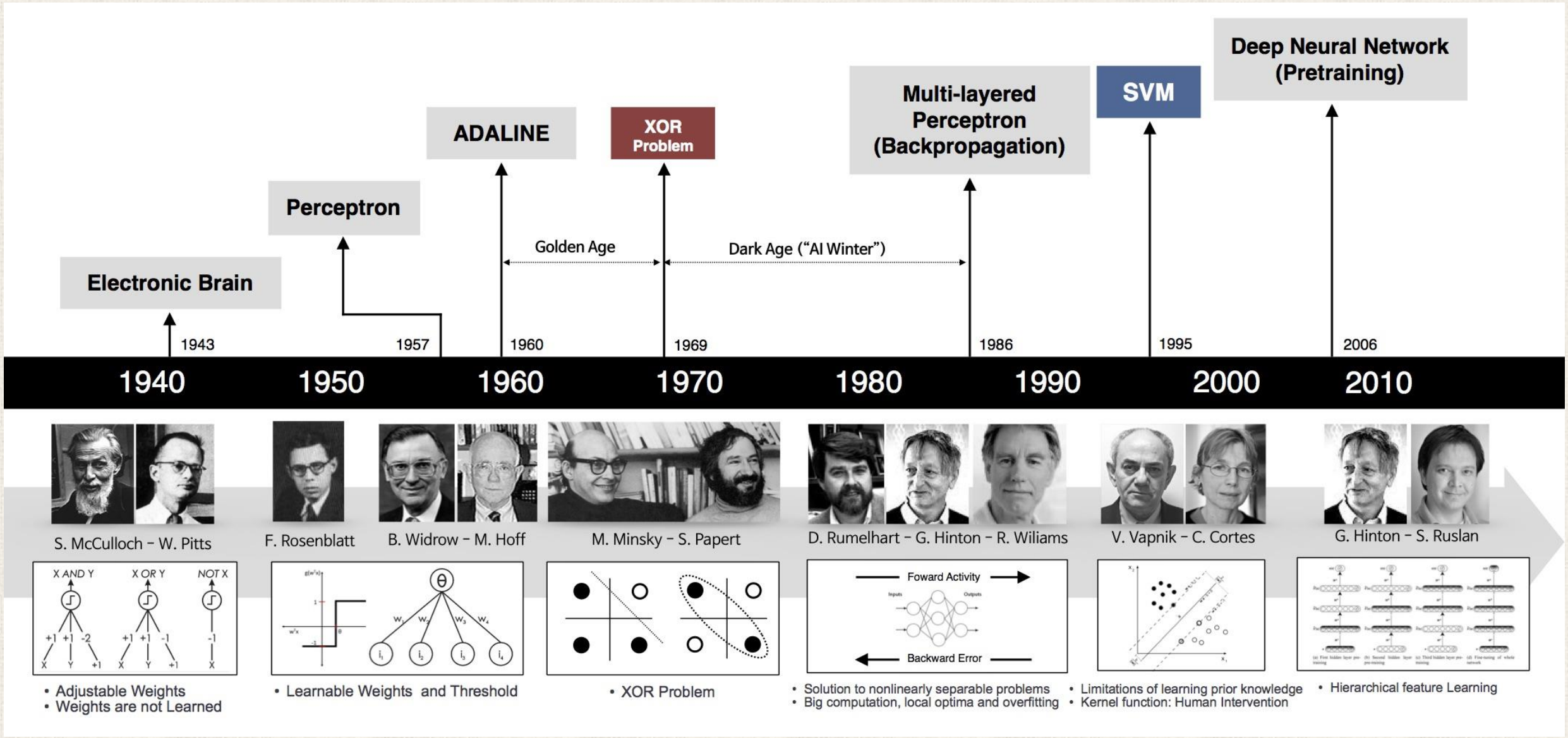
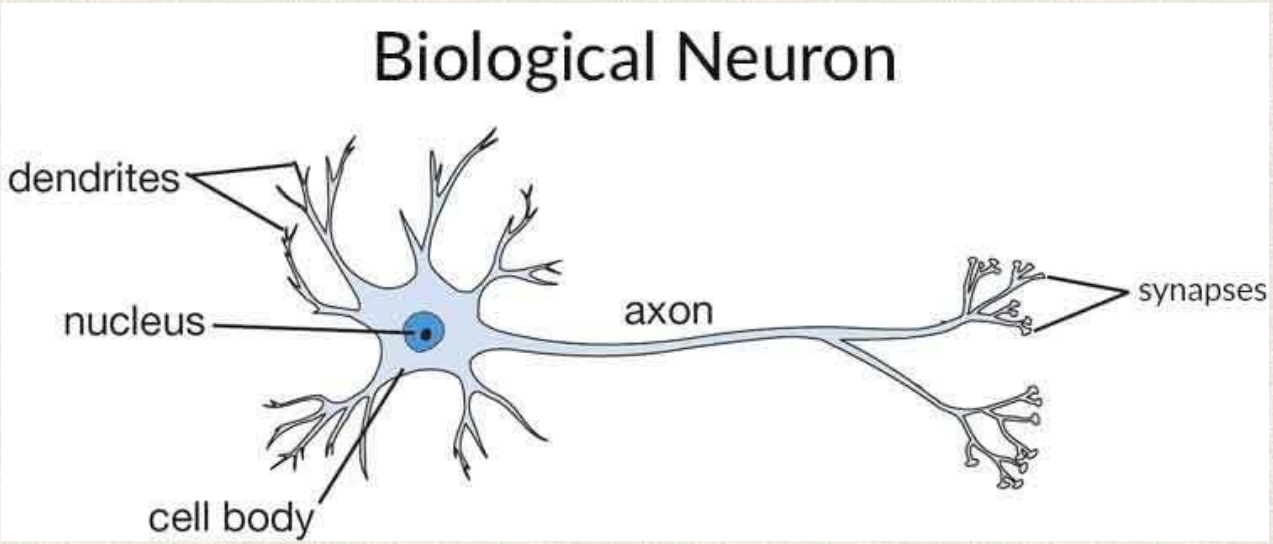
Artificial Neural Networks – A Brief History

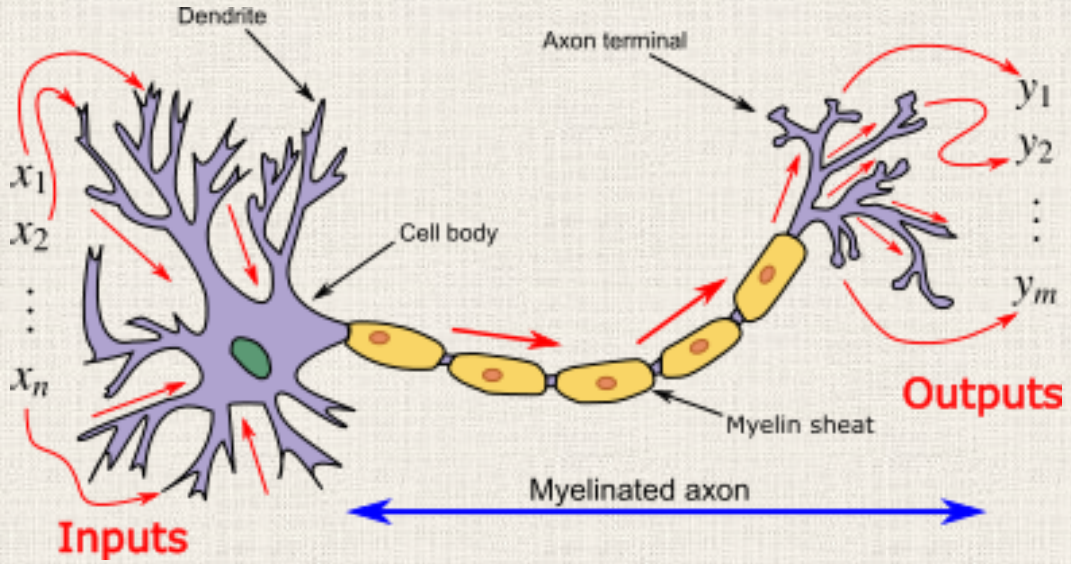


- The preliminary theoretical base for contemporary neural networks was independently proposed by *A. Bain* (1873) and *W. James* (1890). In their work, ***both thoughts and body activities resulted from interactions among neurons within the brain.***
- *McCulloch* and *Pitts* (1943) created a computational model for neural networks based on mathematics and algorithms. They call the model *"threshold logic".*
- In the late 1940s, *Donald Hebb* created a hypothesis of learning based on the mechanism of neural plasticity that is now known as ***Hebbian learning***. "cells that fire together, wire together."
- Farley and *Clark* (1954) first used computational machines to simulate a ***Hebbian network*** at MIT.
- *Rosenblatt* (1958) created the ***Perceptron***.
- Neural network research stagnated after the publication of machine learning research by *Minsky* and *Papert* (1969) which discovered two issues:
  - The single-layer neural networks were incapable of processing the XOR problem
  - Computers are not sophisticated enough to effectively handle the long run time by large neural networks
- *Werbos* (1975) created the ***backpropagation*** algorithm.
- The parallel distributed processing of the mid-1980s become popular under the name ***connectionism***.

Artificial Neural Networks – A Brief History

- ***Artificial Neural Networks*** (ANN) or ***connectionist systems*** are computing systems vaguely inspired by the *biological neural networks* that constitute animal brains. Such systems “learn” to perform tasks by considering examples, generally without being programmed with task-specific rules.

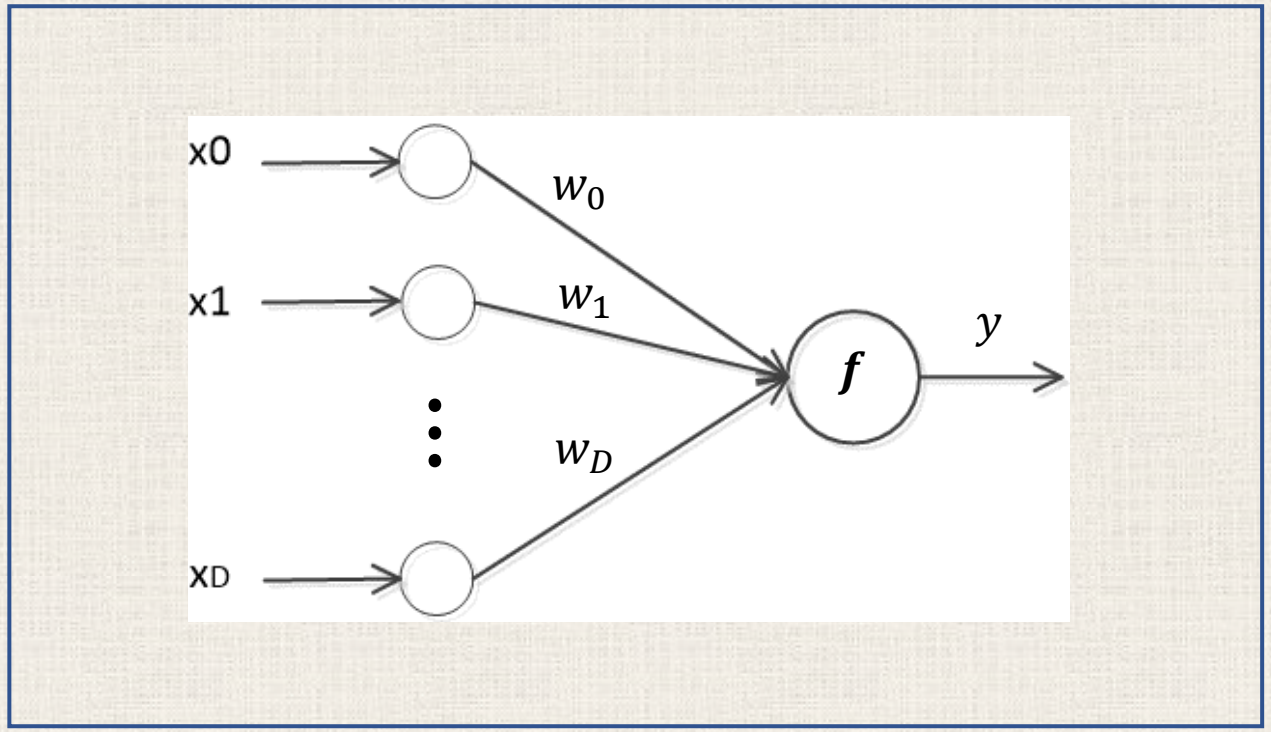




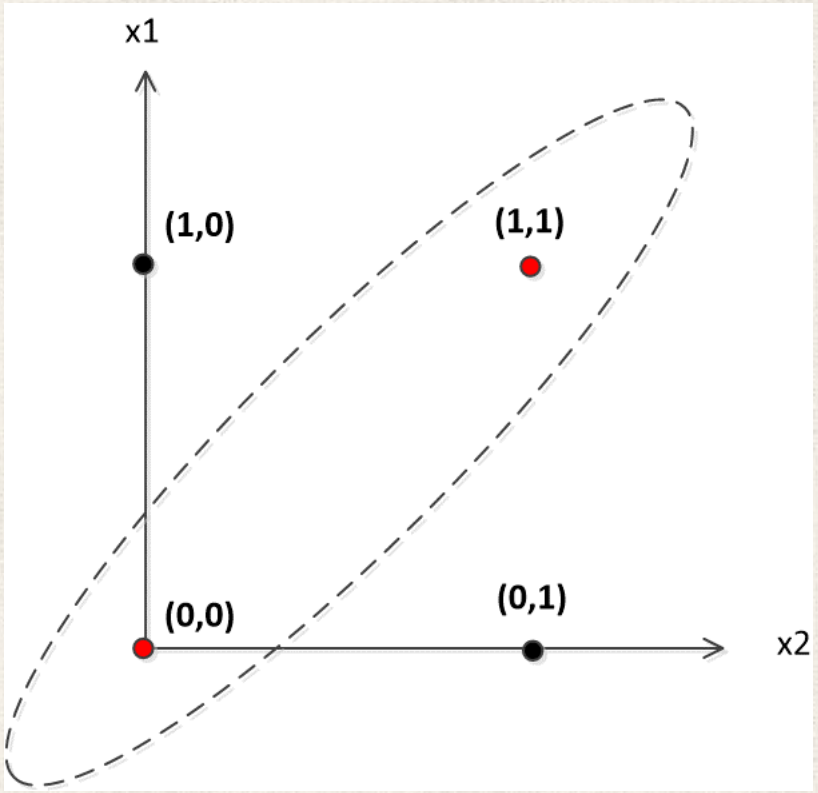
- ***Dendrites*** are fibers which emanate from the cell body and provide the receptive zones that receive activations from other neurons
- ***Axons*** are fibers acting as transmission lines that send activation to other neurons
- ***Synapses*** are the junctions that allow signal transmission between the axons and dendrites (from other neurons)
- The majority of neurons encode their activations or outputs as a series of brief ***electrical pulses***.
- The neuron's ***cell body*** (soma) processes the incoming activations and converts them into output activations
- The neuron's ***nucleus*** contains the genetic material in the form of DNA.

Artificial Neural Networks (ANN) Model

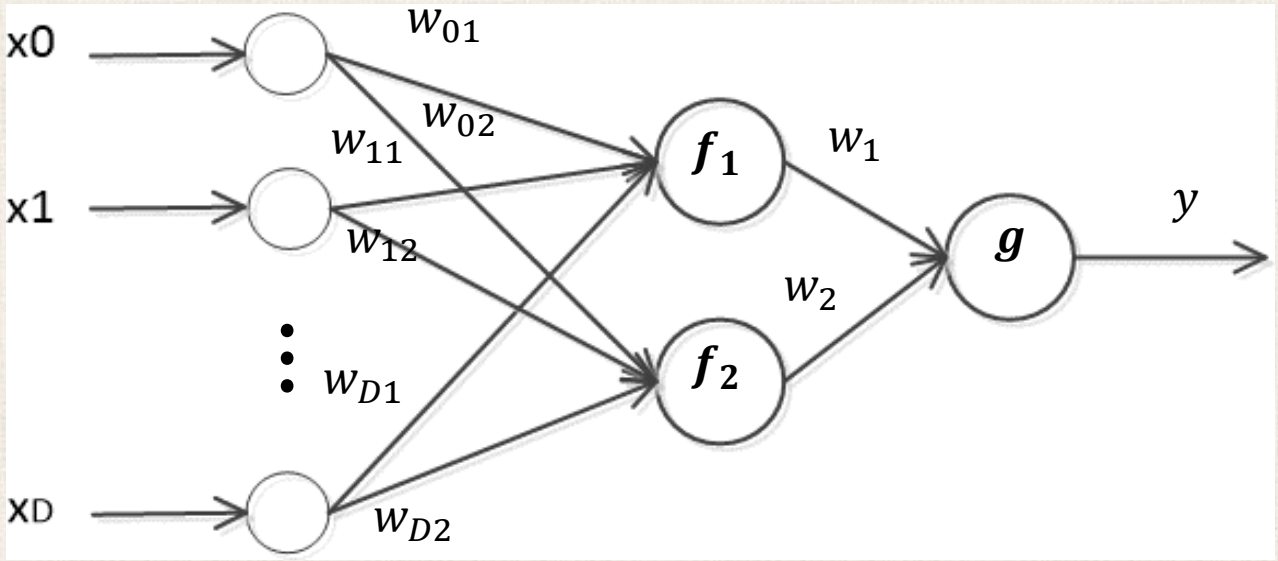
Perceptron is a linear classifier. One single Perceptron model can not even solve the XOR problem

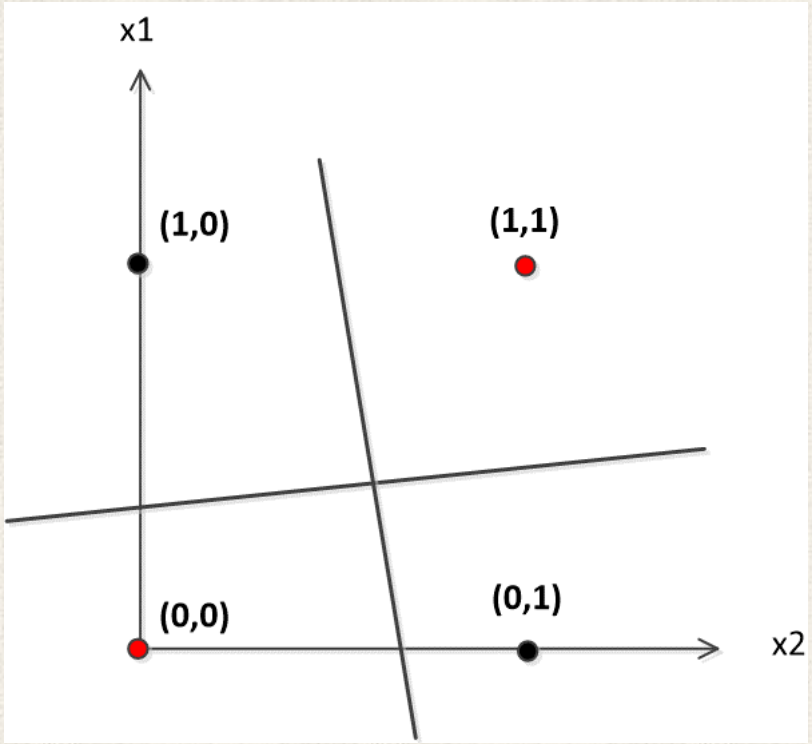


Computational model of a biological neuron
The “Threshold Logic” model

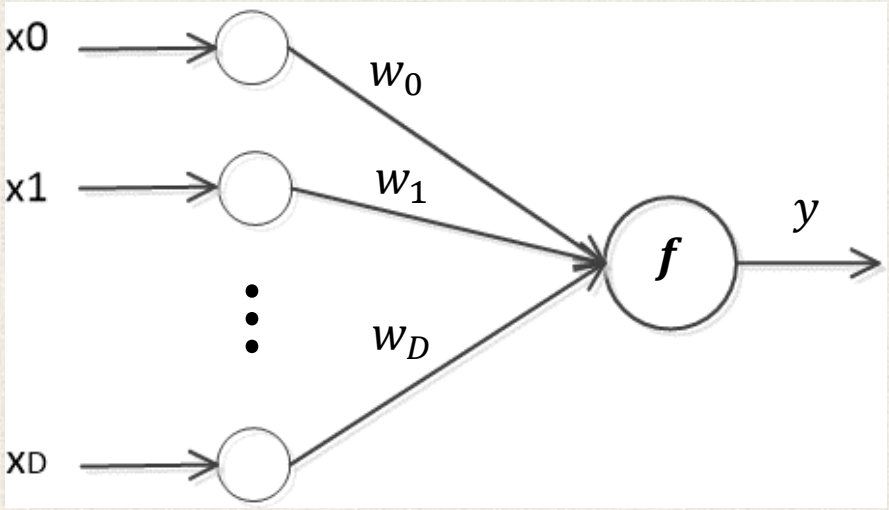


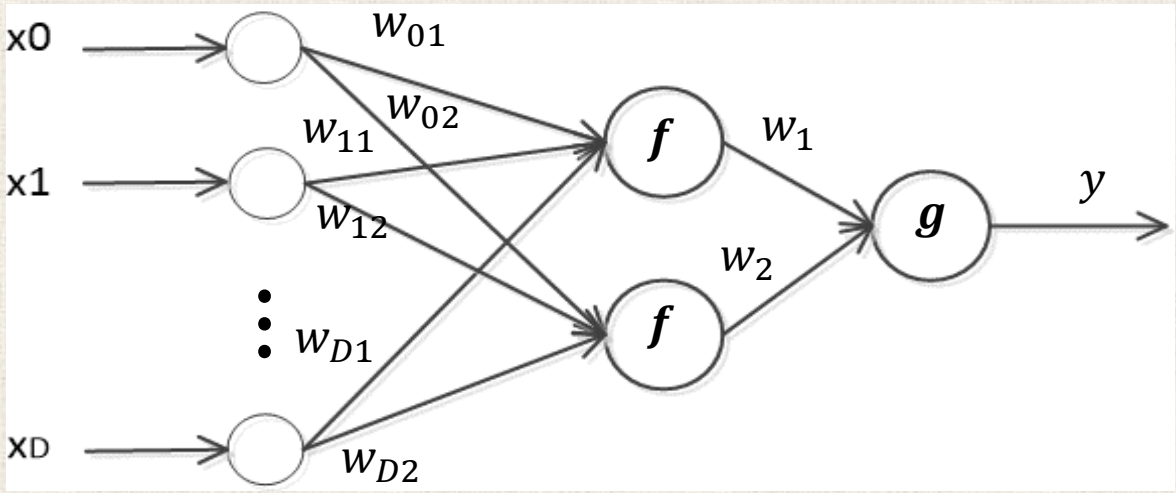
How about using two Perceptrons?



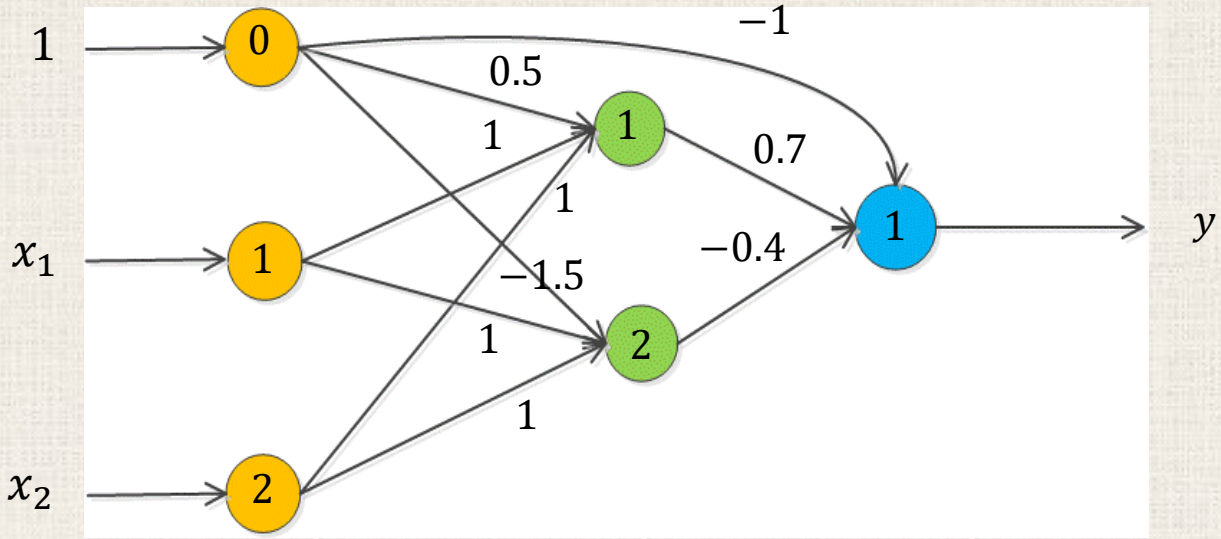


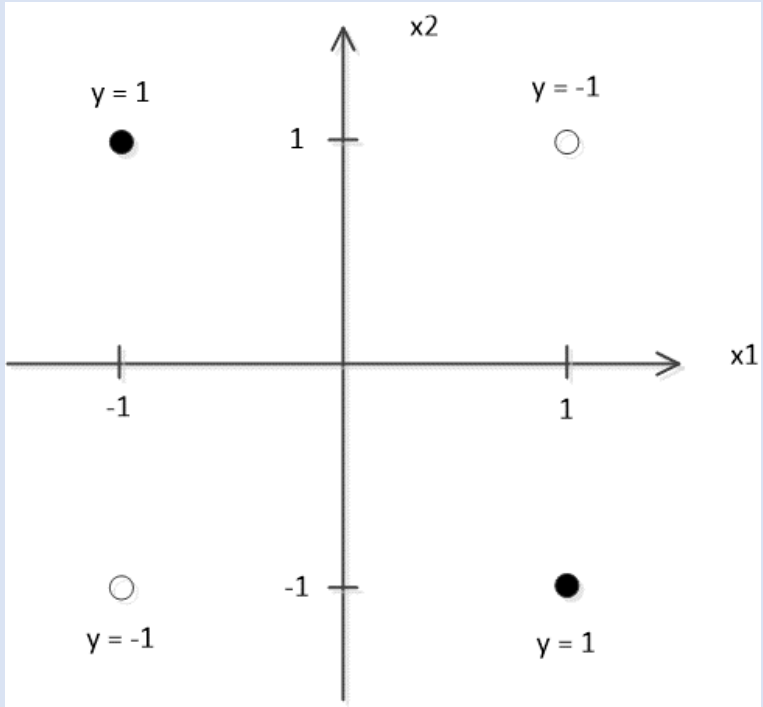
Comparison of a single Perceptron and Multi-layer Perceptron:





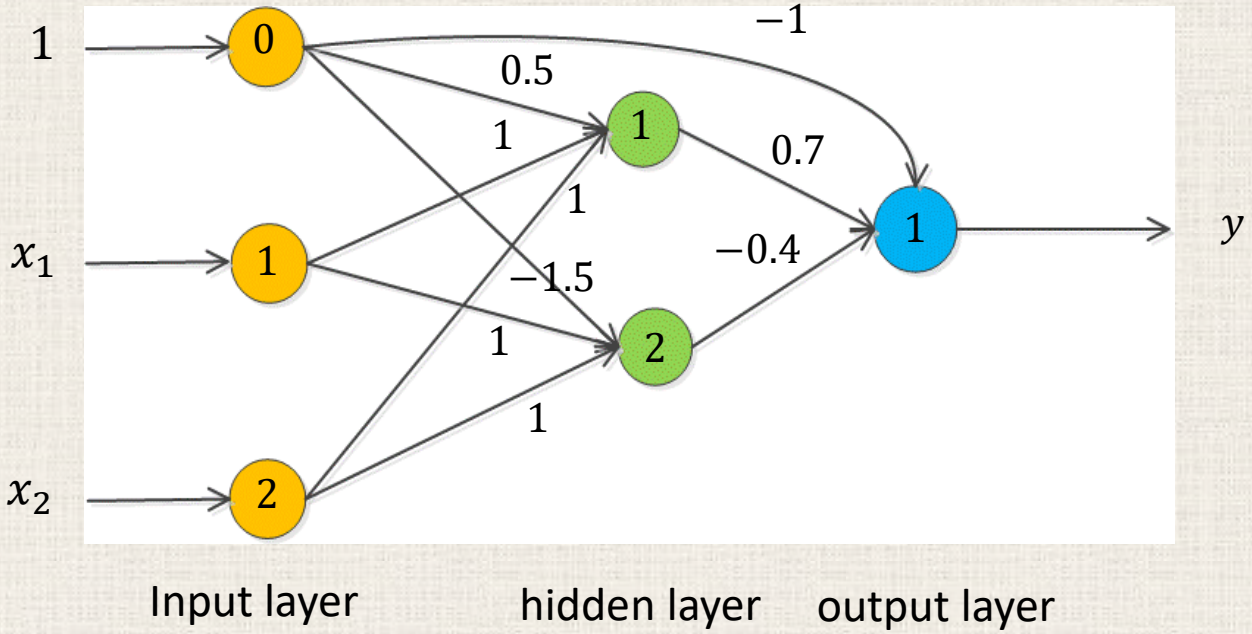
A Multi-layer Perceptron that solves the XOR problem





- At hidden node 1: $net_1 = 0.5 + x_1 + x_2, \quad z_1 = sign(net_1)$
- At hidden node 2: $net_2 = -1.5 + x_1 + x_2, \quad z_2 = sign(net_2)$
- At output node 1: $net = -1 + 0.7z_1 - 0.4z_2, \quad y = sign(net)$

x_1	x_2	y
1	1	−1
−1	−1	−1
1	−1	1
−1	1	1



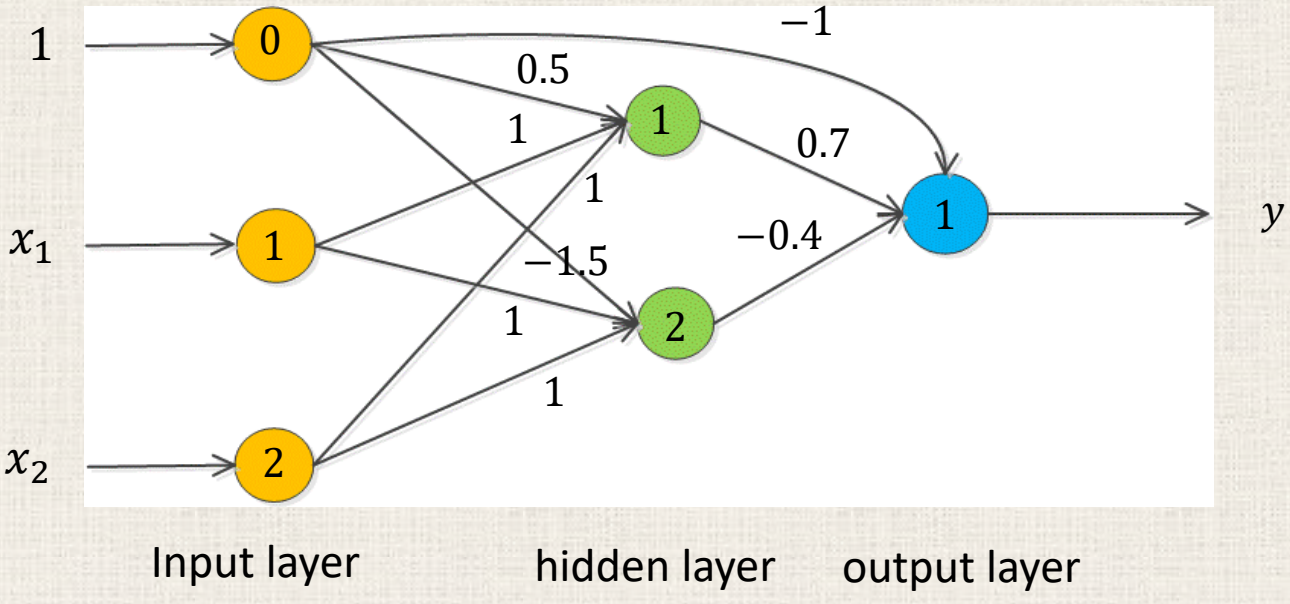
x_1	x_2	y
1	1	-1
-1	-1	-1
1	-1	1
-1	1	1

Let's feed $(1,1)^T$ into the network:

At hidden node 1: $net_1 = 0.5 + x_1 + x_2 = 2.5,$ $z_1 = sign(net_1) = 1$

At hidden node 2: $net_2 = -1.5 + x_1 + x_2 = 0.5,$ $z_2 = sign(net_2) = 1$

At output node 1: $net = -1 + 0.7z_1 - 0.4z_2 = -0.7,$ $y = sign(net) = -1$



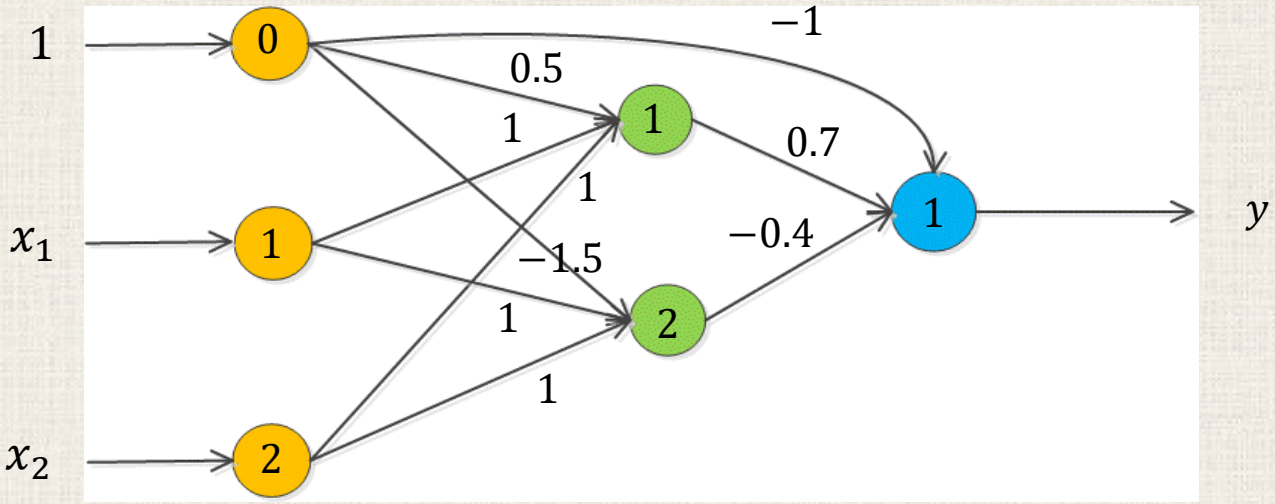
x_1	x_2	y
1	1	−1
−1	−1	−1
1	−1	1
−1	1	1

Let's feed $(-1,-1)^T$ into the network:

At hidden node 1: $net_1 = 0.5 + x_1 + x_2 = -1.5,$ $z_1 = sign(net_1) = -1$

At hidden node 2: $net_2 = -1.5 + x_1 + x_2 = -3.5,$ $z_2 = sign(net_2) = -1$

At output node 1: $net = -1 + 0.7z_1 - 0.4z_2 = -1.3,$ $y = sign(net) = -1$



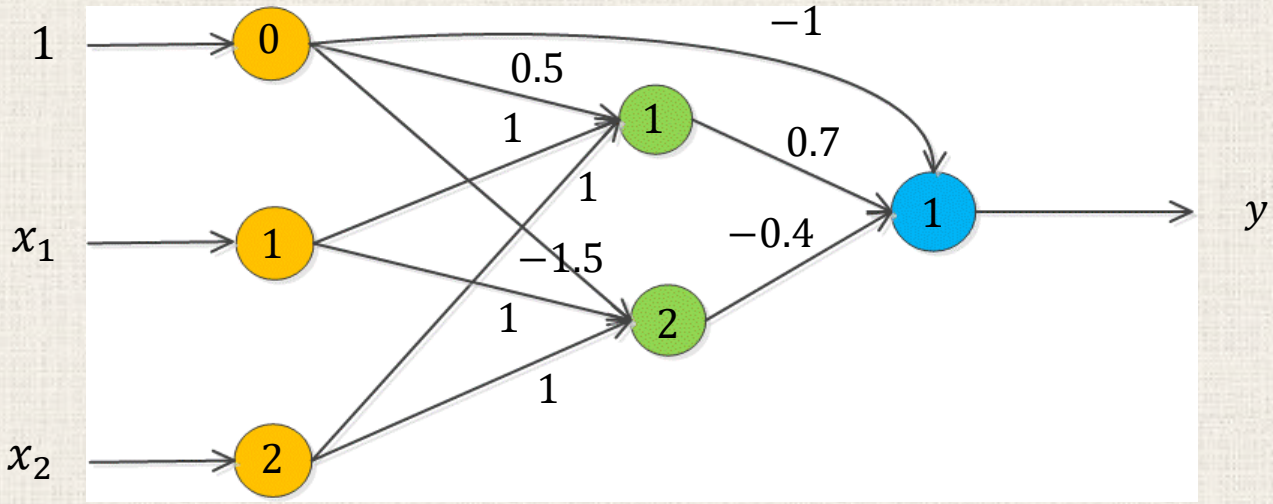
x_1	x_2	y
1	1	-1
-1	-1	-1
1	-1	1
-1	1	1

Let's feed $(1, -1)^T$ into the network:

At hidden node 1: $net_1 = 0.5 + x_1 + x_2 = 0.5,$ $z_1 = sign(net_1) = 1$

At hidden node 2: $net_2 = -1.5 + x_1 + x_2 = -1.5,$ $z_2 = sign(net_2) = -1$

At output node 1: $net = -1 + 0.7z_1 - 0.4z_2 = 0.1,$ $y = sign(net) = 1$



x_1	x_2	y
1	1	−1
−1	−1	−1
1	−1	1
−1	1	1

Let's feed $(-1,1)^T$ into the network:

At hidden node 1: $net_1 = 0.5 + x_1 + x_2 = 0.5,$ $z_1 = sign(net_1) = 1$

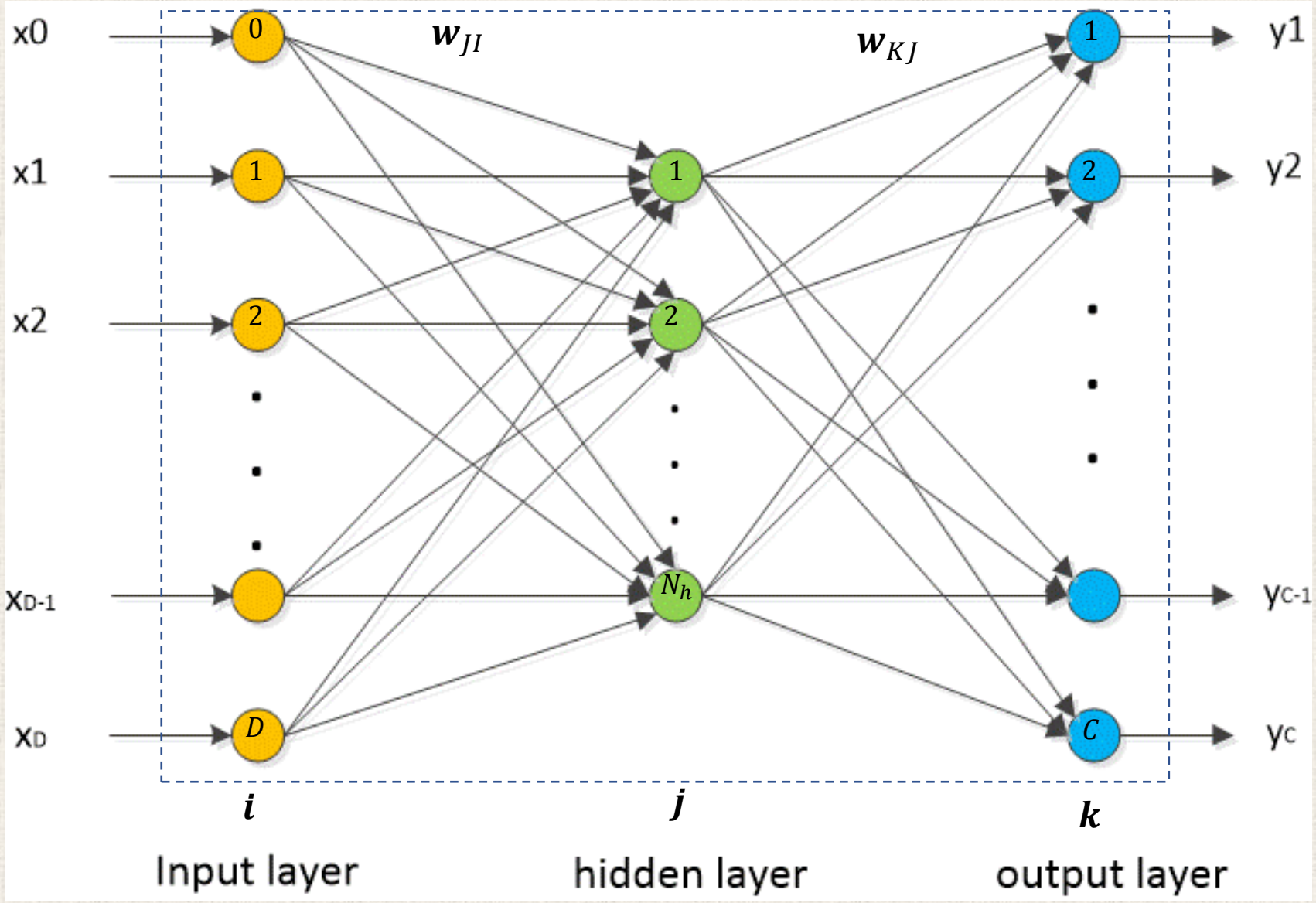
At hidden node 2: $net_2 = -1.5 + x_1 + x_2 = -1.5,$ $z_2 = sign(net_2) = -1$

At output node 1: $net = -1 + 0.7z_1 - 0.4z_2 = 0.1,$ $y = sign(net) = 1$

Note:

Sign function is used in the example as the activation function. However, in real multi-layer perceptron model, we usually use sigmoid function as the activation function instead.

Artificial Neural Networks (ANN) (Feedforward Networks)(Multi-Layer Perceptron)



Artificial Neural Networks (ANN) (Feedforward Networks)(Multi-Layer Perceptron)

- Input layer nodes transmit input values to the hidden layer nodes without doing any computation
- The number of nodes in input layer is determined by the dimension of the input features
- The number of nodes in output layer equals the number of classes (***one-hot vector coding***)
- Lower layer nodes only feed to next layer nodes (***feed forward structure***)
- $i, j\ and\ k$ represent the indexes of nodes in input, hidden and output layers respectively
- ***Complete connection***
- Multiple hidden layers possible

The hidden Layer

- Each hidden node computes the weighted sum of its inputs to form a scalar ***net activation***,

$$net_j = \sum_{i=0}^{D} x_i w_{ji} = \boldsymbol{w}_j^T \boldsymbol{x}$$

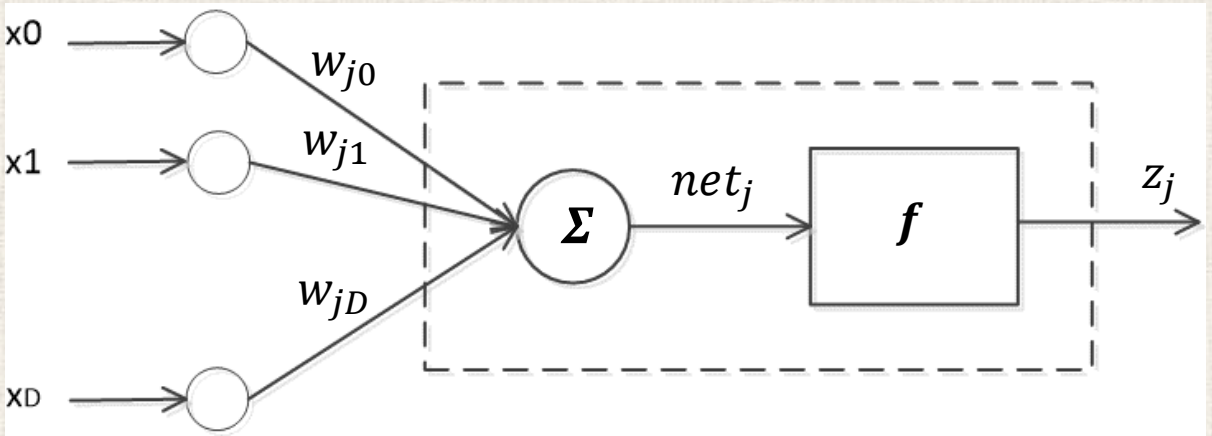
Where, $\boldsymbol{w}_j = [w_{j0} \quad \cdots \quad w_{jD}]^T$ is the input-to-hidden layer weight vector at the hidden node j, $\boldsymbol{x} = [x_0 \quad \cdots \quad x_D]^T$ is the input vector.

- Each hidden node emits an output that is a nonlinear function of its ***activation***

$$z_j = f(net_j)$$

f is the ***activation function***

hidden node j



The output layer

- Each output node similarly computes its ***net activation*** based on hidden nodes outputs

$$net_k = \sum_{j=1}^{N_h} z_j w_{kj} = \boldsymbol{w}_k^T \boldsymbol{z}$$

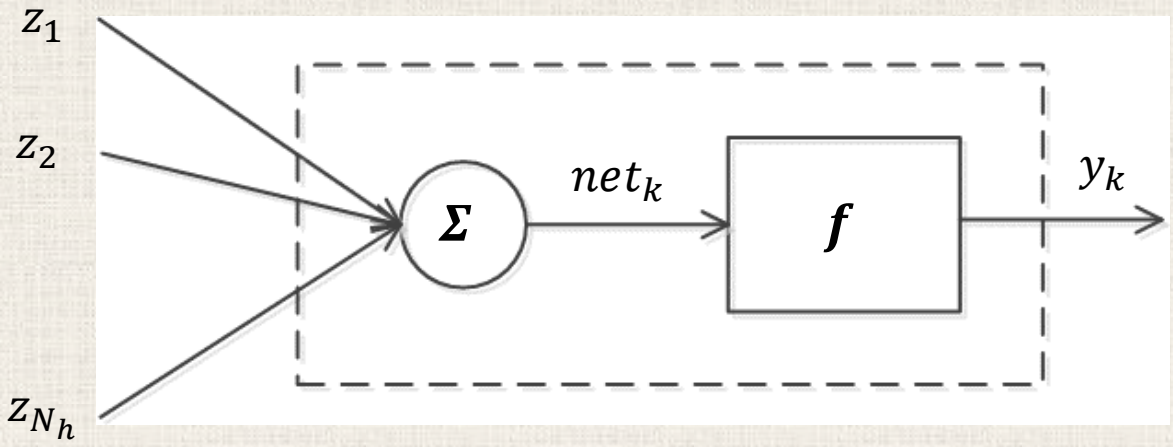
Where, w_{kj} denotes the hidden-to-output layer weights at the output node k. N_h denotes the number of hidden nodes.

- Each output node emits output

$$y_k = f(net_k)$$

f is the ***activation function***

output node k



Artificial Neural Networks (ANN) -The activation function

The nonlinear ***activation function*** should be continuous and differentiable.

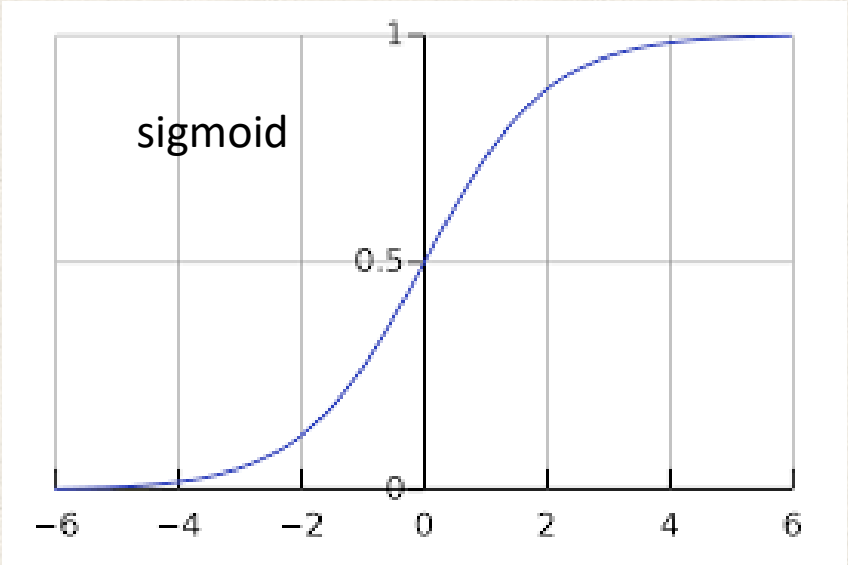
For example, the ***sigmoid*** function and ***tanh*** are commonly used ones:

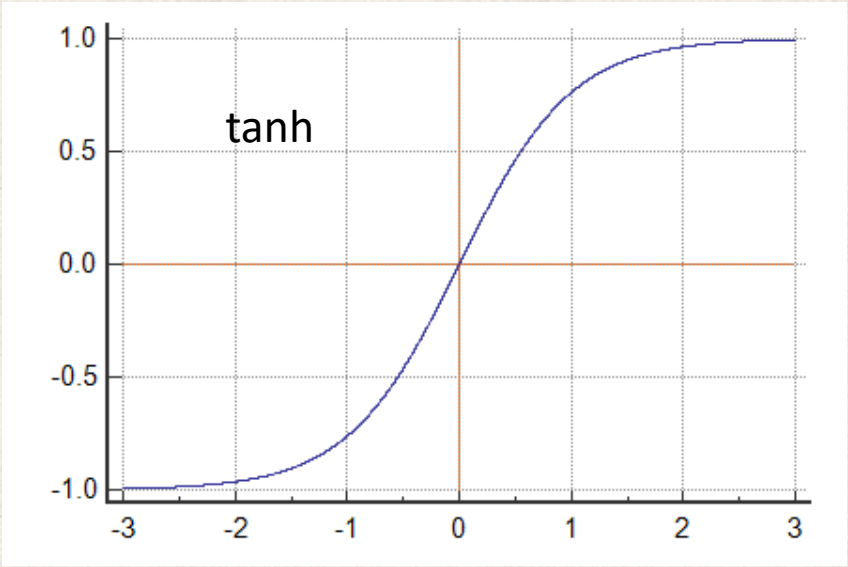
$$f(net) = \frac{1}{1 + e^{-net}}$$

$$f'(net) = f(net)(1 - f(net))$$

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

$$f'(net) = 1 - f^2(net)$$



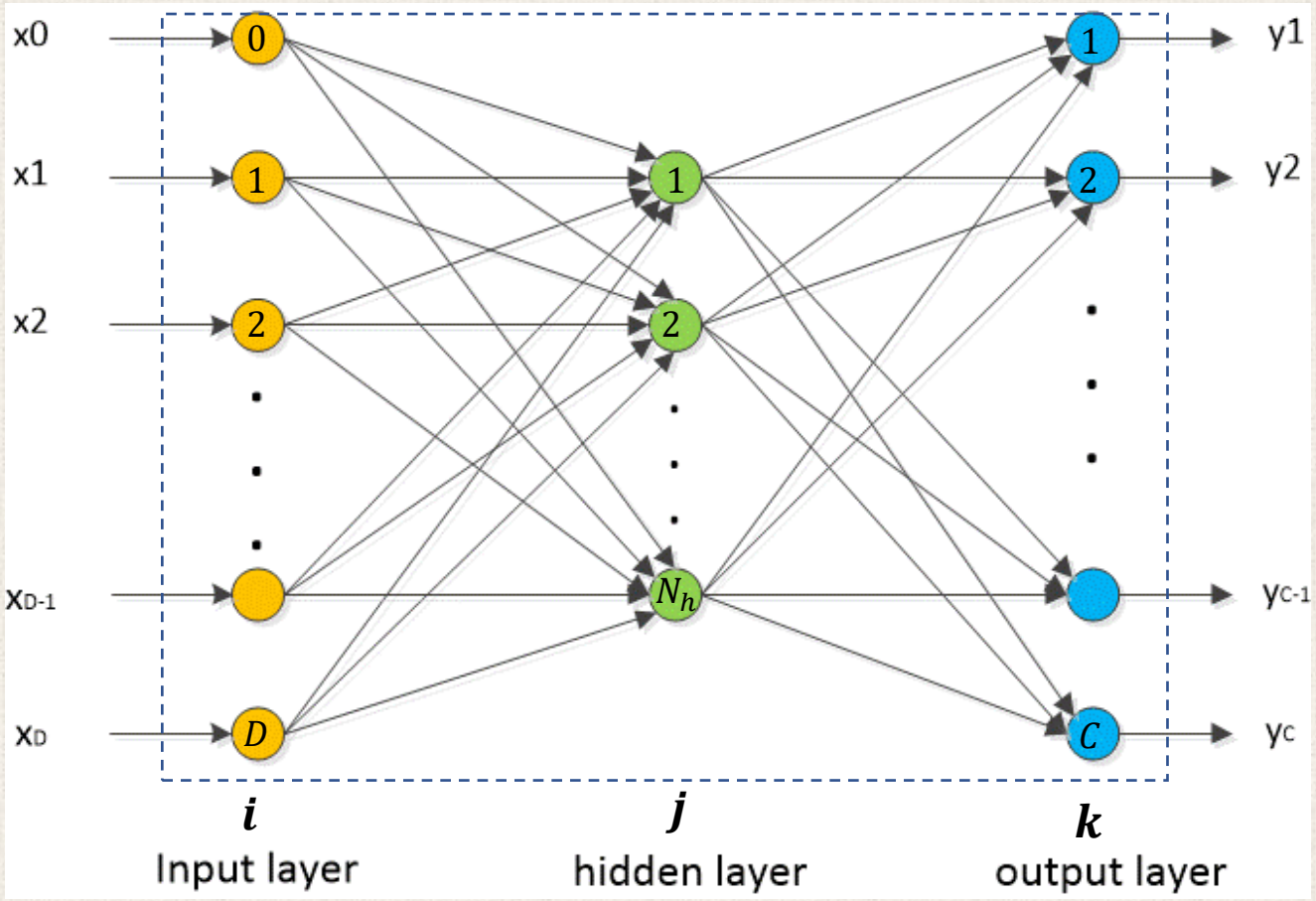


Artificial Neural Networks (ANN) as a universal approximator

- ANN works like a discriminant function as

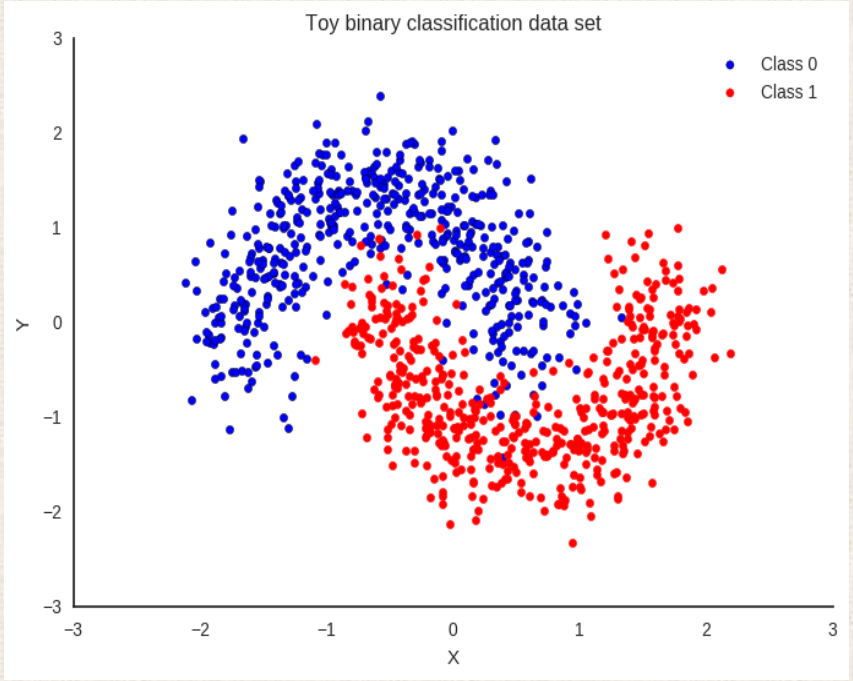
$$g(\boldsymbol{x}) = y_k = f\left(\sum_{j=1}^{N_h} w_{kj} f\left(\sum_{i=0}^{D} x_i w_{ji}\right)\right)$$

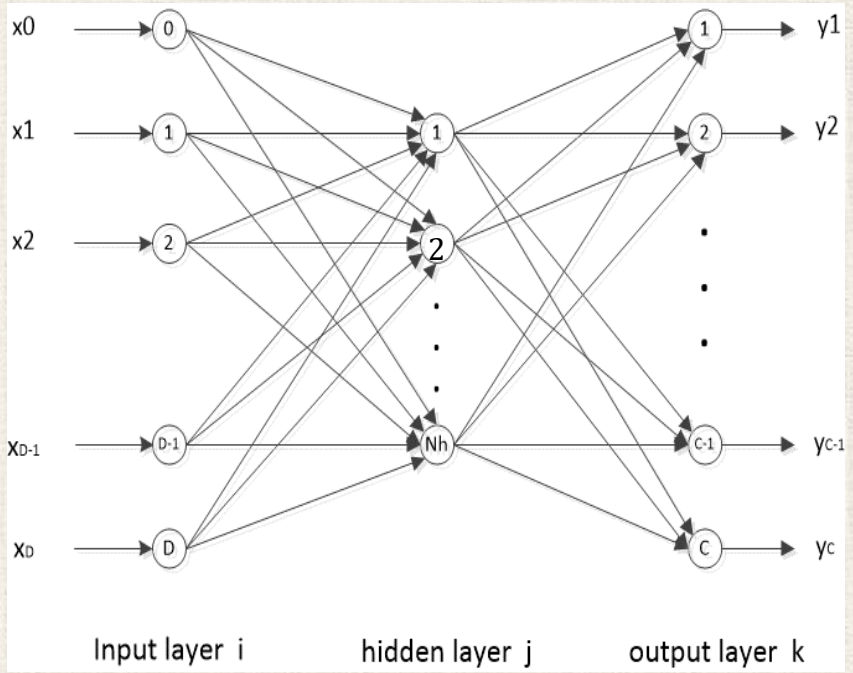
- ***Universal expressive power***: *Any continuous mapping can be approximated as close as needed by a trained three-layer ANN with enough number of nodes in the hidden layer.*



What if a linear function is used as activation function?

Artificial Neural Networks (ANN) - The training problem





$$\min_{\boldsymbol{w}} E(\boldsymbol{w})$$

Artificial Neural Networks (ANN) - The training problem

- Given a training data set comprising of a set of input vectors $\{\boldsymbol{x}_n\}, n = 1,2, \dots, N$, together with a corresponding set of target values $\{\boldsymbol{t}_n\}$, we want to determine the network parameters (the weights) $\boldsymbol{w}$ that minimizes the error function:

$$E(\boldsymbol{w}) = \sum_{n=1}^{N} E_n(\boldsymbol{w}) = \frac{1}{2}\sum_{n=1}^{N} \|\boldsymbol{y}(\boldsymbol{x}_n, \boldsymbol{w}) - \boldsymbol{t}_n\|^2$$

  Where $\boldsymbol{y}(\boldsymbol{x}_n, \boldsymbol{w})$ is the output generated by the network for input $\boldsymbol{x}_n$

  Which is a nonlinear function of the parameter $\boldsymbol{w}$ and is nonconvex in general.

- Notice that $\boldsymbol{w}$ consists of the weights associated with all the hidden and output nodes

- How to solve the training problem of ANN?

Gradient descent + Backpropagation

- We use gradient descent method to update the weights:

$$\boldsymbol{w}^{\tau+1} = \boldsymbol{w}^{\tau} - \eta \nabla E(\boldsymbol{w}^{\tau})$$

Where, τ denote the iteration steps, $\eta > 0$ is known as the *learning rate*. $\nabla E(\boldsymbol{w}^{\tau})$ is the *gradient* of the error function evaluated at $\boldsymbol{w} = \boldsymbol{w}^{\tau}$.

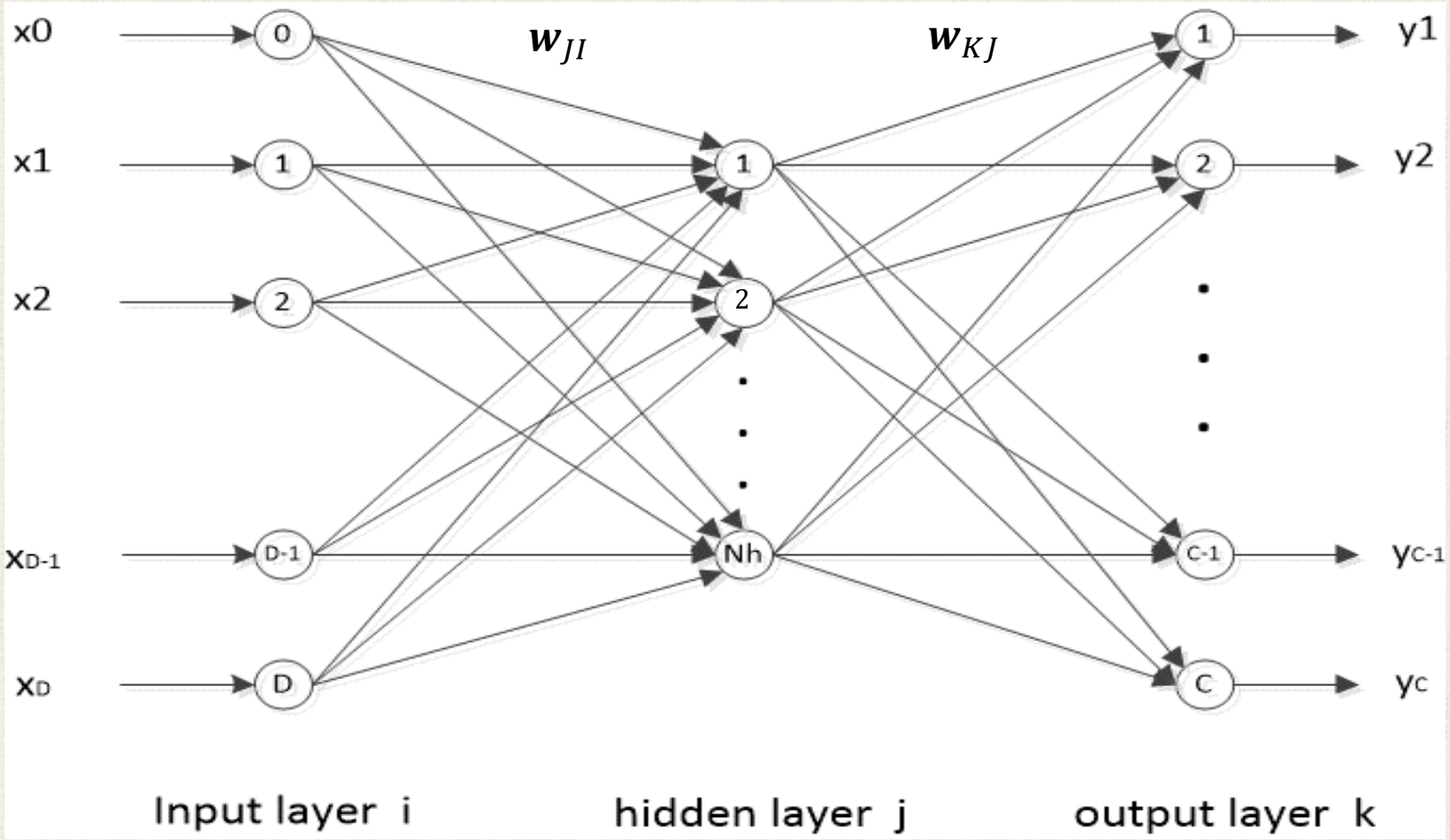
- We use the ***backpropagation*** algorithm to determine $\nabla E(\boldsymbol{w}^{\tau})$.

How the backpropagation algorithm works?

forward propagation

Training samples

Classification using current weights



backward propagation

Training errors

Weights update

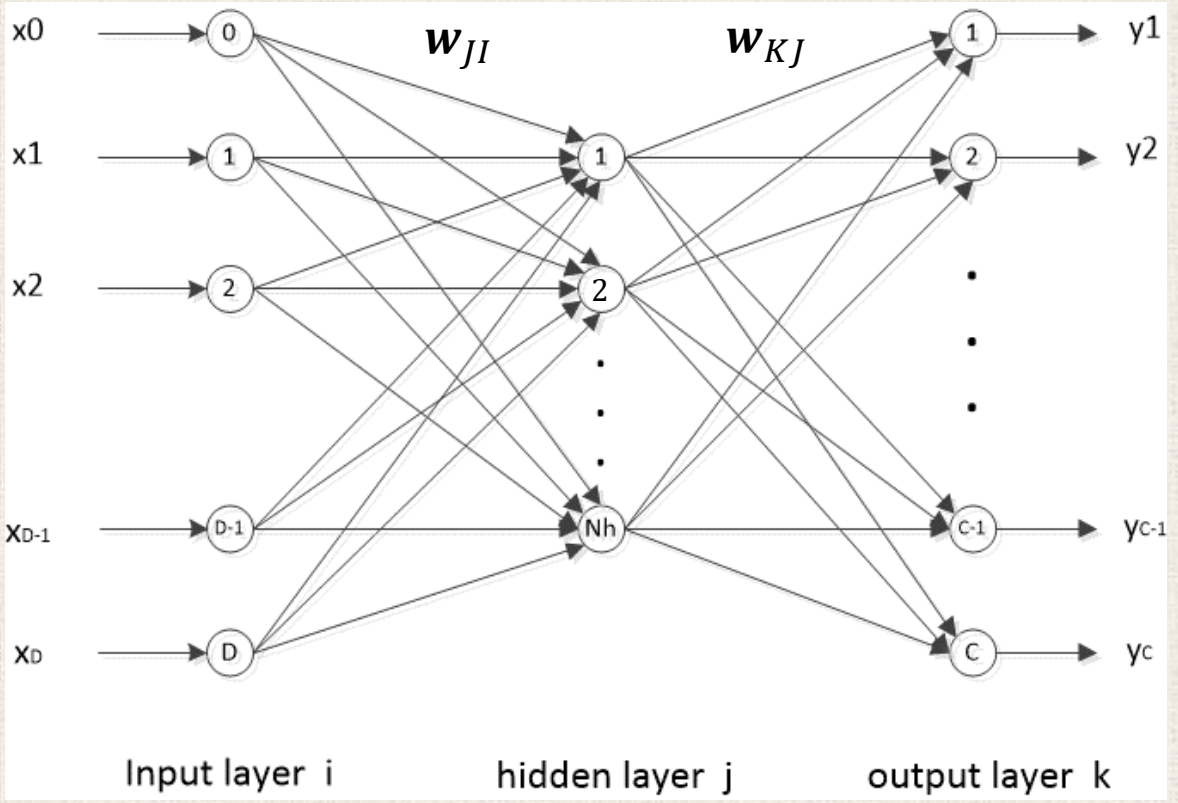
The network structure and notations:

At hidden layer, node j:

$$net_j = \sum_{i=0}^{D} x_i w_{ji} = \boldsymbol{w}_j^T \boldsymbol{x}; \quad z_j = f(net_j)$$

Where,

- net_j is the ***net activation*** to node j;
- z_j is the output of node j;
- w_{ji} is the weight from node j to the node i in the input layer; $\boldsymbol{w}_j = (w_{j0}\ w_{j1}\ w_{j2}\ \dots w_{jD})^T$ is the weight vector toward node j;
- $\boldsymbol{x}^T = (x_0\ x_1\ x_2\ \dots x_D)$ is the input feature vector including the bias.



$$net = \boldsymbol{w}_{JI}\boldsymbol{x}$$

$$\begin{bmatrix} net_1 \\ \vdots \\ net_{Nh} \end{bmatrix} = \begin{bmatrix} w_{10} & w_{11} & \cdots & w_{1D} \\ \vdots & \vdots & \vdots & \vdots \\ w_{Nh0} & w_{Nh1} & \cdots & w_{NhD} \end{bmatrix} \begin{bmatrix} x_0 \\ x_1 \\ \vdots \\ x_D \end{bmatrix}$$

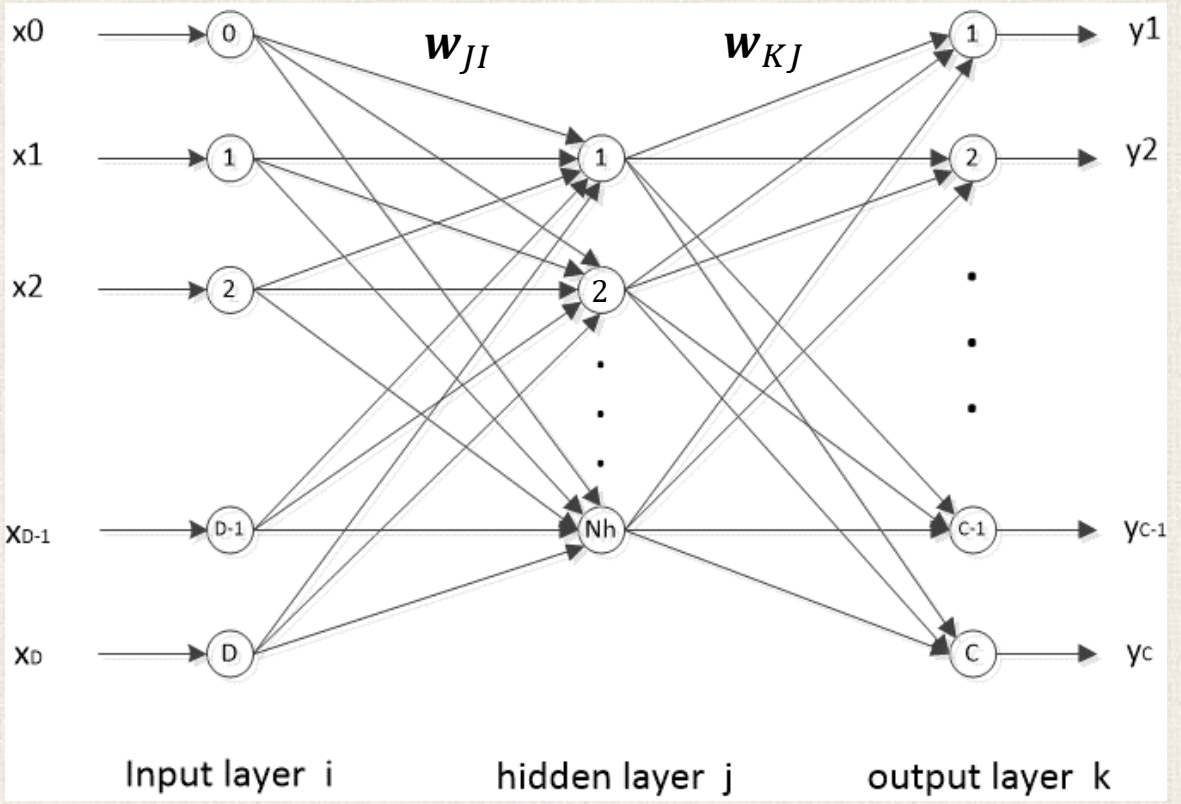
The network structure and notations:

At output layer, node k:

$$net_k = \sum_{j=1}^{N_h} z_j w_{kj} = \boldsymbol{w}_k^T \boldsymbol{z}; \quad y_k = f(net_k)$$

Where,

- net_k is the ***net activation*** to node k;
- y_k is the output of node k;
- $\boldsymbol{w}_k = (w_{k1}\ w_{k2} \ldots w_{kN_h})$ is the weight vector toward node k;
- $\boldsymbol{z}^T = (z_1\ z_2\ \ldots z_{Nh})$ is the output vector of the hidden layer



$$net = \boldsymbol{w}_{KJ}\boldsymbol{z}$$

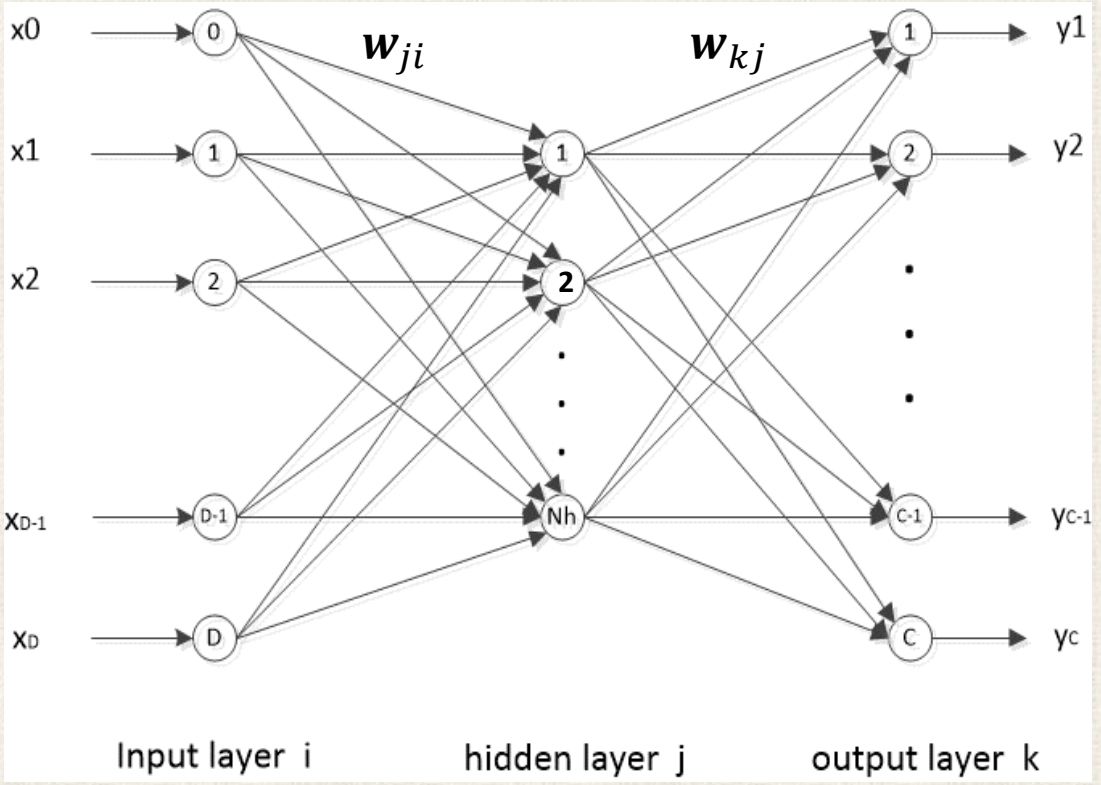
$$\begin{bmatrix} net_1 \\ \vdots \\ net_C \end{bmatrix} = \begin{bmatrix} w_{11} & \ldots & w_{1Nh} \\ \vdots & \vdots & \vdots \\ w_{C1} & \ldots & w_{CNh} \end{bmatrix} \begin{bmatrix} z_1 \\ \vdots \\ z_{Nh} \end{bmatrix}$$

Let's assume that an instance $(\boldsymbol{x}_n, \boldsymbol{t}_n)$ from the training data set is fed to the network, the forward propagation process is done, and the output $\boldsymbol{y}_n$ is generated. We have,

$\boldsymbol{t}_n = \begin{bmatrix} t_1 \\ t_2 \\ \vdots \\ t_c \end{bmatrix}$ is the target value, and $\boldsymbol{y}_n = \begin{bmatrix} y_1 \\ y_2 \\ \vdots \\ y_c \end{bmatrix}$ is the prediction using current weights. Then,

- Let's define the ***training error*** of an instance $(\boldsymbol{x}_n, \boldsymbol{t}_n)$ as:

$$E_n = \frac{1}{2}\sum_{k=1}^{C}(y_k - t_k)^2 = \frac{1}{2}\|\boldsymbol{y}_n - \boldsymbol{t}_n\|^2$$



- **First**, let's consider a hidden-to-output weights, w_{kj}

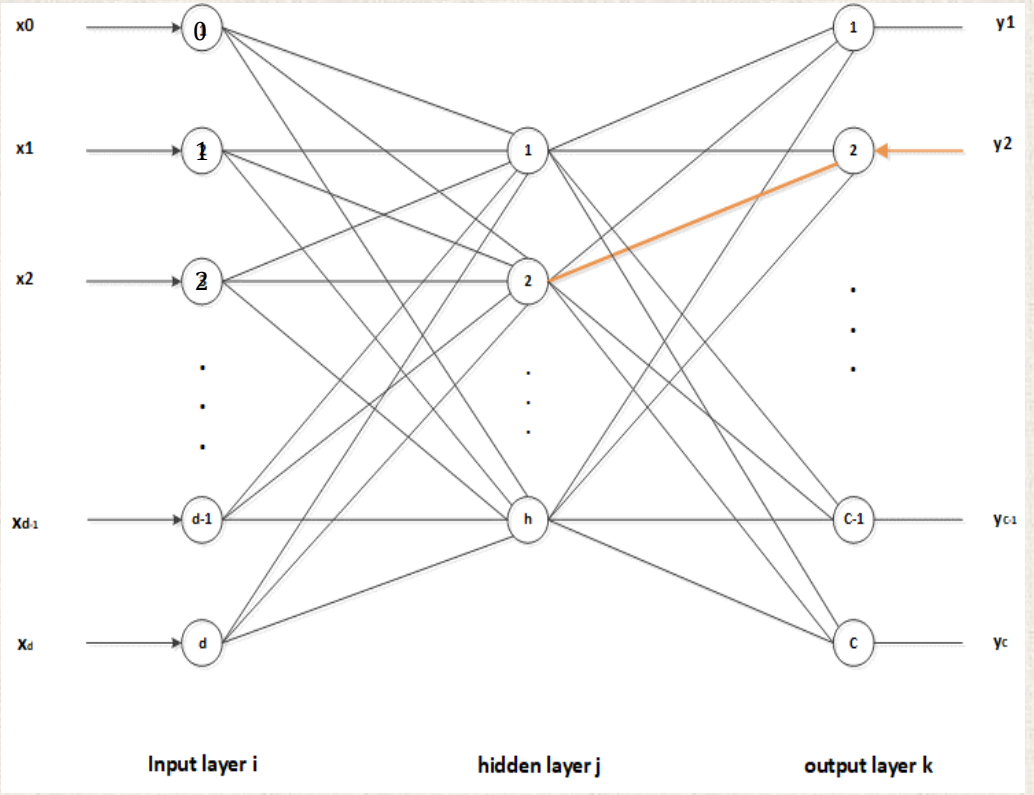
$$\frac{\partial E_n}{\partial w_{kj}} = \frac{\partial E_n}{\partial y_k} \cdot \frac{\partial y_k}{\partial net_k} \cdot \frac{\partial net_k}{\partial w_{kj}}$$

Where, we have,

$$\frac{\partial E_n}{\partial y_k} = y_k - t_k; \frac{\partial y_k}{\partial net_k} = f'(net_k); \frac{\partial net_k}{\partial w_{kj}} = z_j$$

Hence,

$$\frac{\partial E_n}{\partial w_{kj}} = (y_k - t_k) f'(net_k) z_j$$



$$\frac{\partial E_n}{\partial w_{kj}} = (y_k - t_k) f'(net_k) z_j$$

- Let's define the ***sensitivity of node k*** as:

$$\delta_k = -\frac{\partial E_n}{\partial net_k} = -\frac{\partial E_n}{\partial y_k} \cdot \frac{\partial y_k}{\partial net_k} = (t_k - y_k) f'(net_k)$$

Then,

$$\frac{\partial E_n}{\partial w_{kj}} = -\delta_k z_j$$

- ***Secondly,*** let's consider the input-to-hidden weights, w_{ji}

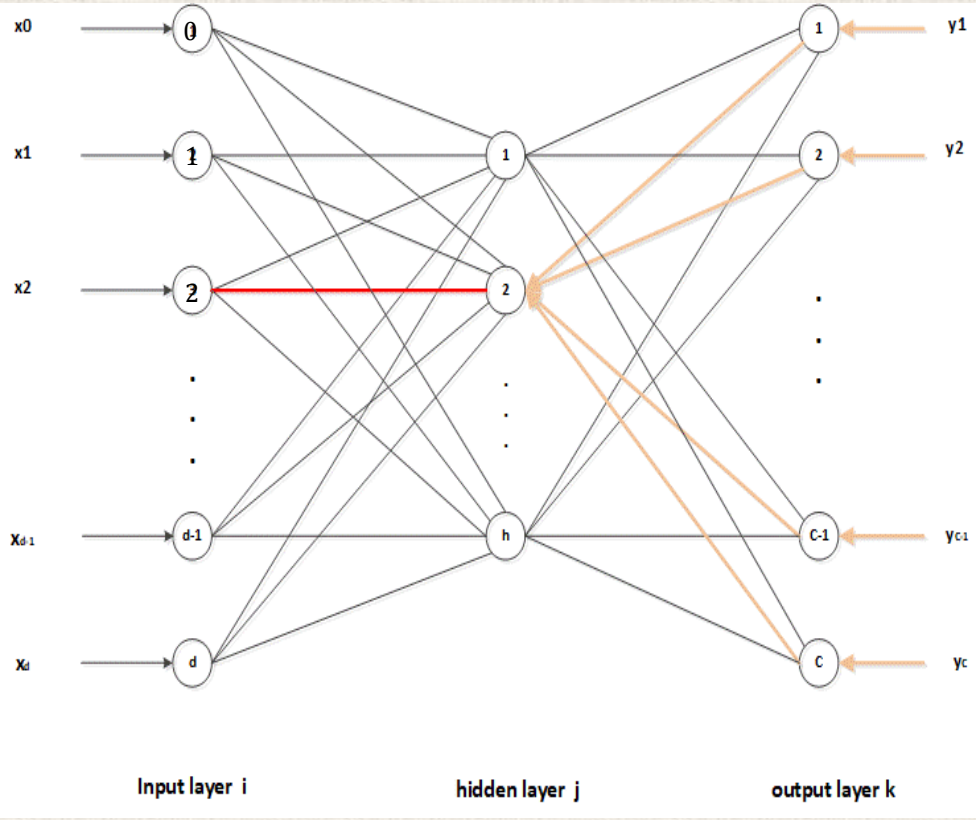
$$\frac{\partial E_n}{\partial w_{ji}} = \frac{\partial E_n}{\partial y_k} \cdot \frac{\partial y_k}{\partial net_k} \cdot \frac{\partial net_k}{\partial z_j} \cdot \frac{\partial z_j}{\partial net_j} \cdot \frac{\partial net_j}{\partial w_{ji}}$$

Where,

$$\frac{\partial E_n}{\partial y_k} = -\sum_{k=1}^{C} (t_k - y_k) \; ; \; \frac{\partial y_k}{\partial net_k} = f'(net_k)$$

$$\frac{\partial net_k}{\partial z_j} = w_{kj}$$

$$\frac{\partial z_j}{\partial net_j} = f'(net_j); \quad \frac{\partial net_j}{\partial w_{ji}} = x_i$$



Hence,

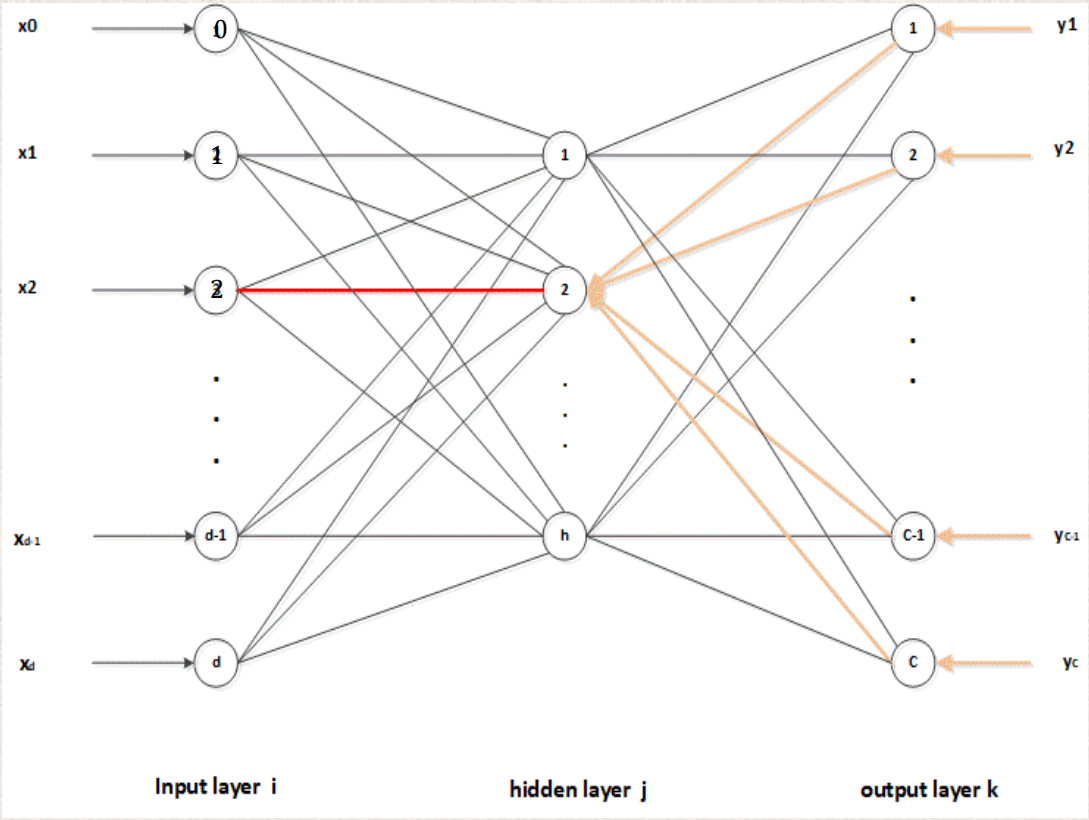
$$\frac{\partial E_n}{\partial w_{ji}} = \left\{ -\sum_{k=1}^{C} (t_k - y_k)\, f'(net_k) w_{kj} \right\} f'(net_j) x_i$$

$$= -\left(\sum_{k=1}^{C} w_{kj} \delta_k \right) f'(net_j) x_i = -\delta_j x_i$$

Where,

$$\delta_j = \left(\sum_{k=1}^{C} w_{kj} \delta_k \right) f'(net_j)$$

is the ***sensitivity for hidden node j.***



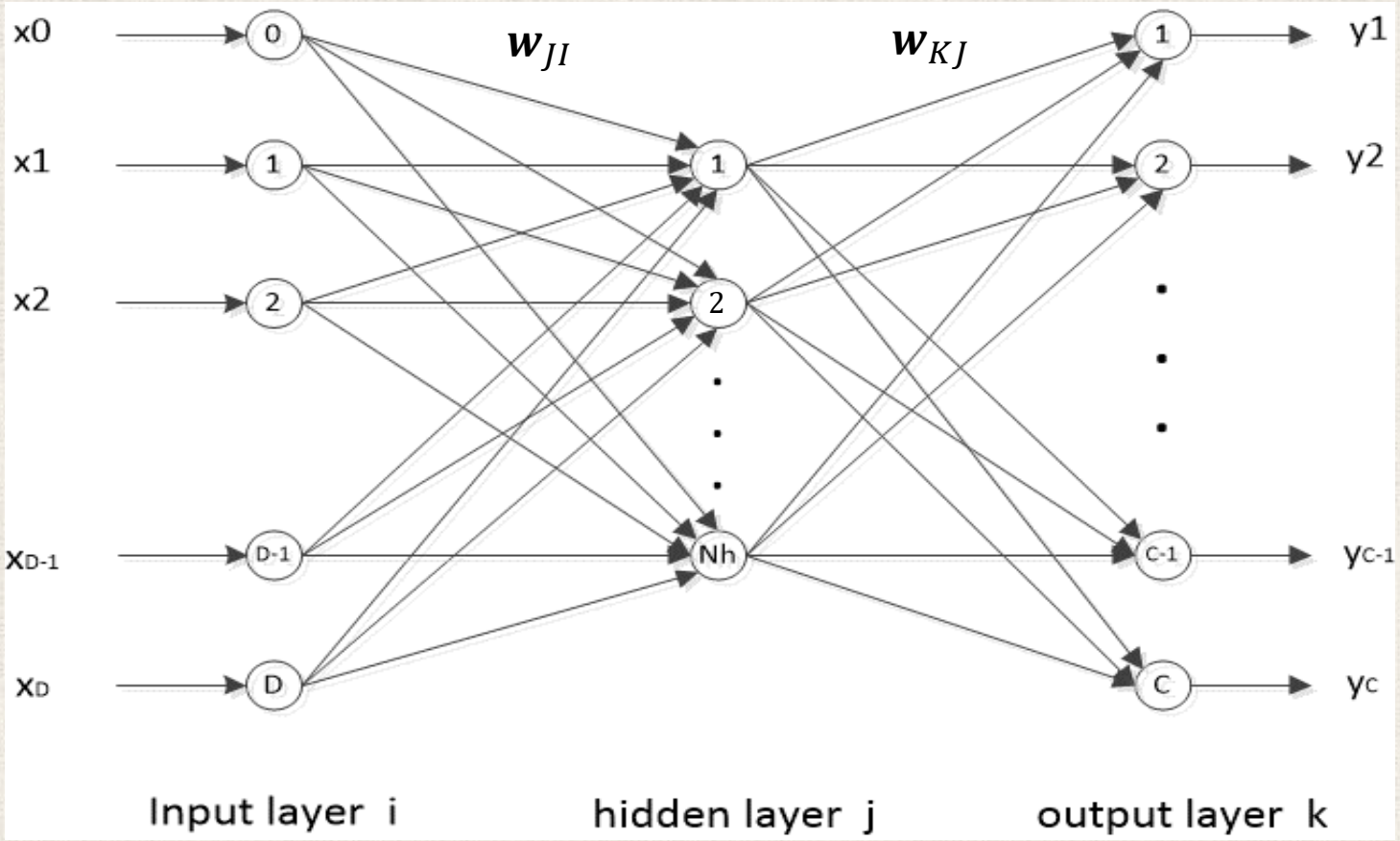
How the backpropagation algorithm works

Training samples

forward propagation

classification

physical



gradient calculation

virtual

Training errors

backward propagation

Artificial Neural Networks (ANN) – training Protocols

Stochastic training: data instances are chosen randomly from the training set and the network weights are updated for each data instance presented.

Algorithm: (***stochastic backpropagation***)

Begin

initialize $\boldsymbol{w}$, assign values to θ, η, let $m = 0$

do $m \leftarrow m + 1$

$\boldsymbol{x}_m \leftarrow randomly\ chosen\ pattern, feed\ \boldsymbol{x}_m\ to\ the\ network$

$w_{kj} \leftarrow w_{kj} + \eta\delta_k z_j;\ \ w_{ji} \leftarrow w_{ji} + \eta\delta_j x_i$

until $|\Delta E(\boldsymbol{w})| < \theta$

return $\boldsymbol{w}$

End

Batch training: All patterns in a batch are presented to the network before weights update take place

Algorithm: (***batch backpropagation***)

Begin

initialize $\boldsymbol{w}$, assign values to N, θ, η, let $r = 0$

do $r \leftarrow r + 1$ (epoch)

$m \leftarrow 0;\ \Delta w_{ji} \leftarrow 0;\ \Delta w_{kj} \leftarrow 0$

do $m \leftarrow m + 1$

$\boldsymbol{x}_m \leftarrow selected\ pattern$, *feed* $\boldsymbol{x}_m$ *to the network*

$\Delta w_{kj} \leftarrow \Delta w_{kj} + \eta \delta_k z_j;\ \ \Delta w_{ji} \leftarrow \Delta w_{ji} + \eta \delta_j x_m$

until $m == N$

$w_{kj} = w_{kj} + \Delta w_{kj};\ \ w_{ji} = w_{ji} + \Delta w_{ji}$

until $\|\Delta E(\boldsymbol{w})\| < \theta$

return $\boldsymbol{w}$

End

Practical techniques for improving backpropagation training

1. Initializing weights:

- **The weights can not be initialized as all zero**, otherwise, training can not take place.
- We choose weights from a uniform distribution: $-\tilde{w} < w < \tilde{w}$
- Usually,

$$-\frac{1}{\sqrt{D}} < w_{ji} < \frac{1}{\sqrt{D}}; \quad -\frac{1}{\sqrt{N_h}} < w_{kj} < \frac{1}{\sqrt{N_h}}$$

Where, D is the dimension of the inputs and N_h is the number of hidden nodes.

2. ***Activation function:***

- Must be nonlinear function $f(x)$
- $f(x)$ saturates. It has some maximum and minimum output values to keep the weights and activations bounded
- Continuity and smoothness

3. ***Scaling input (normalization): (to be done separately for training set and test set)***

- ***Statistical normalization (standardization)***: For each feature x across the set of training data, the mean and standard deviation (μ, σ) are calculated. Then the transform of x to x' as:

$$x' = \frac{x - \mu}{\sigma}$$

Then, each feature across the data set will have zero mean and unit variance.

- ***Min-Max normalization***: A feature x is scaled to a range of 0 to 1 by:

$$x' = \frac{x - x_{min}}{x_{max} - x_{min}}$$

Where, $x_{max}\ and\ x_{min}$ are the maximum and minimum values in x.

3. ***Scaling input (normalization):***

- ***Median normalization***:

$$x' = \frac{x}{median(x)}$$

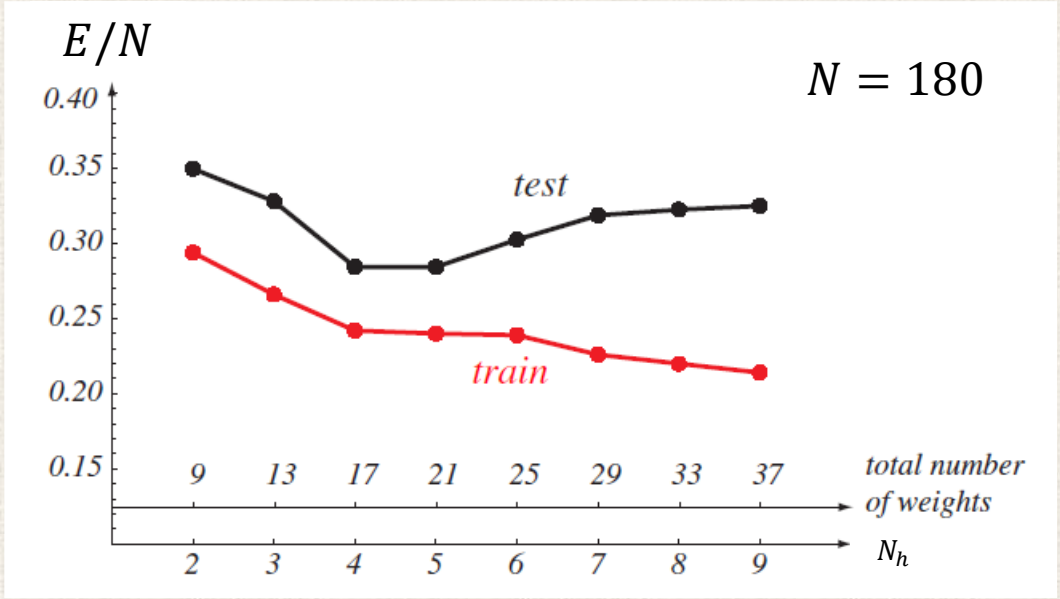
- ***Sigmoid normalization***:

$$x' = \frac{e^{x} - e^{-x}}{e^{x} + e^{-x}}$$

4. ***Number of hidden nodes:***

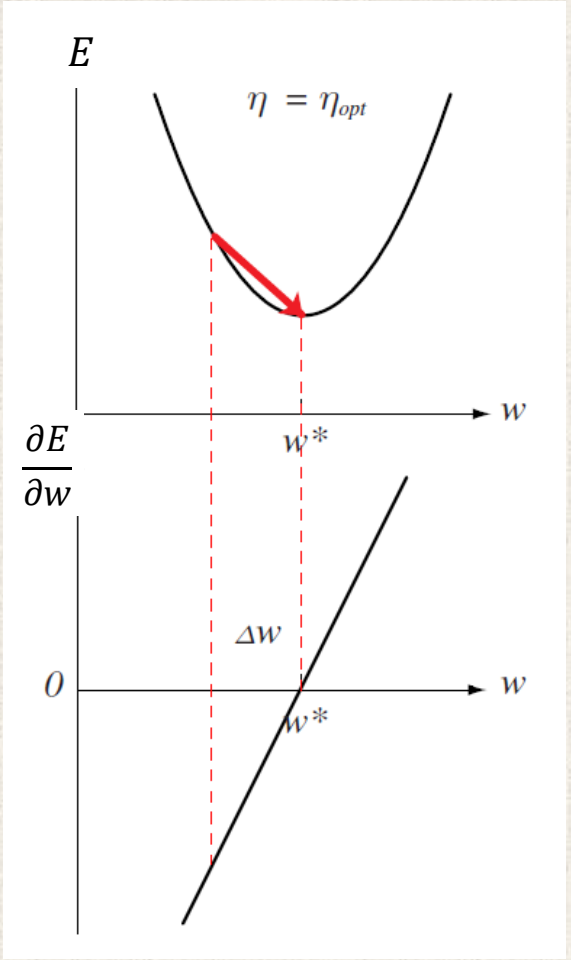
- The number of hidden nodes, N_h, governs the *expressive power* of the net and thus the *complexity of the decision boundary*.
  - If the patterns are linearly separable, then few hidden nodes are needed.
- ***Higher number of hidden nodes means smaller training error, but harder generalization (overfitting can happen)***
- ***Rule of thumb:***

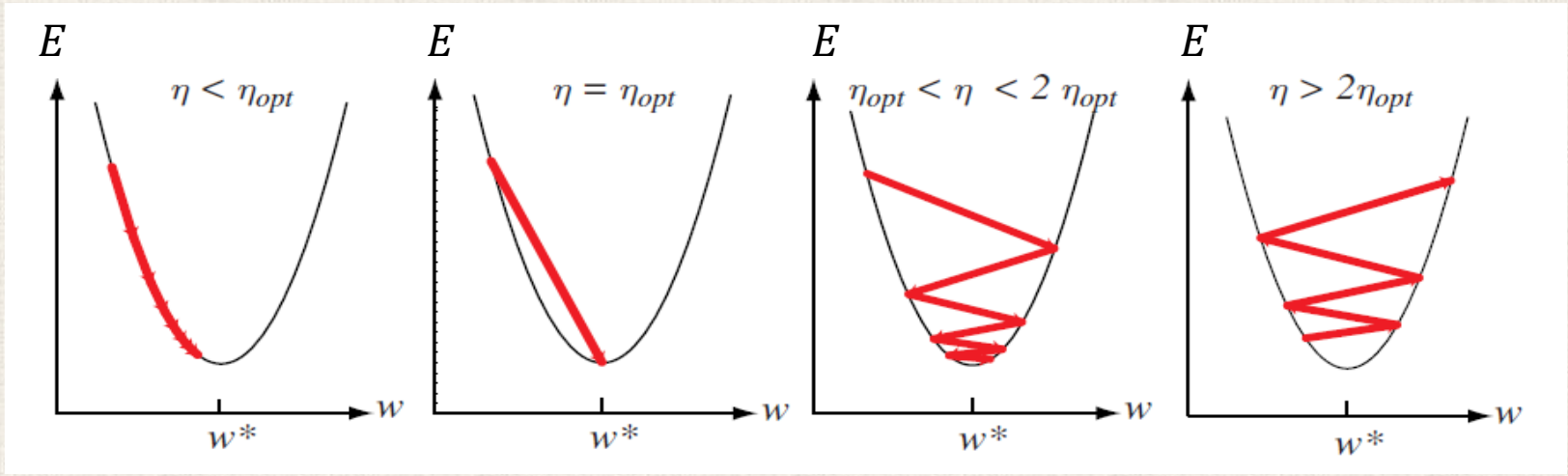
  Choose N_h such that the total number of weights in the network is roughly $\frac{N}{10}$, where N is the number of training instances in the training data set.



5. ***Learning rate:***

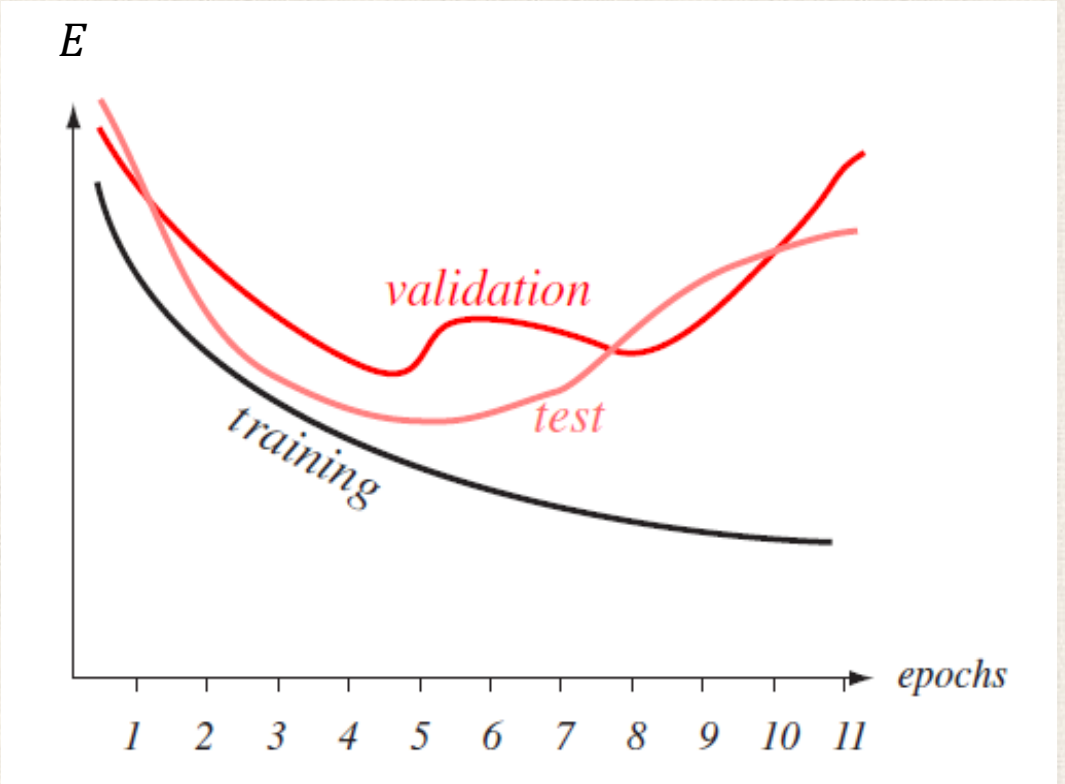
- As long as the learning rate is small enough, it only affect the training time, not the final weight.
- The optimal learning rate is the one that leads to the local error minimum in one learning step.
- In practice, a learning rate of $\eta \cong 0.1$ is often adequate as a first choice.
- The learning rate should be lowered if the criterion function diverges during training, or instead should be raised if training seems unduly slow.



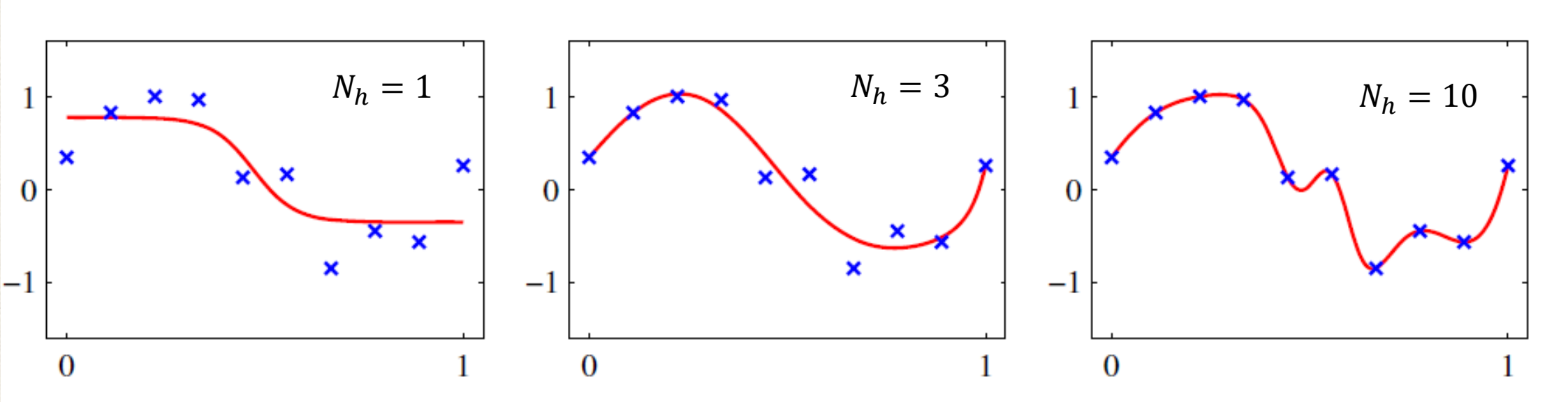


6. ***Overtraining problem***:

- For the Perceptron model, which is a two-layer network, we can train as long as we like without fear that it would degrade final recognition accuracy because the complexity of the decision boundary is not changed (it is always a hyperplane).
- For three-layer networks, excessive training can lead to poor generalization as the net implements a complex decision boundary tuned to the specific training data rather than the general properties of the underlying distribution. This is called ***overtraining***.
- In practice, training should be stopped when the error on a separate validation set reaches a minimum to avoid overfitting.



Regularization in Neural Networks



A network trained on 10 data points drawn from a sinusoidal data set. The graphs show the results of fitting network with N_h = 1, 3, and 10 hidden nodes.

- Underfitting and overfitting can happen when using neural networks
- The complexity of the model can be adjusted by adding a regularization term to the error function:

$$\tilde{E}(\boldsymbol{w}) = E(\boldsymbol{w}) + \alpha \|\boldsymbol{w}\|_2^2$$