Classification

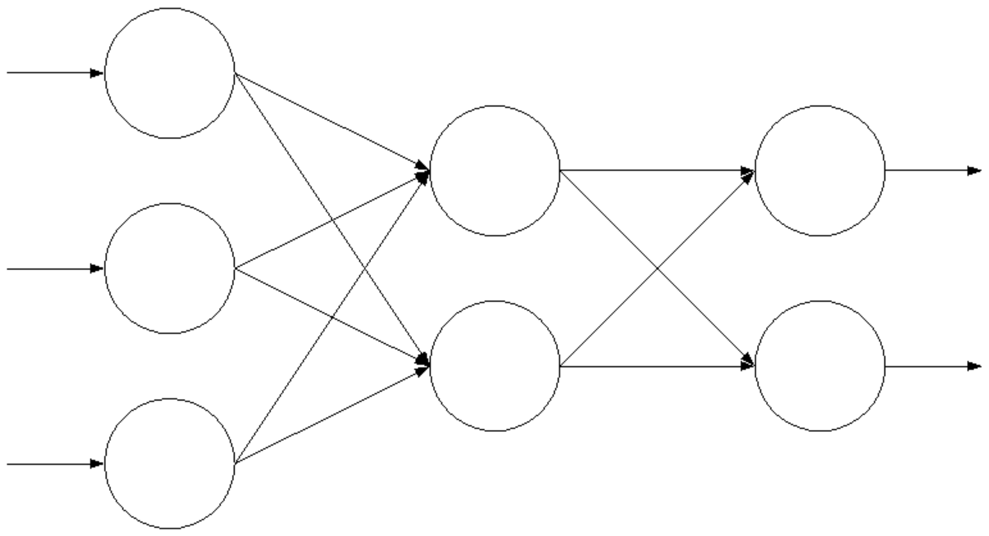
Neural Network-Based Algorithms

Introduction

- Using neural networks, a model representing how to classify objects is built, then used
- Neural Networks (NN) aka Artificial Neural Networks(ANN)
- ANN are based on observed functioning of human brain
  - Brain is estimated to have 10^{11} neurons
  - Each neuron is connected to about 10^{4} other neurons
- An ANN is composed of artificial neurons that are interconnected

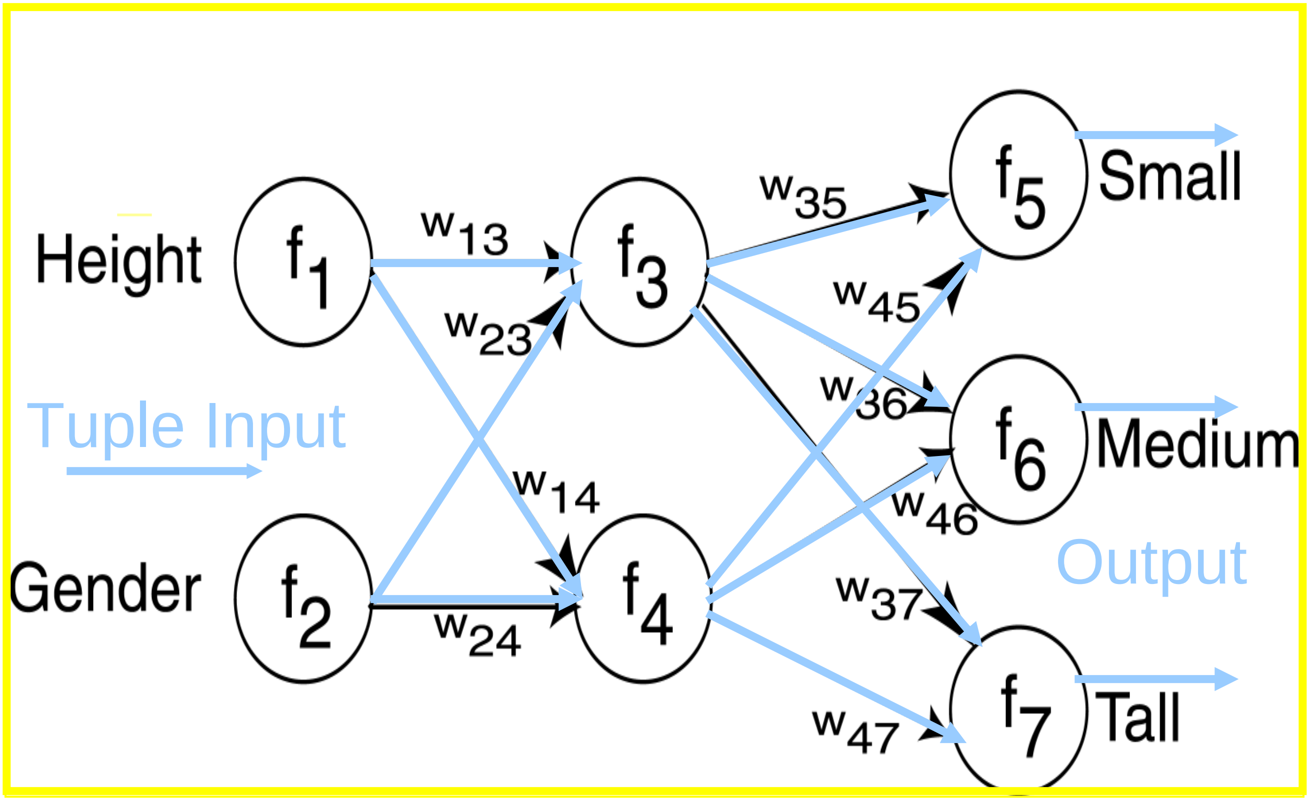
NN as a Graph

- We can view a neural network (NN) as a directed graph

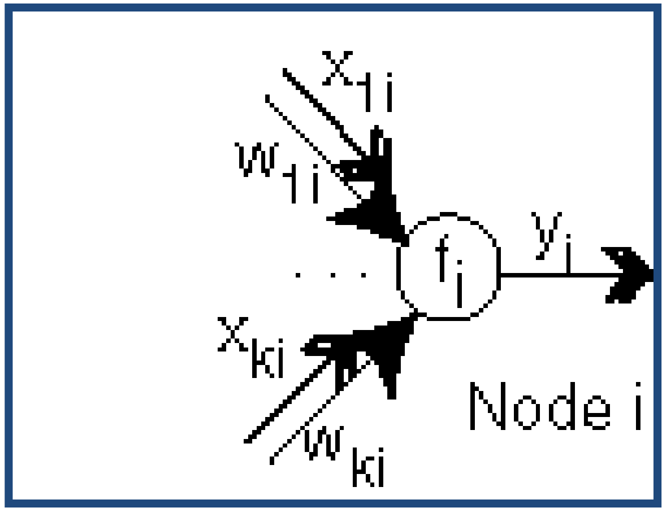


A Node/Neuron

- A neuron can receive inputs and weights on the in-edges
- It applies a processing function to weights and edges to produce an output
- Output can be sent to other neurons or as an output



Node/Neuron

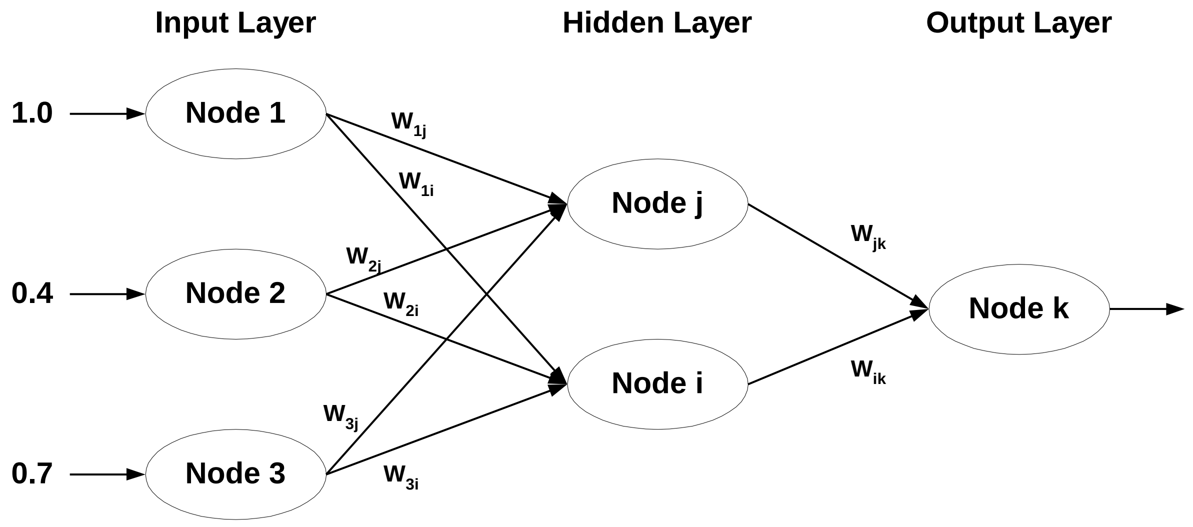


$$y_i = f_i\left(\sum_{j=1}^{k} w_{ji}\, x_{ji}\right) = f_i\left([w_{1i} \ldots w_{ki}] \begin{bmatrix} x_{1i} \\ \ldots \\ x_{ki} \end{bmatrix}\right)$$

- Notice: function is applied to $\vec{\mathbf{w}} \,.\, \vec{\mathbf{x}}$

Layers in ANN

- An ANN contains neurons in many layers:
  - Input/source layer
  - Hidden/internal layer(s)
  - Output/sink layer

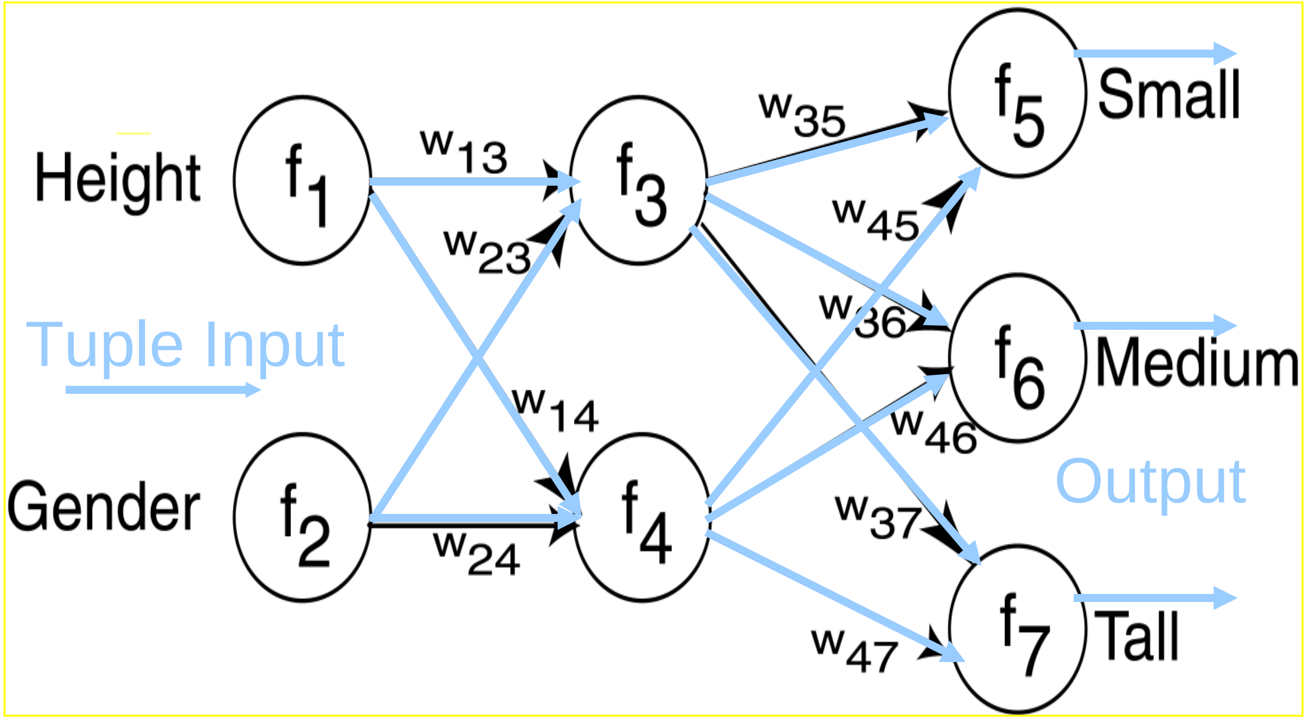


Types of Neural Networks

- Edges connect the nodes in the different layers
- In a **feedforward** network connections are from a node in one layer to every node in the next layer
- In a **feedback** network some connections go back to nodes in previous layers

How to Classify an Object?

- Object is input through the input nodes
  - In general, there is one input node for every attribute
- An output is produced on the output nodes
  - Usually, one output for every classification class
  - One node is sufficient if only two classes
  - Output values are estimates of object belonging to that class



Applications

- Character recognition
- Face recognition
- Speech recognition
- Computer vision
- Driving a car
- Classification

Characteristics of NN

- They can solve nonlinear problems
- High ability to learn and generalize to new situations
- High tolerance to the existence of noise
- Training time can be slow
- Hard to interpret
- Provide accuracy comparable to techniques such as DT
- Usually work only with numerical data
  - How about categorical attributes?
- Need to normalize input values

Example – Deciding NN Architecture

- Using the university students data set, we want to build a NN to classify university students as short, medium, or tall according to height and gender. Decide on the basic structure of a NN graph to start with.

Name	Gender	Height	Output1	Output2
Kristina	F	1.6m	Short	Medium
Jim	M	2m	Tall	Medium
Maggie	F	1.9m	Medium	Tall
Martha	F	1.88m	Medium	Tall
Stephanie	F	1.7m	Short	Medium
Bob	M	1.85m	Medium	Medium
Kathy	F	1.6m	Short	Medium
Dave	M	1.7m	Short	Medium
Worth	M	2.2m	Tall	Tall
Steven	M	2.1m	Tall	Tall
Debbie	F	1.8m	Medium	Medium
Todd	M	1.95m	Medium	Medium
Kim	F	1.9m	Medium	Tall
Amy	F	1.8m	Medium	Medium
Wynette	F	1.75m	Medium	Medium

Source: Dunham, Data Mining – Introductory and Advanced Topics, Table 4.1, page 78

Example (cont)

- First decide on the input nodes
  - Two input attributes, height and gender. Therefore, two input nodes
- Second, decide on the output nodes
  - Three classification classes: short, medium, and tall. Therefore, three output nodes
- Third, decide on the number of hidden layers and number of hidden nodes in each hidden layer
  - Usually one or two hidden layers with few nodes in each layer are enough
  - We will use one hidden layer with two nodes

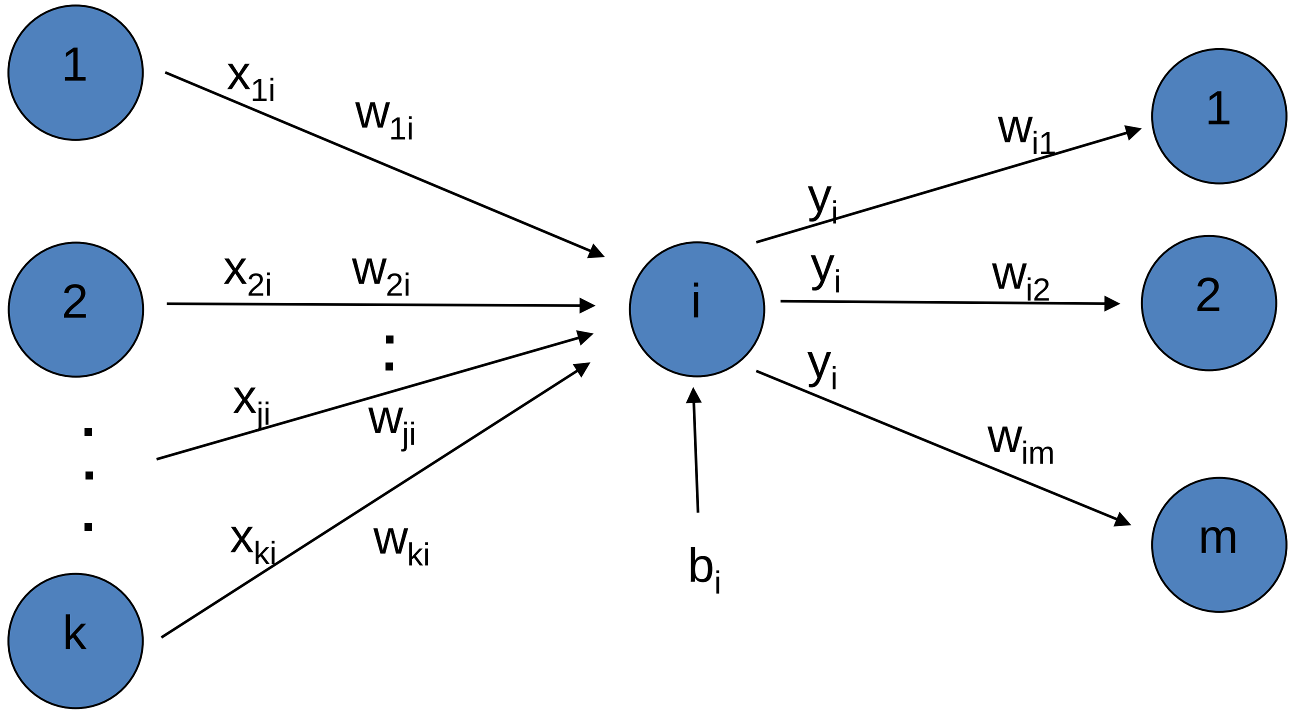
Example (cont)

- Fourth, decide on the weights on the edges
  - Starting weights are usually chosen as small random numbers in the range [0,1], or [-1, 1]
  - Weights will be updated and determined by training
- Finally, decide on the kind of activation function to apply at each function
  - At input nodes, the identity function is used
  - Functions at hidden and output nodes can perform complicated tasks and different functions can be used at different nodes

Example (cont)

- Activation functions are applied to the dot product of the input values and the weights on the nodes to produce the output

$$x_{1i}w_{1i} + x_{2i}w_{2i} + \ldots + x_{ki}w_{ki} = \vec{\mathbf{x}} \cdot \vec{\mathbf{w}}$$

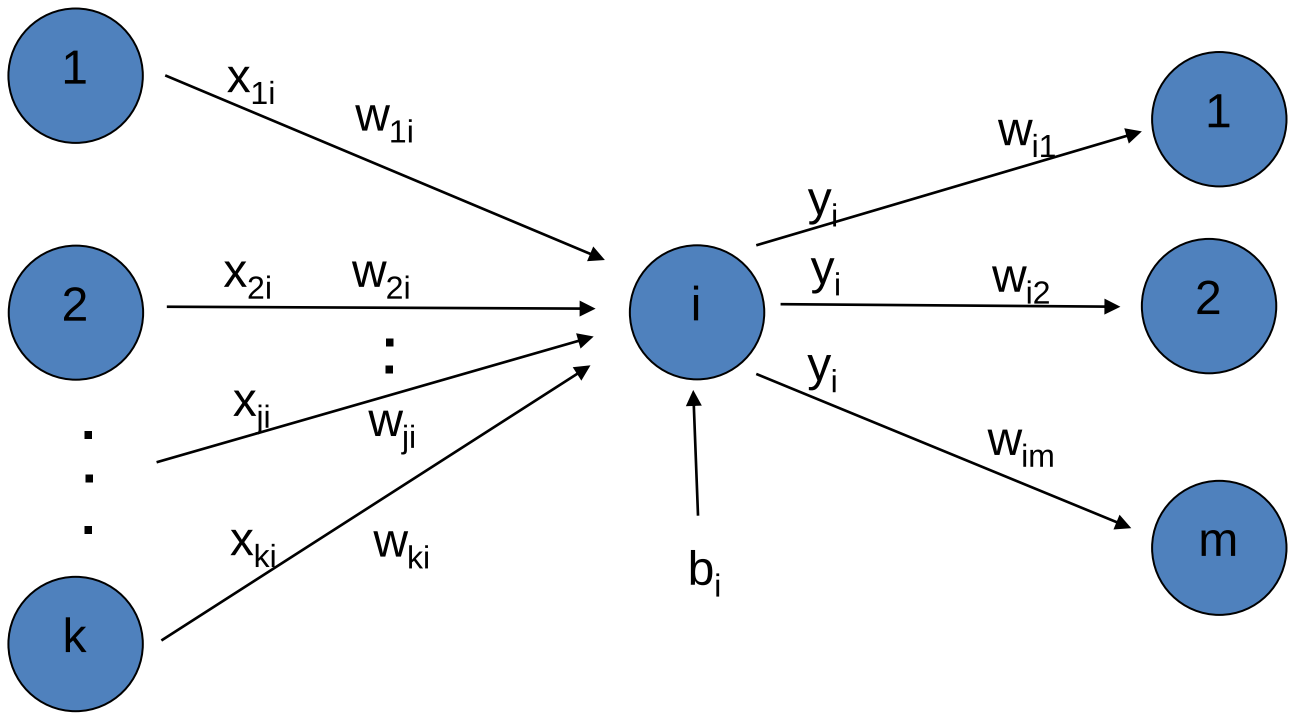


Example (cont)

$x_{1i}w_{1i} + x_{2i}w_{2i} + \ldots + x_{ki}w_{ki} = \vec{\mathbf{x}} \, . \, \vec{\mathbf{w}}$

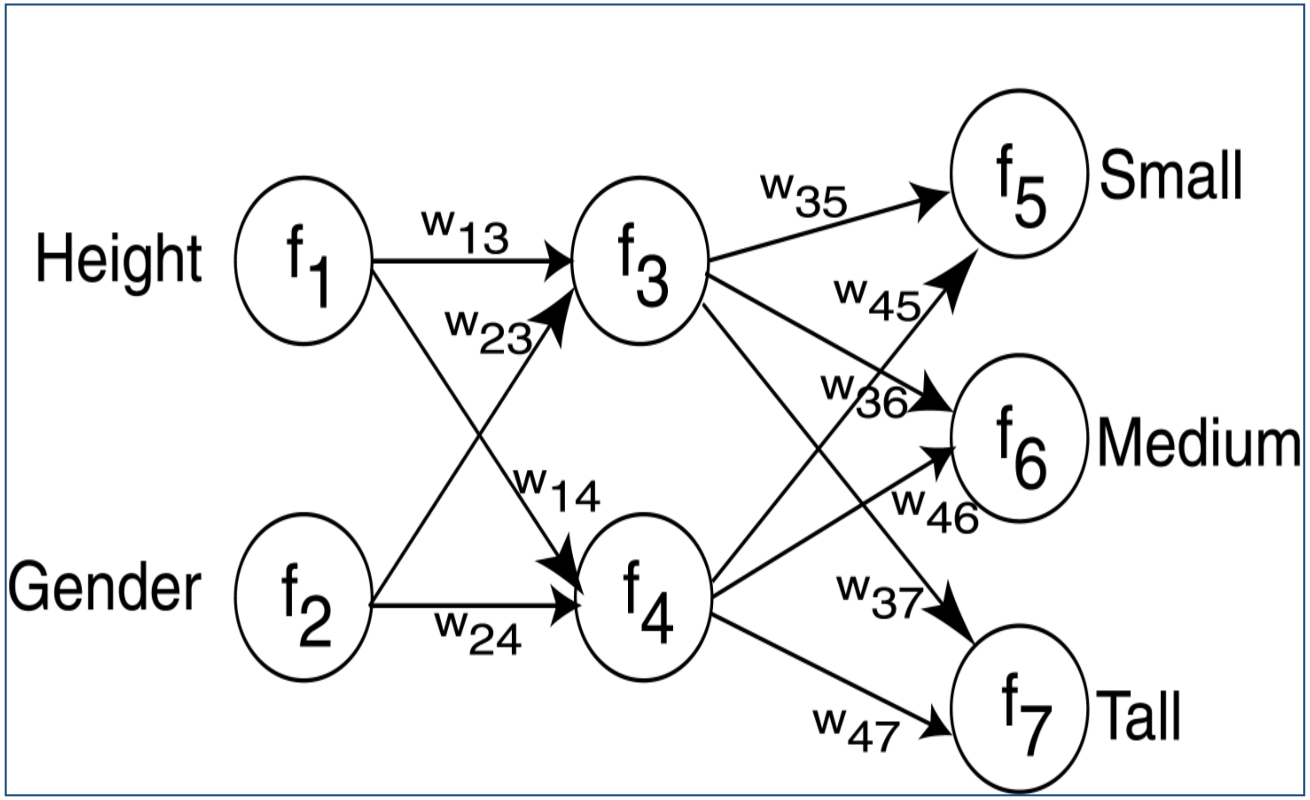
- A bias value b_i is usually added to indicate that the node fires when $\vec{\mathbf{x}} \, . \, \vec{\mathbf{w}} > b_i$
- Assume $b_i = w_{0i}$ and $x_{0i} = 1$. Therefore,

$\vec{\mathbf{x}} \, . \, \vec{\mathbf{w}} = x_{0i}w_{0i} + x_{1i}w_{1i} + x_{2i}w_{2i} + \ldots + x_{ki}w_{ki} = \sum_{j=0}^{k} x_{ji}w_{ji}$



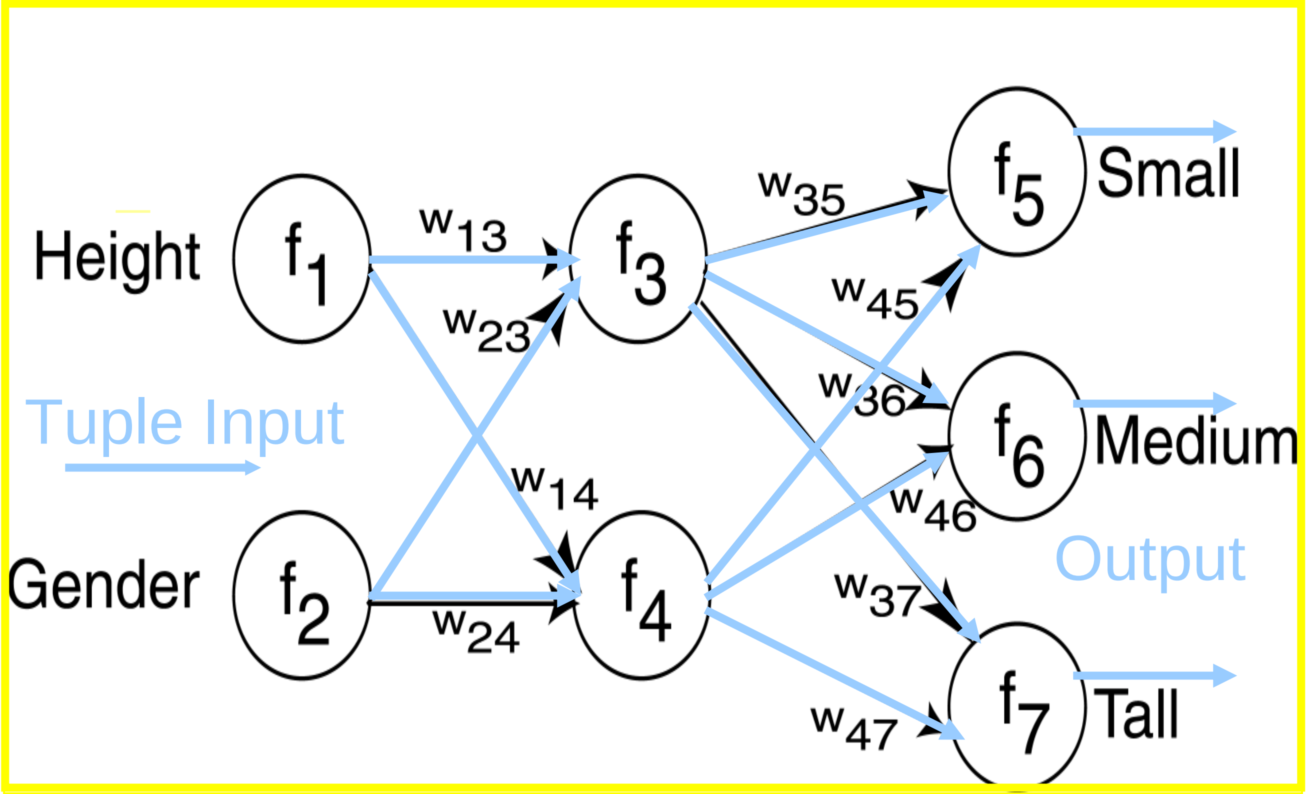
Example (cont)

- The NN basic structure to classify university students is as follows (assuming $b_i = 0$ for all nodes)



Activation Functions

- $y3 = f3(x_{13}w_{13} + x_{23}w_{23} + x_{03}w_{03}) = \sum_{j=0}^{3} x_{j3}w_{j3}$



Dot Product Example

- Let $\vec{\mathbf{x}}$ = (2, 3, 4) and $\vec{\mathbf{w}}$ = (0.5, 0.1, 0.25)
- $\vec{\mathbf{x}}$. $\vec{\mathbf{w}}$ = (2, 3, 4) . (0.5, 0.1, 0.25)

  = (2)(0.5) + (3)(0.1) + (4)(0.25)

  = 1 + 0.3 + 1

  = 2.3

Activation Functions

- Aka squashing functions, firing rules, processing element functions, transfer functions
- Activation functions are applied to $\vec{\mathbf{x}} \,.\, \vec{\mathbf{w}}$, where $\vec{\mathbf{x}}$ = (x_{0i}, x_{1i}, x_{2i}, . . . , x_{ki}) and $\vec{\mathbf{w}}$ = (w_{0i}, w_{1i}, w_{2i}, . . . , w_{ki})
- Let S = $\vec{\mathbf{x}} \,.\, \vec{\mathbf{w}}$
- Activation functions are called unipolar if f(S) [0,1] and are called bipolar if f(S) [-1, 1]
- Notation: f_i(S) means the activation function at node i applied to S = $\vec{\mathbf{x}} \,.\, \vec{\mathbf{w}}$

Examples of Activation Functions

Linear:

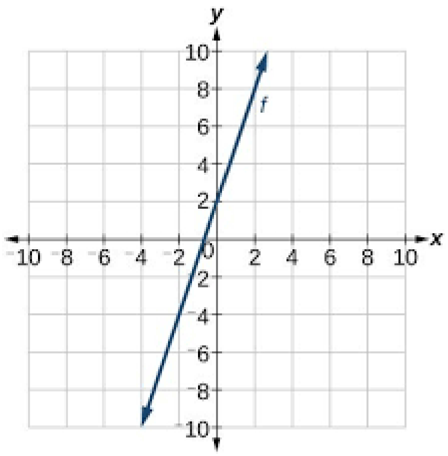
$$f_i(S) = c\ S$$

Threshold or Step:

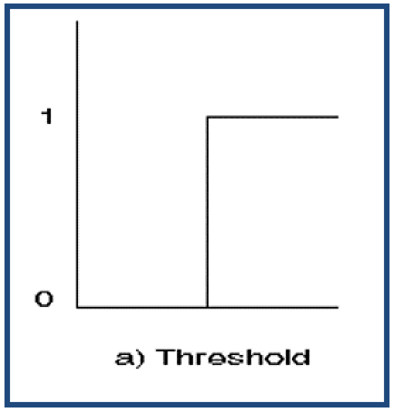
$$f_i(S) = \left\{ \begin{array}{ll} 1 & if S > T \\ 0 & otherwise \end{array} \right\}$$

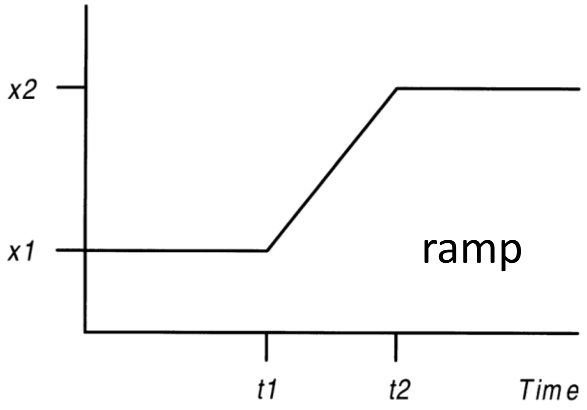
Ramp:

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



linear





ramp

Examples of Activation Functions

Sigmoid:

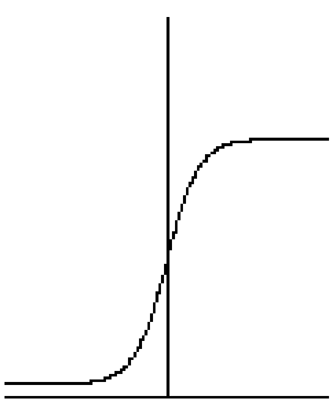
$$f_i(S) = \frac{1}{(1 + e^{-c\,S})}$$

Hyperbolic Tangent:

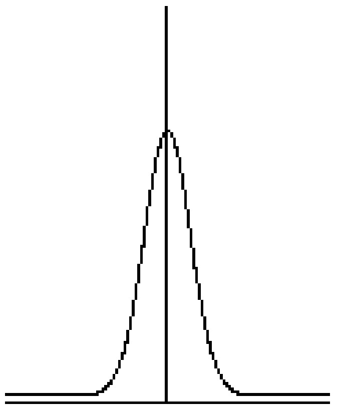
$$f_i(S) = \frac{(1 - e^{-S})}{(1 + e^{-c\,S})}$$

Gaussian:

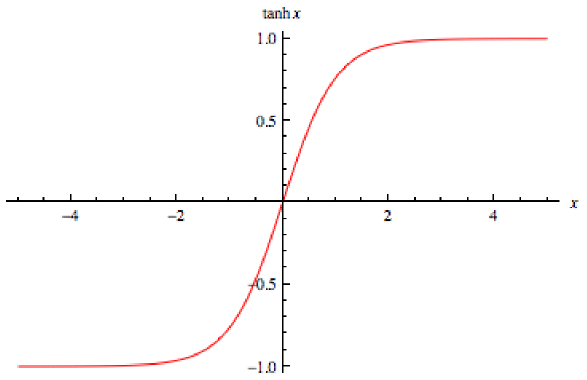
$$f_i(S) = e^{\frac{-S^2}{v}}$$



b) Sigmoid



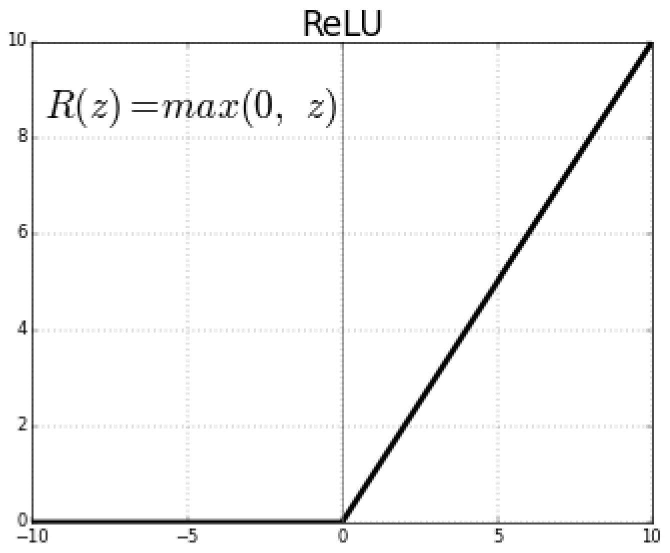
c Gaussian



tanh

Examples of Activation Functions

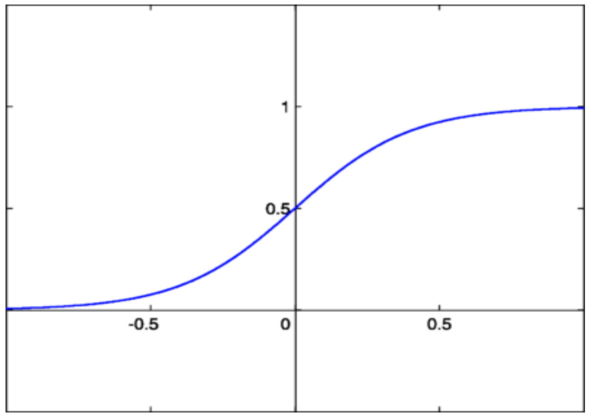
- Rectified Linear Unit (ReLU)
- $f_i(S) = \max(0, S) = \begin{cases} S \text{ if } S > 0 \\ 0 \text{ if } S \leq 0 \end{cases}$



Derivative of the Sigmoid Function

- Sigmoid function

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

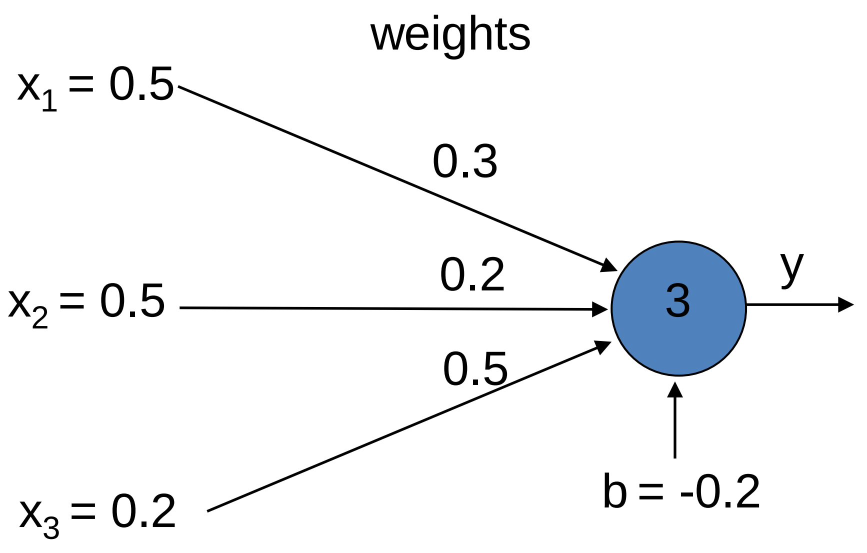


The derivative of the sigmoid function is f(1-f)

Example

- Consider the following neuron, which is part of an ANN. Inputs, weights, and a bias value are as shown on the diagram. Find the output value, y, for different activation functions.

S = $\vec{\mathbf{x}} \cdot \vec{\mathbf{w}}$ = (1, 0.5, 0.5, 0.2) .(-0.2, 0.3, 0.2, 0.5)
= -0.2 + 0.15 + 0.1 + 0.1 = 0.15



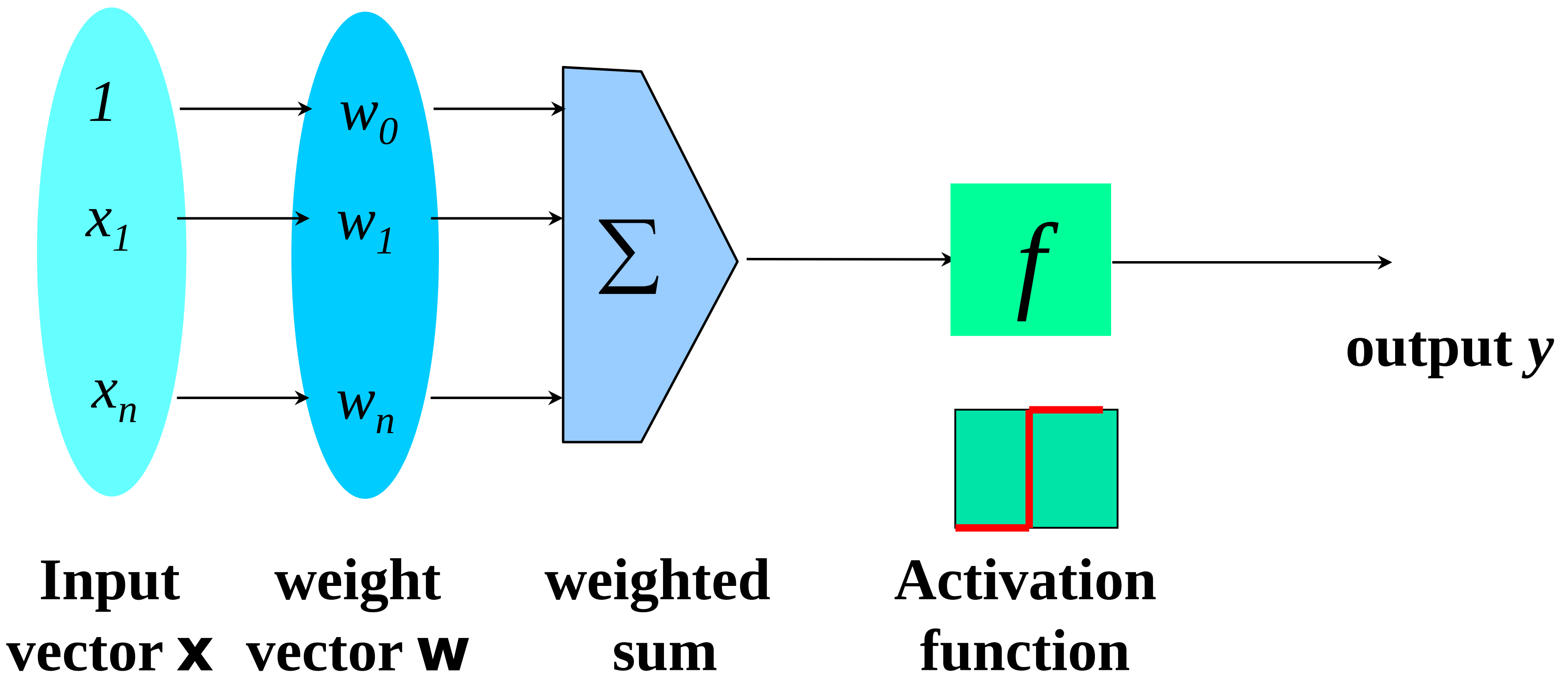
For threshold function:
Assume T is zero
y = f_i(S) = f_i(0.15) = 1

For sigmoid function:
$f(S) = \frac{1}{1+e^{-cS}}$, assume c = 1

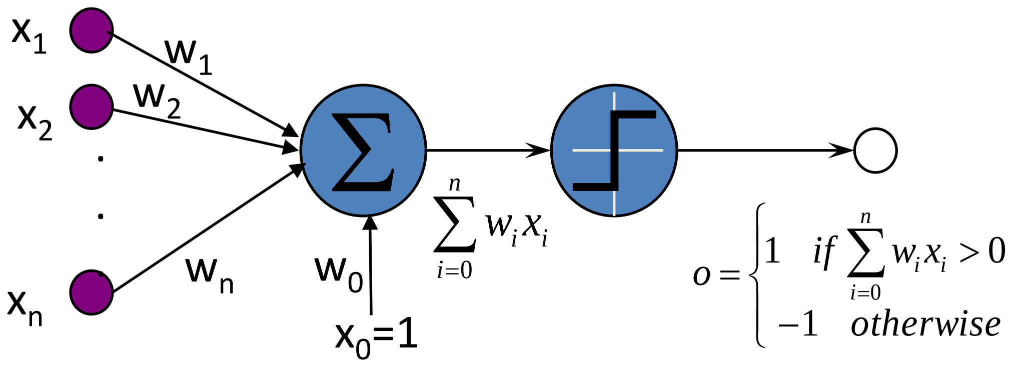
y = f_i(0.15) = $\frac{1}{1+e^{-0.15}}$ = 0.54

Perceptron

- A NN consisting of one neuron

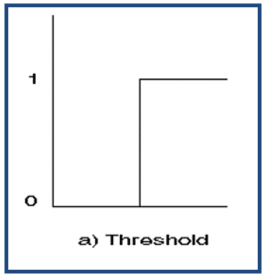


Perceptron (cont.)



x_1
x_2
x_n
w_1
w_2
w_n
w_0
x_0=1
Σ
$\sum_{i=0}^{n} w_i x_i$
$o = \begin{cases} 1 & \text{if } \sum_{i=0}^{n} w_i x_i > 0 \\ -1 & \text{otherwise} \end{cases}$

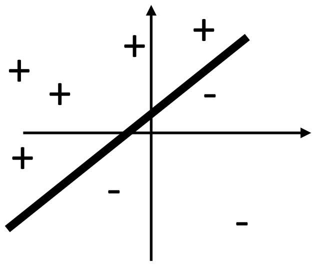
- Usually a step/threshold activation function is used, although other functions are possible

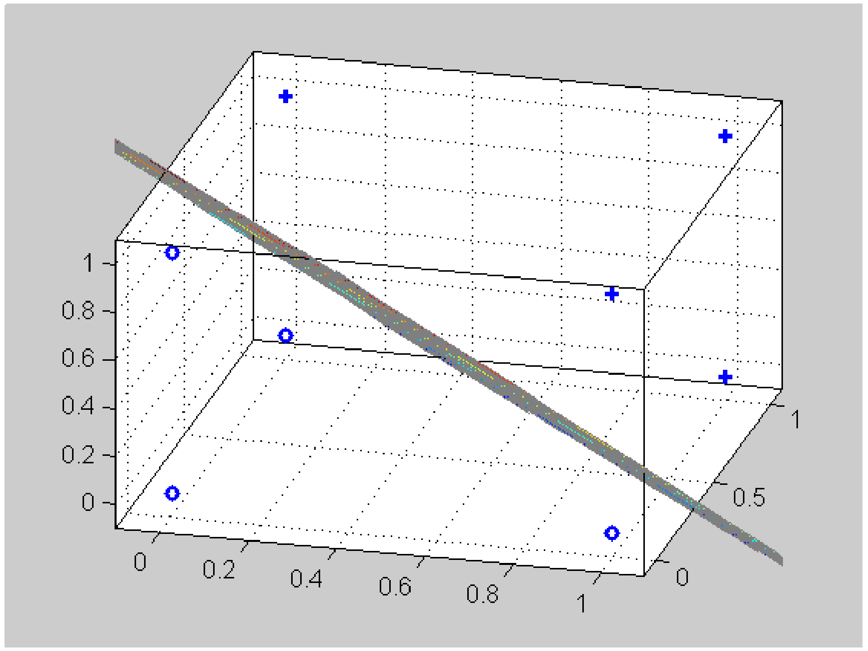


- $f(s) = \begin{cases} 1 & \text{if } S > 0 \\ -1 & \text{if } S \leq 0 \end{cases}$
- or, output$(x_1, x_2, \ldots, xk) \quad = \begin{cases} 1 & \text{if } w_0x_0 + w_1x_1 + \cdots + x_kw_k > 0 \\ -1 & \text{otherwise} \end{cases}$

Representation Power of a Perceptron

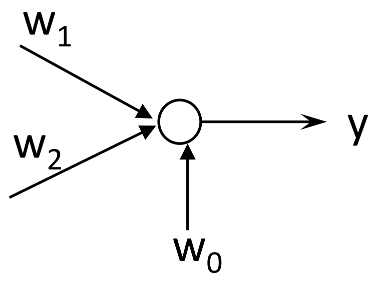
- A simple perceptron can be used to separate any two sets of numbers that are linearly separable
- Equation of the separating line (hyperplane) is $\vec{\mathbf{x}} \,.\, \vec{\mathbf{w}} = 0$





Representation Power of a Perceptron (cont)

- A single perceptron can represent many primitive Boolean functions
  - AND, OR, NAND, NOR



AND: $w_0=-0.8$, $w_1=w_2=0.5$

Assume 1 means true and -1 means false

Use the function $f(s) = \begin{cases} 1 & \text{if } S > 0 \\ -1 & \text{if } S \leq 0 \end{cases}$

S = $\vec{\mathbf{x}} \cdot \vec{\mathbf{w}} = (1, x_1, x_2).(w_0, w_1, w_2)$

$= w_0 + w_1x_1 + w_2x_2 = -0.8 + 0.5x_1 + 0.5x_2$

True ∧ True = 1 ∧ 1 = -0.8 + 0.5 + 0.5 = 0.2> 0

$f(0.2) = 1 =$ True

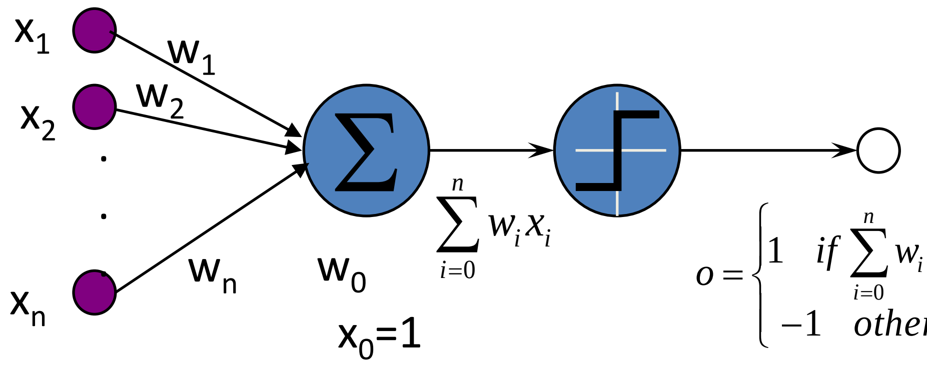
True ∧ False = 1 ∧ -1 = -0.8 + 0.5 - 0.5 = -0.8 < 0

$f(-0.8) =$ -1 = False

False ∧ False = -1 ∧ -1 = -0.8 - 0.5 - 0.5 = -1.8 < 0

$f(-1.8) =$ -1 = False

Perceptron Learning Rule



$$o = \begin{cases} 1 & \text{if } \sum_{i=0}^{n} w_i x_i > 0 \\ -1 & \text{otherwise} \end{cases}$$

$w^{(k+1)} \leftarrow w^{(k)} + \lambda[y_i - f(w^{(k)}, x_i)]x_i$; λ: learning rate; y_i is the actual target

$w^{(k+1)} \leftarrow w^{(k)} + \Delta w$

- Intuition:
  - Update weight based on error: $e = [y_i - f(w^{(k)}, x_i)]$
  - Assume x_i is positive
  - If y_i = f(x,w), e=0: no update needed
  - If y = 1 and f(x,w) = -1, e=2: weight must be increased so that $\vec{\mathbf{x}} \cdot \vec{\mathbf{w}}$ will increase; try λ = 0.1, x_i = 0.8; $\Delta w = 0.1 * 2 * 0.8 = 0.16$
  - If y = -1 and f(x,w) = 1, e=-2: weight must be decreased so that $\vec{\mathbf{x}} \cdot \vec{\mathbf{w}}$ will decrease; $\Delta w = 0.1 * (-2) * 0.8 = -0.16$

The Learning Rate

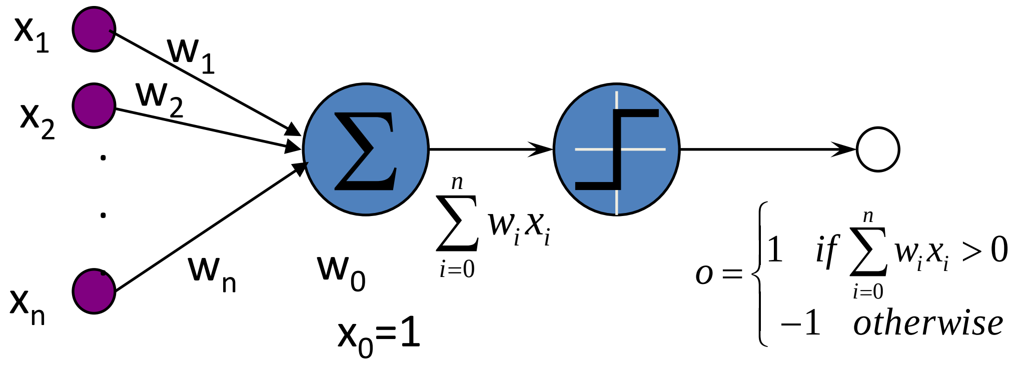
$w^{(k+1)} \leftarrow w^{(k)} + \lambda\left[y_i - f(w^{(k)}, x_i)\right]x_i$; λ: learning rate; y$_i$ is the actual target

$w^{(k+1)} \leftarrow w^{(k)} + \Delta w; where\ \Delta w = \lambda\left[y_i - f(w^{(k)}, x_i)\right]x_i$

$\Delta w = \lambda(yi - \hat{y})x_i$; y$_i$ is actual target/class, $\hat{y}$ is predicted class

- Symbol used is λ, μ, ℓ
- Controls how fast the network learns
- Usually between 0 and 1; 0.1 <λ< 0.4 is usually better
- λ is too large =>
  - Δw can be large
  - changes a lot
  - unstable perceptron; weights can oscillate back and forth
- λ is too small
  - Δw is small
  - slower training
  - perceptron is more stable and resistant to noise

The Perceptron Algorithm



x_1
x_2
x_n
w_1
w_2
w_n
w_0
$x_0=1$
$\sum_{i=0}^{n} w_i x_i$
$o = \begin{cases} 1 & \text{if } \sum_{i=0}^{n} w_i x_i > 0 \\ -1 & \text{otherwise} \end{cases}$

- Initialize the weights (w_0, w_1, …, w_n)
- Repeat
  - For each training example (**x**$_i$, y_i)
    - Compute f(**w**$_k$, **x**$_i$) //w_k means the weight at "current" iteration k
    - Update the weights //k is iteration number; k + 1 means next iteration

$$w^{(k+1)} = w^{(k)} + \lambda\left[y_i - f\left(w^{(k)}, x_i\right)\right]x_i$$ //**w** is a vector

- Until stopping condition is met

Stopping Condition

- Certain number of iterations
- Error rate is below threshold
  - percentage of misclassified objects < threshold, or
  - weight updates < threshold

Perceptron Algorithm

- Initialize weight w
- **Repeat**

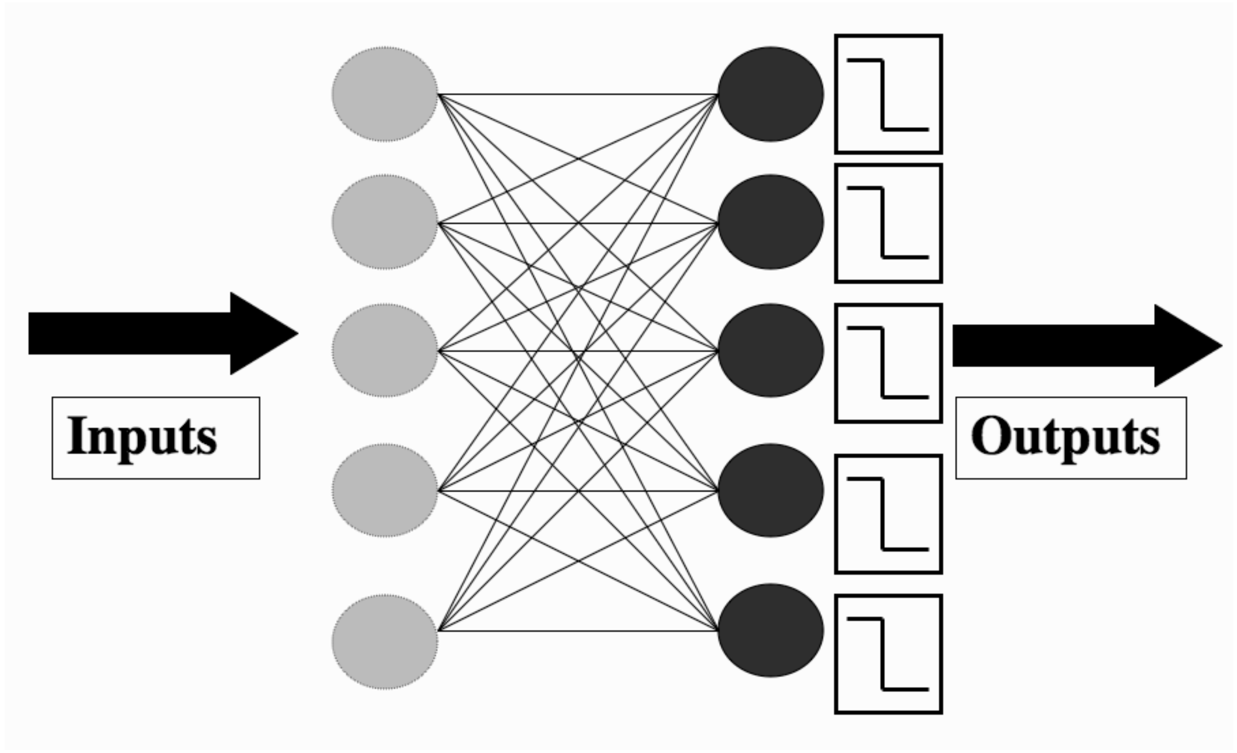
```
For each data point $(x^n, t^n)$    //$t^n$ is the actual target
Classify each data point using current w.
If $w^T x^n t^n > 0$ (correct), do nothing
If $w^T x^n t^n < 0$ (wrong), $w^{new} = w + \eta x^n t^n$
$w = w^{new}$
```

- **Until** w is not changed (all the data will be separated correctly, if data is linearly separable) or error is below a threshold.

Rosenblatt, 1962

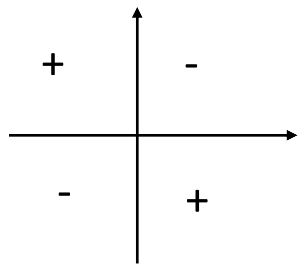
Another View of a Perceptron

- Another view of the perceptron is as a NN with one layer of neurons in addition to the inputs



Perceptron Limitations

- For nonlinearly separable problems, perceptron learning algorithm will fail because no linear hyperplane can separate the data perfectly
- Example: The XOR function



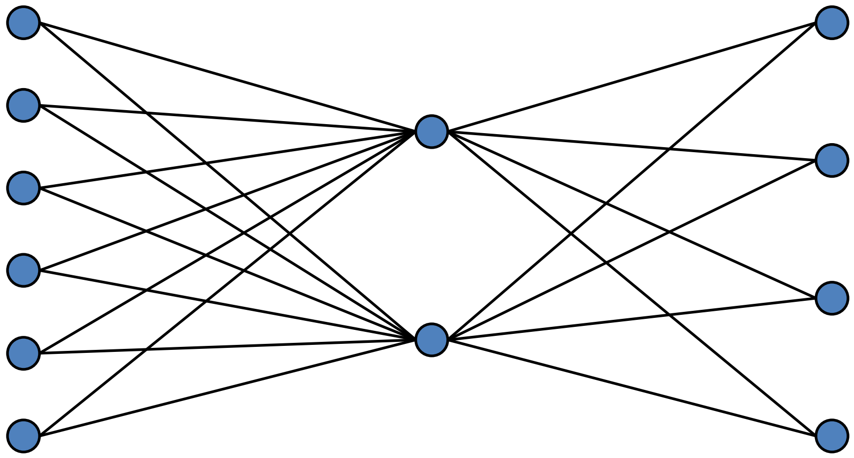
x1	x2	y
1	1	C1
1	-1	C2
-1	1	C2
-1	-1	C1

Multilayer Neural Network

- Aka multilayer perceptron (MLP)
- Hidden layers
  - intermediary layers between input & output layers
- More general activation functions (sigmoid, linear, etc)

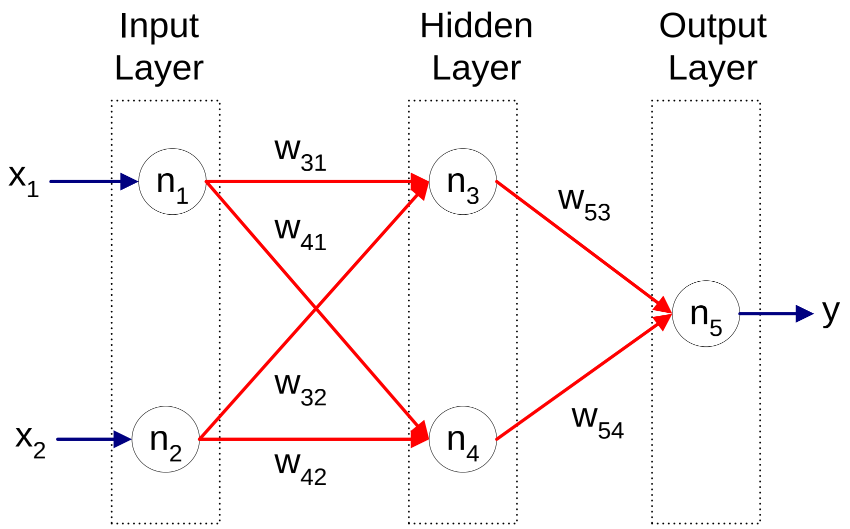
Multilayer Perceptron

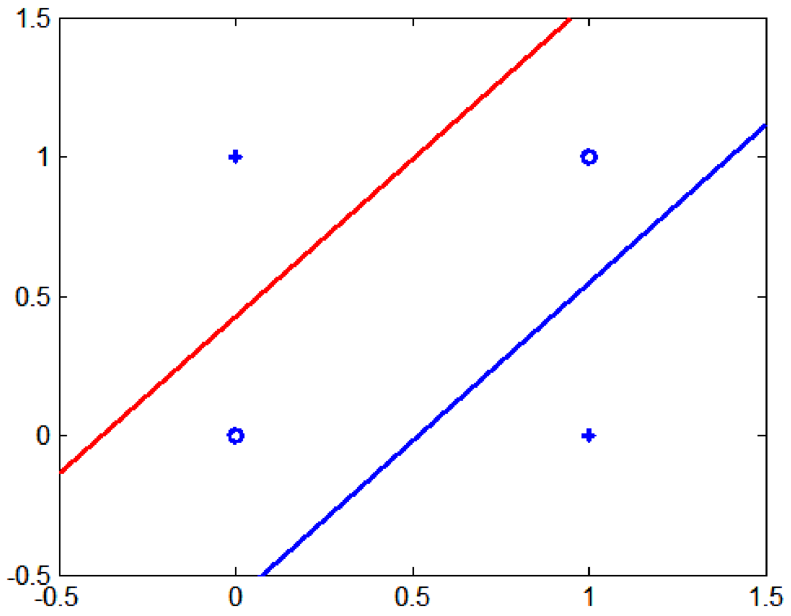
- Add more layers
- Each layer has one ore more neurons
- A unit connects to all units in neighbor layers
- Example: university students example we saw before



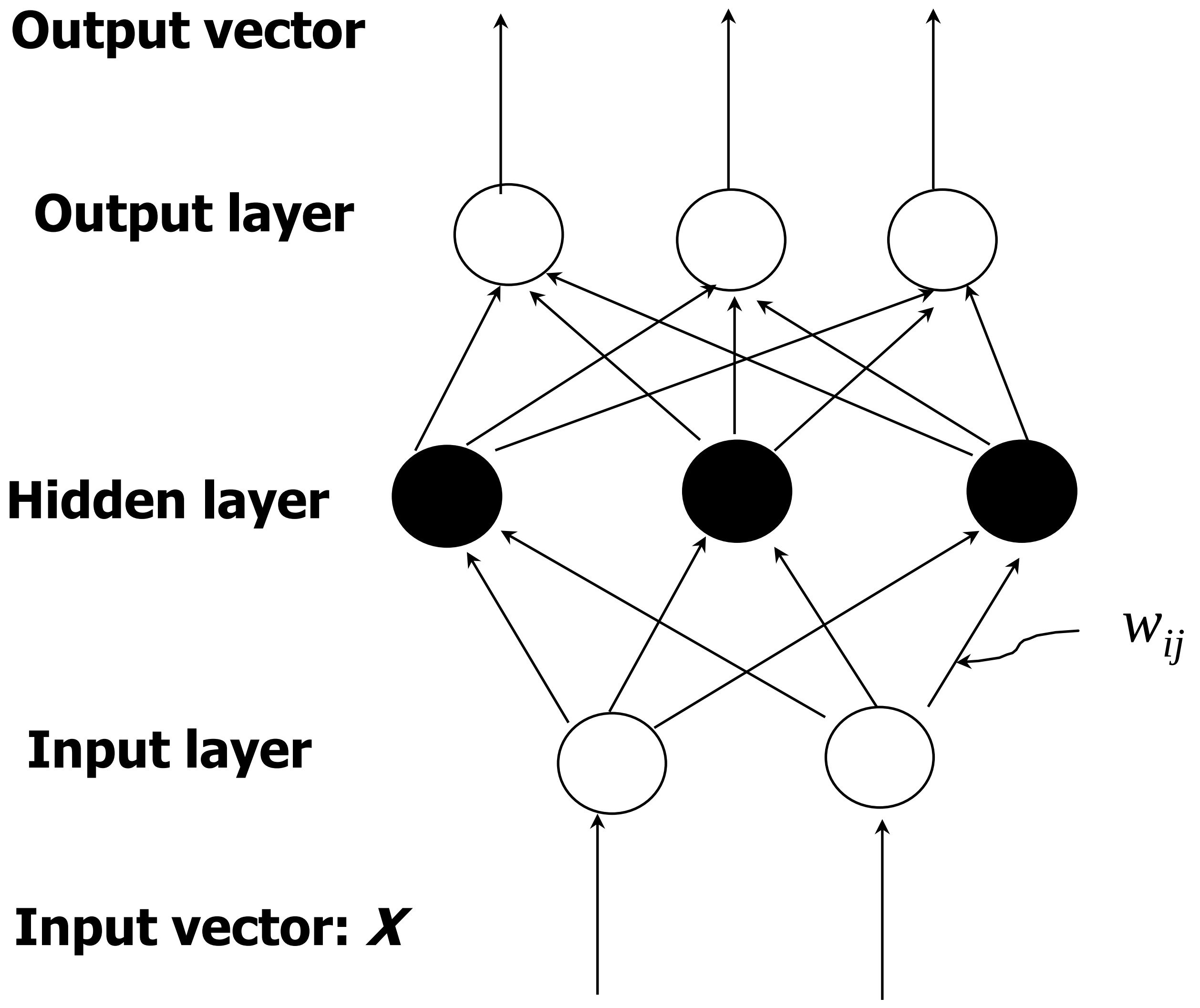
A Multilayer Perceptron to Solve the XOR Problem

- Multi-layer neural network can solve any type of classification task involving nonlinear decision surfaces

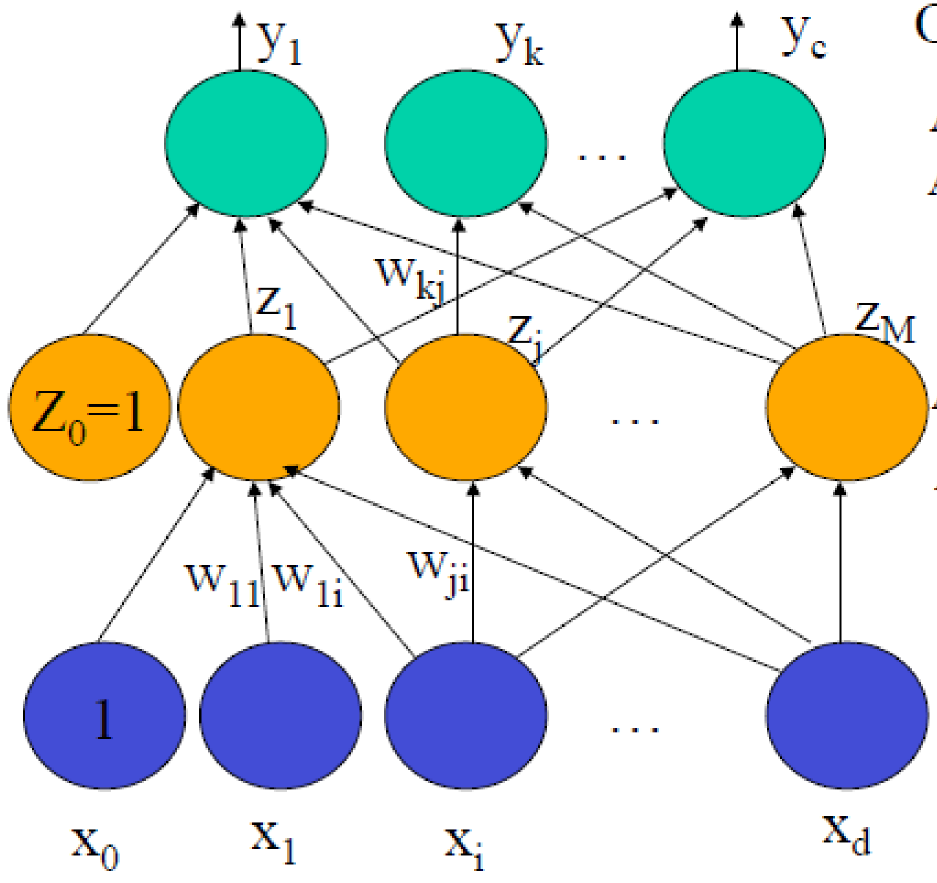




A Multi-Layer Feed-Forward Neural Network



Two-Layer Neural Network



Output

Activation function: *f* (linear,sigmoid, softmax)

Activation of unit a_k:

$$\sum_{j=0}^{M} w_{kj} z_j$$

Activation function: *g* (linear, tanh, sigmoid)

Activation of unit a_j:

$$\sum_{i=0}^{d} w_{ji} x_i$$

$$y_k = f\left(\sum_{j=0}^{M} w_{kj} \times g\left(\sum_{i=0}^{d} w_{ji} x_i\right)\right)$$

Matrix View of Propagation

w_{11}	**w_{12}**	**w_{13}**	**…**	**w_{1d}**
W_{21}	W_{22}	…	….	W_{2d}
….				
W_{m1}	W_{m2}			W_{md}

$* \quad (x_1, x_2, \ldots, x_d) \qquad = (a_1, a_2, \ldots, a_m)$

$\downarrow$ **Apply activation function**

w_{11}	**w_{12}**	**w_{13}**	**…**	**w_{1m}**
W_{21}	W_{22}	…	….	W_{2m}
….				
W_{c1}	W_{m2}			W_{cm}

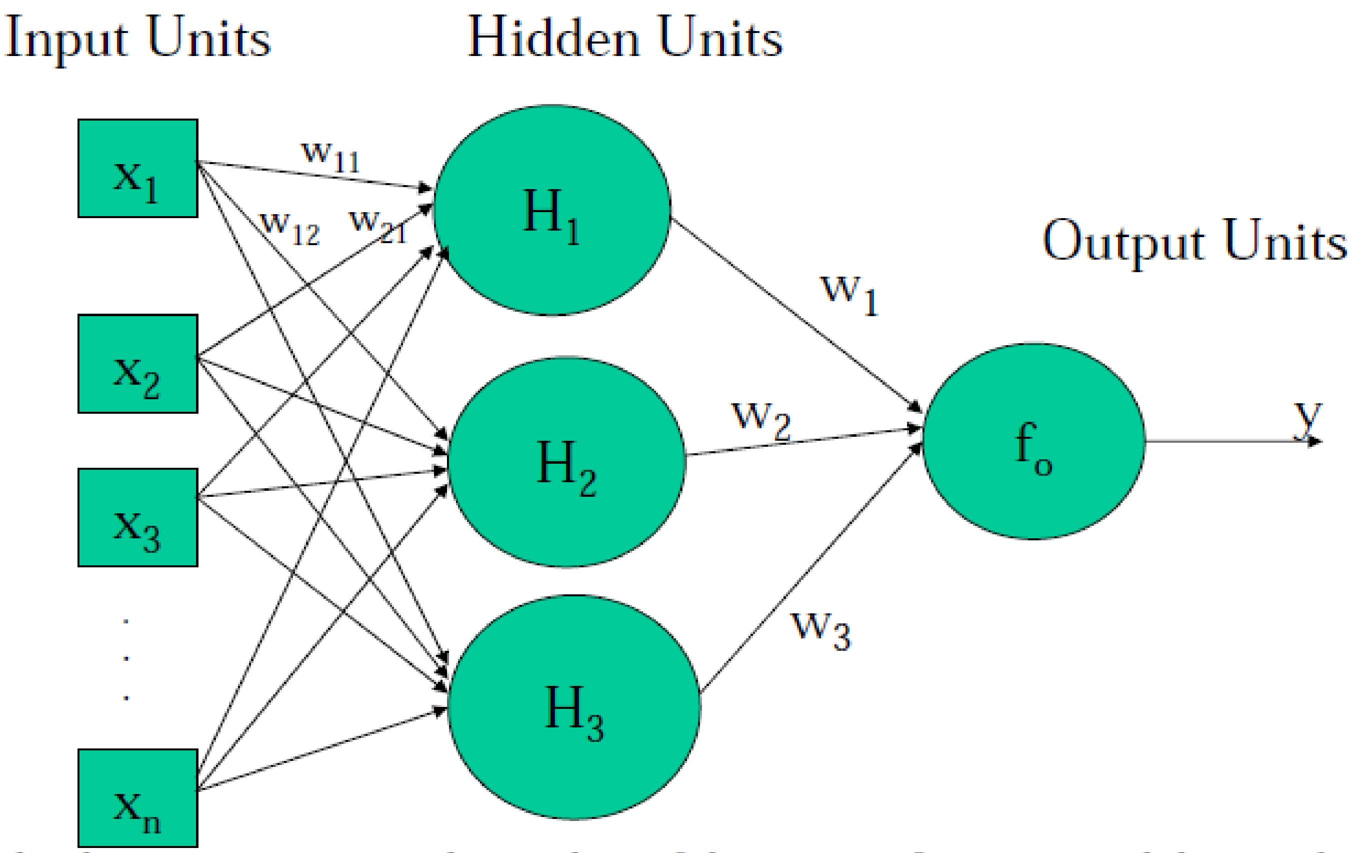
$* \quad (z_1, z_2, \ldots, z_m)$

$(b_1, b_2, \ldots, b_c) =$

Apply activation function

$(O_1, O_2, \ldots, O_c)$

A General Neural Network



Each weighted connection means the product of the output of one unit and the weight is sent to another unit as input. Each hidden unit and output unit have a transfer function to convert the sum of inputs into an output. Let transfer function of hidden unit be f_h (e.g., identity function) output unit to be f_o (e.g., sigmoid function, $1/(1+e^{-x})$).

Neural Network is a Universal Function Approximator

We can represent neural network as an function:

$$y = f_o(\sum_i w_i f_h(\sum_j x_j w_{ij}))$$

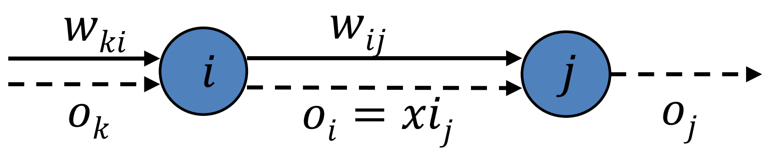
This function is universal, which means that any function y=*f*(x) can be approximated by this function accurately, given a set of appropriate weights W.

So, the key is to adjust weights W to make neural network to approximate the function of our interest. e.g., given input of sequence features, tell if it is a gene or not (1: yes, 0: no)?

Adjust Weights by Training

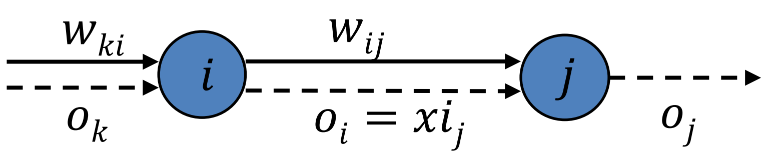
- How to adjust weights?
- Adjust weights using known examples (training data) $(x_1, x_2, x_3, \ldots x_n, y)$. This process is called training or learning
- Try to adjust weights so that the difference between the output of the neural network and y (called target) becomes smaller and smaller.
- Goal is to minimize Error (difference)

How are Weights Adjusted?



- Assume that the output from node j is o_j but should be y_j
- Remark: many times, the letter T_j (or t_j, or d_j) is used instead of o_j (or y_j)
- Error produced by node j is $|o_j - y_j|$
- The mean squared error (MSE) is given by $\frac{(O_j - y_j)^2}{2}$
- We want to minimize the MSE over all the nodes in the output layer
- Assuming m output nodes, then total MSE over output nodes is $\sum_{j=1}^{m} \frac{(O_j - y_j)^2}{2}$

Learning Rule in MLP



- Weight w_{ij} is updated as follows

$$w_{ij} \leftarrow wi_j + \ \Delta wij$$

- For a perceptrons, this was good, but for MLP we have nodes and edges in several layers
- Moreover, MSE is an indirect function of w_{ij}
- Training in a MLP uses a technique called gradient descent

Adjust Weights using Gradient Descent (back-propagation)

Known:

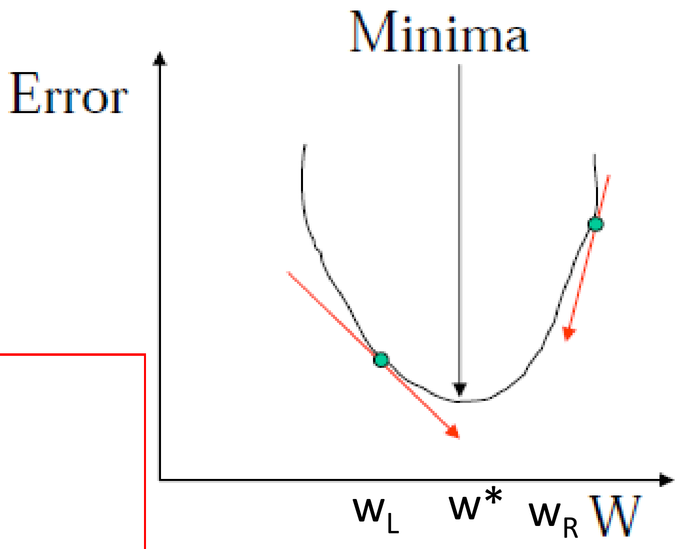
Data: $(x_1,x_2,x_3,\ldots,x_n)$ (y)

Unknown weights w:

$w_{11}, w_{12},\ldots\ldots$

```
Randomly initialize weights
Repeat
    for each example, compute output $o$
        calculate error $E = (o-y)^2$
        compute the derivative of $E$ over $w$: $dw=\frac{\partial E}{\partial w}$
        $w_{new} = w_{prev} - \eta * dw$
Until error doesn't decrease or max num of iterations
```

Note: η is learning rate or step size.



$$\Delta w_{ij} = -\mu \frac{\partial E}{\partial wij}$$

where E = MSE = $\sum_{j=1}^{m} \frac{(O_j - yj)^2}{2}$

The Backpropagation (BP) Algorithm

- BP is a popular NN algorithm
- It performs learning on a MLP

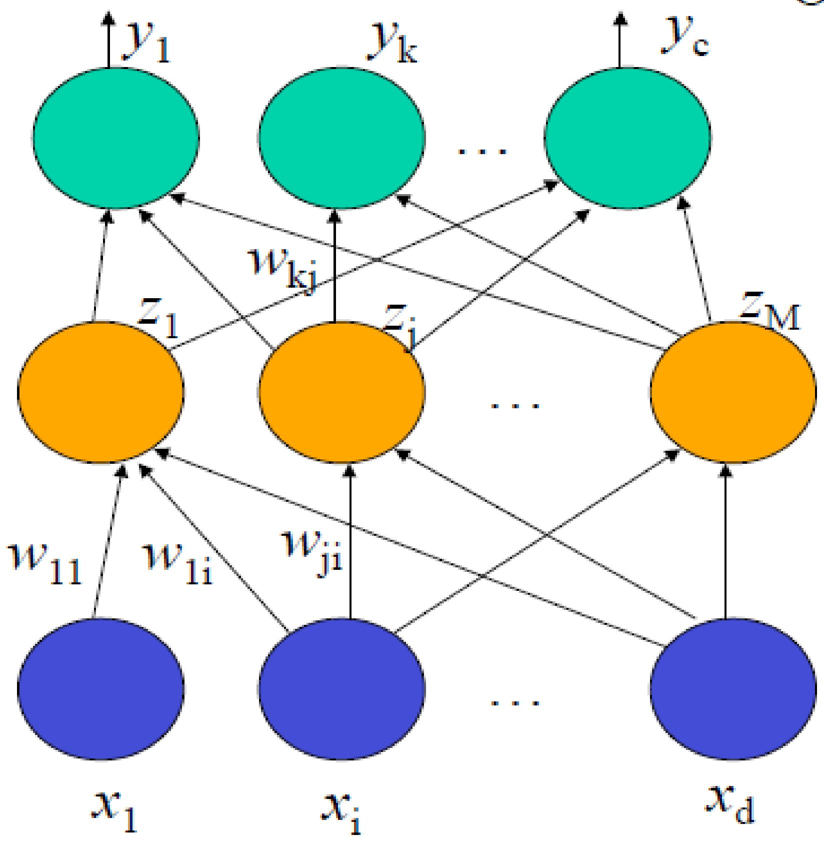
Backpropagation Algorithm

- Decide on the network topology
- Initialize all network weights to small random numbers (e.g., -0.5 to 0.5)
- Repeat
  - For each training sample
    - Propagate the input forward through the network
    - Propagate the errors backward through the network
    - Update each network weight
- Until the termination condition is met

Termination Conditions for Training a NN

- A fixed number of iterations has been reached
  - This could be hundreds of thousands of iterations
- The error on the training examples is below some threshold
  - percentage of misclassified samples is < threshold
  - $\Delta w_{ij}'s$ are < threshold

Forward Propagation



Output y_k

Activation function: f (linear,sigmoid, softmax)

Activation of unit a_k: $\sum_{j=1}^{M} w_{kj} z_j$

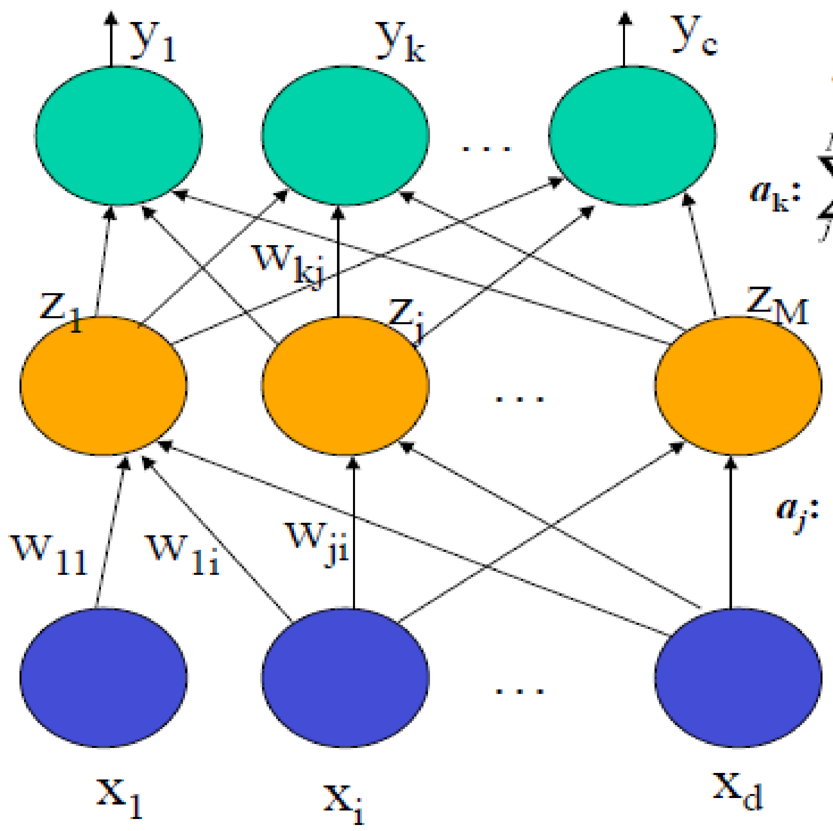
z_j

Activation function: g (linear, tanh, sigmoid)

Activation of unit a_j:

$$\sum_{i=1}^{d} w_{ji} x_i$$

Backward Propagation



f

a_k: $\sum_{j=1}^{M} w_{kj} z_j$

g

a_j: $\sum_{i=1}^{d} w_{ji} x_i$

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

$$\frac{\partial E}{\partial y_k} = y_k - t_k$$

$$\frac{\partial E}{\partial a_k} = \frac{\partial E}{\partial y_k}\frac{\partial y_k}{\partial a_k} = (y_k - t_k)f'(a_k) = \delta_k$$

$$\frac{\partial E}{\partial w_{kj}} = \frac{\partial E}{\partial a_k}\frac{\partial a_k}{\partial w_{kj}} = \delta_k z_j$$

$$\frac{\partial E}{\partial a_j} = \sum_{k=1}^{c}\frac{\partial E}{\partial y_k}\frac{\partial y_k}{\partial a_k}\frac{\partial a_k}{\partial z_j}\frac{\partial z_j}{\partial a_j} = \sum_{k=1}^{C}\delta_k w_{kj} g'(a_j) = \delta_j$$

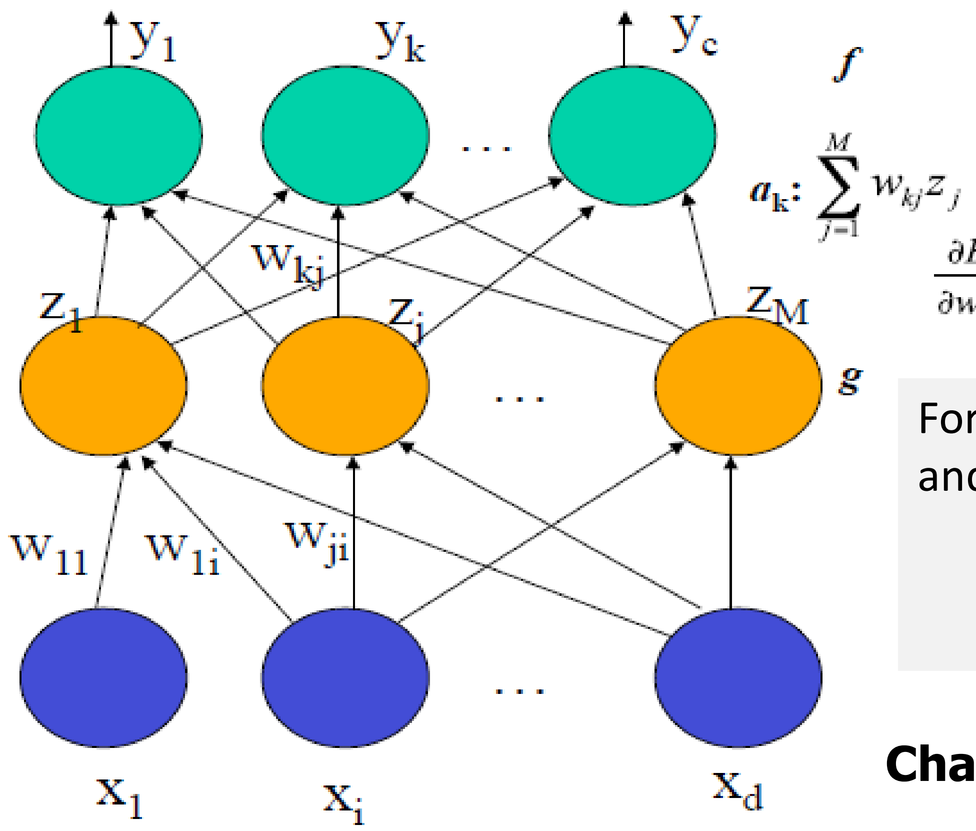
$$\frac{\partial E}{\partial w_{ji}} = \frac{\partial E}{\partial a_j}\frac{\partial a_j}{\partial wj\,i} = \delta_j x_i$$

Chain Rule of Calculating Derivatives

Modifications are made in the “**backwards**” direction **backpropagation**

For an Edge Between an Output Node and a Node in Last Hidden Layer

Backward Propagation



E is a function of y_k

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

y_k is a function of a_k

$$\frac{\partial E}{\partial y_k} = y_k - t_k$$

$$\frac{\partial E}{\partial a_k} = \frac{\partial E}{\partial y_k}\frac{\partial y_k}{\partial a_k} = (y_k - t_k) f'(a_k) = \delta_k$$

a_k is a function of w_{kj}

$$\frac{\partial E}{\partial w_{kj}} = \frac{\partial E}{\partial a_k}\frac{\partial a_k}{\partial w_{kj}} = \delta_k z_j$$

For an edge between output layer and hidden layer:

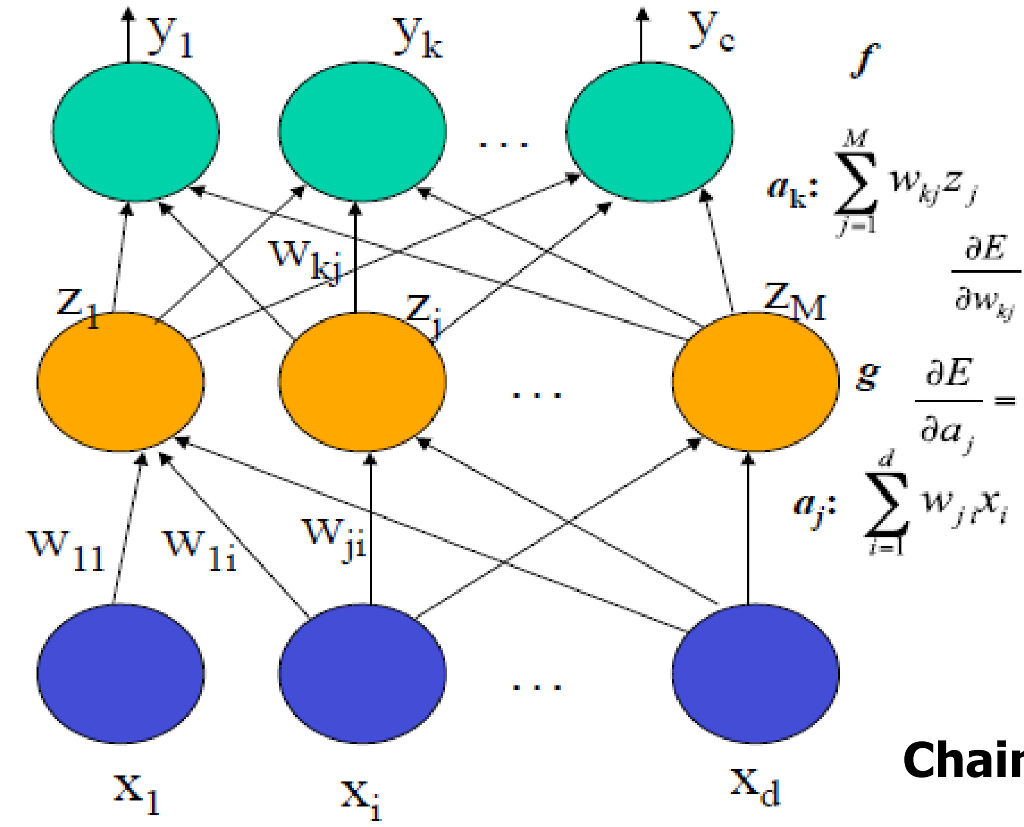
$$\Delta w k_j = -\mu \frac{\partial E}{\partial wkj}$$

Chain Rule of Calculating Derivatives

Modifications are made in the "**backwards**" direction **backpropagation**

For an Edge Between an Input Node and a Node in a Hidden Layer $\Delta wji = -\mu \frac{\partial E}{\partial wji}$

Backward Propagation



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

E is a function of y_k

$$\frac{\partial E}{\partial y_k} = y_k - t_k$$

y_k is a function of a_k

$$\frac{\partial E}{\partial a_k} = \frac{\partial E}{\partial y_k}\frac{\partial y_k}{\partial a_k} = (y_k - t_k) f'(a_k) = \delta_k$$

$$\frac{\partial E}{\partial w_{kj}} = \frac{\partial E}{\partial a_k}\frac{\partial a_k}{\partial w_{kj}} = \delta_k z_j$$

a_k is a function of z_j

$$\frac{\partial E}{\partial a_j} = \sum_{k=1}^{c}\frac{\partial E}{\partial y_k}\frac{\partial y_k}{\partial a_k}\frac{\partial a_k}{\partial z_j}\frac{\partial z_j}{\partial a_j} = \sum_{k=1}^{C}\delta_k w_{kj} g'(a_j) = \delta_j$$

z_j is a function of a_j

$$\frac{\partial E}{\partial w_{ji}} = \frac{\partial E}{\partial a_j}\frac{\partial a_j}{\partial wj\,i} = \delta_j x_i$$

a_j is a function of w_{ji}

Chain Rule of Calculating Derivatives

Modifications are made in the “**backwards**” direction **backpropagation**

Detailed Backpropagation Algorithm

Algorithm: Backpropagation. Neural network learning for classification or numeric prediction, using the backpropagation algorithm.

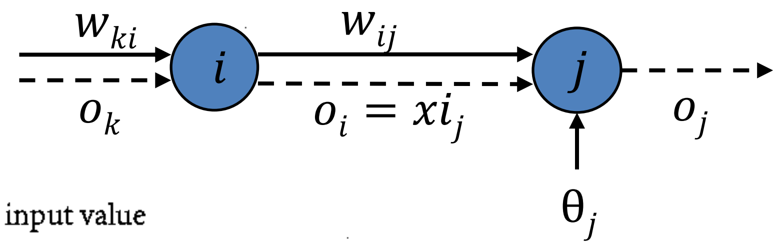
Input:

- *D*, a data set consisting of the training tuples and their associated target values;
- *l*, the learning rate;
- *network*, a multilayer feed-forward network.

Output: A trained neural network.

Method:

```
(1)  Initialize all weights and biases in network;
(2)  while terminating condition is not satisfied {
(3)      for each training tuple X in D {
(4)          // Propagate the inputs forward:
(5)          for each input layer unit j {
(6)              $O_j = I_j$; // output of an input unit is its actual input value
(7)          for each hidden or output layer unit j {
(8)              $I_j = \sum_i w_{ij} O_i + \theta_j$; //compute the net input of unit j with respect to
                     the previous layer, i
(9)              $O_j = \frac{1}{1+e^{-I_j}}$; } // compute the output of each unit j
(10)         // Backpropagate the errors:
(11)         for each unit j in the output layer
(12)             $Err_j = O_j(1-O_j)(T_j - O_j)$; // compute the error
(13)         for each unit j in the hidden layers, from the last to the first hidden layer
(14)             $Err_j = O_j(1-O_j)\sum_k Err_k w_{jk}$; // compute the error with respect to
                     the next higher layer, k
(15)         for each weight $w_{ij}$ in network {
(16)             $\Delta w_{ij} = (l) Err_j O_i$; // weight increment
(17)             $w_{ij} = w_{ij} + \Delta w_{ij}$; } // weight update
(18)         for each bias $\theta_j$ in network {
(19)             $\Delta \theta_j = (l) Err_j$; // bias increment
(20)             $\theta_j = \theta_j + \Delta \theta_j$; } // bias update
(21)         } }
```



Updating Errors

- Errors are updated either after considering each training sample in the network or after all training samples are considered
- The first approach is called the incremental, online, or case approach “or updating”
- The second approach is called the batch, offline, or epoch approach “or updating”
- An iteration on all tuples is called an **Epoch**
- An iteration through the whole training set is called an epoch