partial y_i} \, \frac{\partial y_i}{\partial S_i} \, \frac{\partial S_i}{\partial w_{ji}}$$

$$\frac{\partial y_i}{\partial S_i} = \frac{\partial}{\partial S_i}\left(\frac{1}{(1+e^{-S_i})}\right) = \left(1 - \left(\frac{1}{1+e^{-S_i}}\right)\right)\left(\frac{1}{1+e^{-S_i}}\right) = (1 - y_i)y_i$$

$$\frac{\partial S_i}{\partial w_{ji}} = y_j .$$

$$\frac{\partial E}{\partial y_i} = \frac{\partial}{\partial y_i}(1/2 \, \sum_m (d_m - y_m)^2) = -(d_i - y_i)$$

$$\Delta w_{ji} = \eta \, (d_i - y_i) \, y_j \, \left(1 - \frac{1}{1+e^{-S_i}}\right) \frac{1}{1+e^{-S_i}} = \boxed{\eta \, (d_i - y_i) \, y_j \, (1 - y_i) \, y_i}$$

Hidden Layer Learning

$$\Delta w_{kj} = -\eta \, \frac{\partial E}{\partial w_{kj}}$$

$$\frac{\partial E}{\partial w_{kj}} = \sum_m \frac{\partial E}{\partial y_m} \, \frac{\partial y_m}{\partial S_m} \, \frac{\partial S_m}{\partial y_j} \, \frac{\partial y_j}{\partial S_j} \, \frac{\partial S_j}{\partial w_{kj}}.$$

$$\frac{\partial E}{\partial y_m} = -(d_m - y_m)$$

$$\frac{\partial y_m}{\partial S_m} = (1 - y_m) y_m$$

$$\frac{\partial S_m}{\partial y_j} = w_{jm}$$

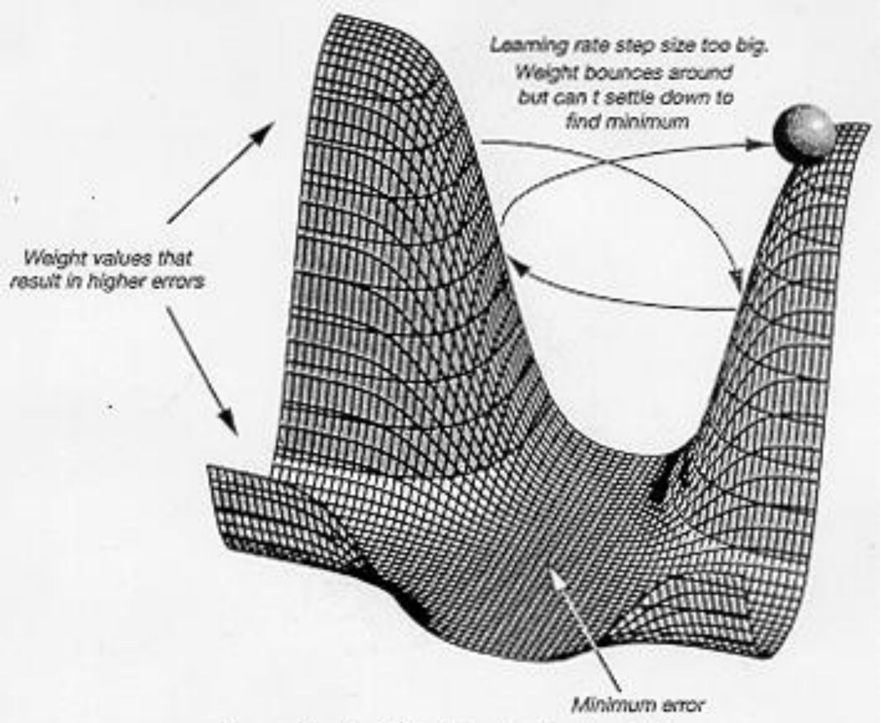
$$\frac{\partial y_j}{\partial S_j} = \frac{\partial}{\partial S_j}\left(\frac{(1 - e^{-S_j})}{(1 + e^{-S_j})}\right) = \frac{(1 + (\frac{1-e^{-S_j}}{1+e^{-S_j}}))\;(1 - (\frac{1-e^{-S_j}}{1+e^{-S_j}}))}{2} = \frac{1 - y_j^2}{2}$$

$$\frac{\partial S_j}{\partial w_{kj}} = y_k$$

$$\Delta w_{kj} = \boxed{\eta \; y_k \; \frac{1 - (y_j)^2}{2} \sum_m (d_m - y_m) \; w_{jm} \; y_m \; (1 - y_m)}$$

Problems
Rate of Gradient Descent too Fast

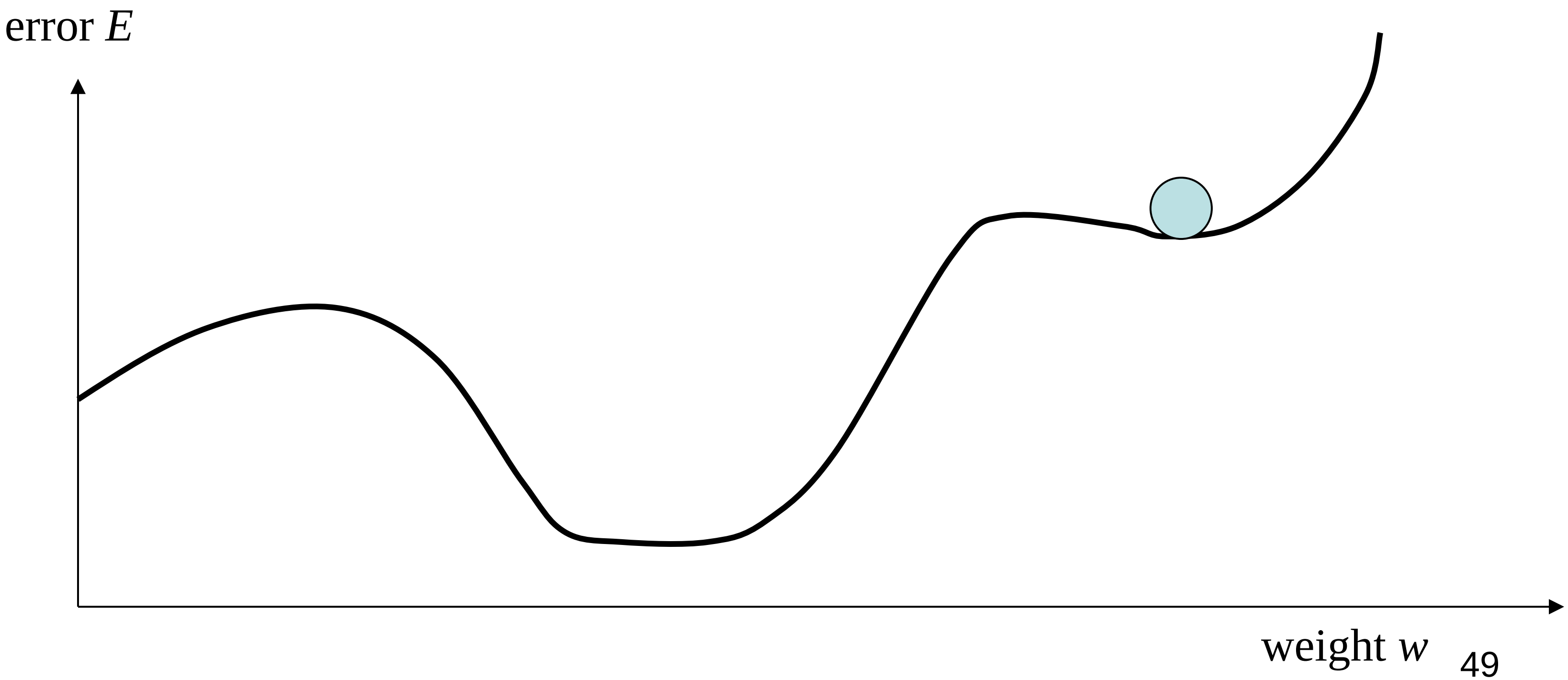
- Picture the weight-to-error graph as a surface with hills (high errors) and valleys (low errors).
- If the rate of gradient descent (i.e. **learning rate**) is **too fast** (i.e. weights are adjusted too quickly) then:
    - the net may miss a local minimum as it jumps into another valley with a higher minimum error.
    - the net may bounce around within a given valley, without settling down (converging) to a minimum. (See figure alongside). Some nets use an adjustment called the 'momentum' to promote uni-directional movement in weights and prevent high-frequency oscillation around a minimum.


Learning rate step size too big.
Weight bounces around but can t settle down to find minimum
Weight values that result in higher errors
Minimum error

Error surface for different neural network weights

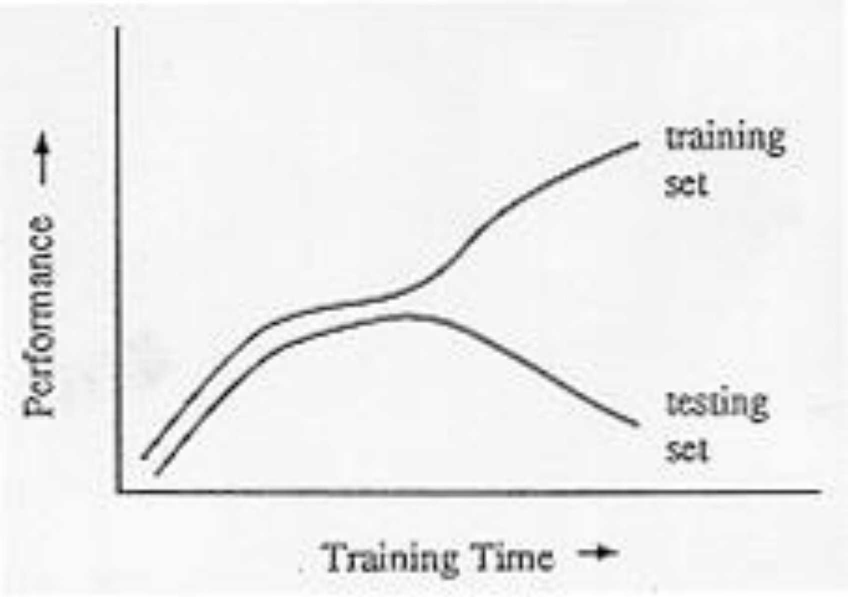
Problems
Rate of Gradient Descent too Slow

- If the rate of gradient descent is **too slow**:
  - the net may take a very long time to train
  - the net may get stuck at a local minimum as it is unable to jump out of a valley and find a global minimum (illustrated below).

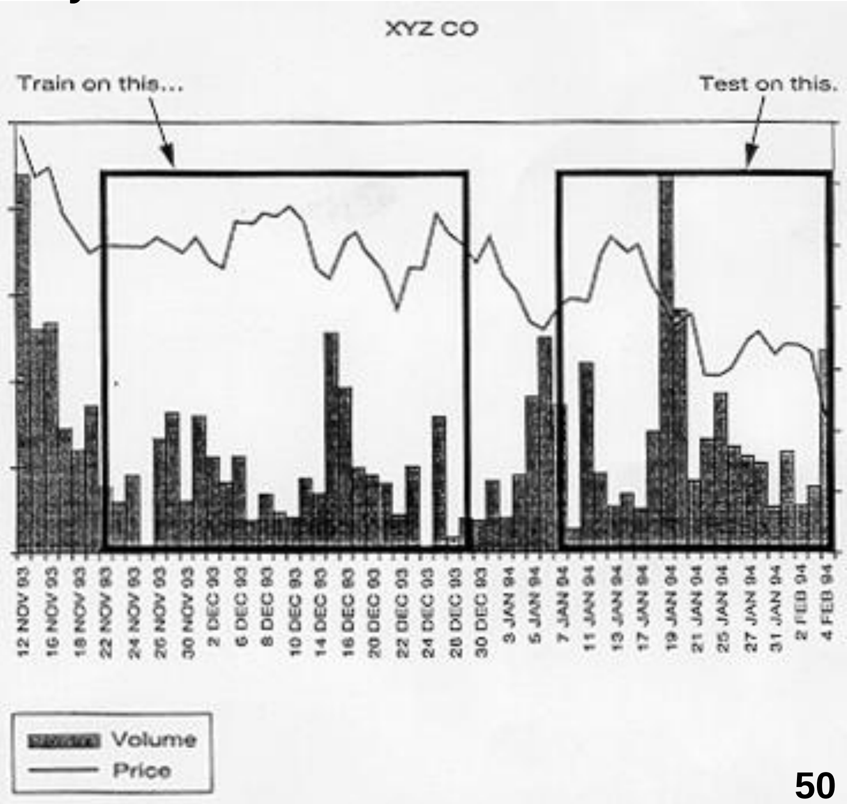


Problems
Overtraining

- If you let a neural net run for too long, it can memorize the training data, meaning it doesn't generalize well to new data.
- The usual resolution to this is to continually test the performance of the net against hold-out data (test data) while it is being trained. Training is stopped when the accuracy on the test set starts decreasing markedly.



Source: Dhar and Stein



Adjustable Parameters

You can adjust a number of parameters when producing neural nets:

- number of hidden layers (usually, two is sufficient)
- number of hidden nodes
- transfer functions for each node
- rate of learning (weight adjustment)
- mode of weight adjustment (e.g. gradient descent versus genetic algorithms)
- number of input nodes: e.g. by selecting a sub-set of attributes which is more predictive. Remember, there is one input node for each chosen attribute.

Steps in Solving Classification Problems

1. Determine the number of output nodes, input nodes which relate to number of classes and attributes of the problem domain respectively. Performed by a domain expert.
2. Determine NN configurations: the number of hidden layers, the number of hidden nodes, weights, functions. This steps is performed by a technical expert.
3. Train the network by propagating the training data through the network. Evaluate the output and adjust the weights as necessary.
4. Classify the queries using the trained network.

Design Issues in ANN

- Number of nodes in input layer
  - One input node per **binary/continuous** attribute
  - k or $\log_2$ k nodes for each **categorical** attribute with k values
- Number of nodes in output layer
  - One output for binary class problem
  - k or $\log_2$ k nodes for k-class problem
- Number of hidden layers and nodes per layer
- Initial weights and biases
- Learning rate, max. number of epochs, mini-batch size for mini-batch SGD, …

NN Issues

- Number of source nodes (attributes): Which attributes should be used in classifier?
- Number of hidden layers and hidden nodes:
  - For simple problems, 1 (or 0) hidden layer with few nodes is fine.
  - For more complicated problems, more nodes and/or layers may be needed.
  - Depend on activation functions, problem domain, training algorithm.
- Training data: Similar to decision tree, too few → inaccurate classifier, too many → overfitting.
- Number of output nodes (sinks): Usually, the number of classes, but can be fewer. E.g. two classes with one output node, the class assignment is based on the sign of the output.

NN Issues (cont.)

- Interconnections: In the simplest case, connect to all nodes in the next level.
- Weights: Assign small random number as initial weights.
- Activation functions: **Sigmoidals** are commonly used because they are smooth functions (required by backpropagation algorithm)
- Learning technique: Most common approach is backpropagation.
- Stopping criteria: The learning may stop when all the training tuples have propagated through the network or when it reaches the time limit or a specific error rate.

NN: Techniques for Backpropagation

- Activation function: There are many desirable properties for an activation function.
  - It must be a continuous function, that is, its derivative is defined throughout the domain of the function. This is a requirement for the backpropagation algorithm.
  - Its derivative should be continuous, that is, the function is smooth. Backpropagation can work with piecewise linear activation function (ramp function), but the process is more complicated.
  - It must be a nonlinear function. Otherwise, the expression power of a 3 layer network is no more than that of a 2 layer network.
  - It should saturate, that is, have some upper and lower bounds. This property is particularly desirable if the output of the network represent a probability value.
  - It should be monotonic, the sign of the derivative is the same throughout the domain of the function.

NN: Techniques for Backpropagation (cont.)

- Sigmoidal function has all of the desirable properties.
- Parameters for sigmoidal: Sigmoidal functions in the form of hyperbolic tangent is good.
  - Function centered on zero
  - Has antisymmetric property $f(-net) = -f(net)$
  - The learning rate is faster.
- Number of hidden units indicates the expressive power of the network.
  - Too many hidden units → the complicated decision boundaries, small error rate on training set, but may lead to overfitting.
  - Too few hidden units → the number of parameters for defining decision boundary is too restricted, high error rate on training set.

NN: Techniques for Backpropagation (cont.)

- Initial weights: Goal is to have uniform learning rate for all classes.
  - Must not be 0
  - Chosen from a uniform distribution $-w' < w < +w'$, should be small enough not to saturate the activation function and should be large enough to be outside the (almost) linear section of activation function (the center part of sigmoidal function).
- Learning rate, η : Again, it should be big enough so that the learning process makes some progress, but small enough so that it does not skip over the solution.
  - The optimal rate is the one that reaches the local minimum in one step:

$$\eta_{opt} = \left(\frac{\partial^2 E}{\partial w^2} \right)^{-1}$$

NN: Techniques for Backpropagation (cont.)

- Momentum: Keep moving in the previous direction. Helpful when the learning process is at a plateau. Let α be momentum fraction. The formula for updating the weight set is:

$$\mathbf{w}(m+1) = \mathbf{w}(m) + (1-\alpha)\Delta\mathbf{w}_{bp}(m) + \alpha\Delta\mathbf{w}(m-1)$$

- Number of hidden layers: Even 3 layer NNs are sufficient to implement arbitrary function, some application may require more layers in order to have a faster learning rate.
- Stopped training: When to stop the training. Like most other methods, the network with excessive training becomes very specific to the training set and poorly handles general data.

NNs

Advantages of NNs

- More robust, provide higher classification power.
- Better at handling noisy data.
- The NNs can be improved even after the training phase.
- Can be parallelized.

Disadvantages of NNs

- Hard to understand the structure of NN.
- Hard to configure the network.
- Not straightforward to generate rules from NNs.
- Attributes have to be numeric.
- In training phase, it may fail to converge .
- Slow.

Characteristics of ANN

- Multilayer ANN are universal approximators but could suffer from overfitting if the network is too large
  - Naturally represents a hierarchy of features at multiple levels of abstractions
- Gradient descent may converge to local minimum
- Model building is compute intensive, but testing is fast
- Can handle redundant and irrelevant attributes because weights are automatically learnt for all attributes
- Sensitive to noise in training data
  - This issue can be addressed by incorporating model complexity in the loss function
- Difficult to handle missing attributes

Deep Learning Trends

- Training **deep** neural networks (more than 5-10 layers) could only be possible in recent times with:
  - Faster computing resources (GPU)
  - Larger labeled training sets
- Algorithmic Improvements in Deep Learning
  - Responsive activation functions (e.g., RELU)
  - Regularization (e.g., Dropout)
  - Supervised pre-training
  - Unsupervised pre-training (auto-encoders)
- Specialized ANN Architectures:
  - Convolutional Neural Networks (for image data)
  - Recurrent Neural Networks (for sequence data)
  - Residual Networks (with skip connections)
- Generative Models: Generative Adversarial Networks