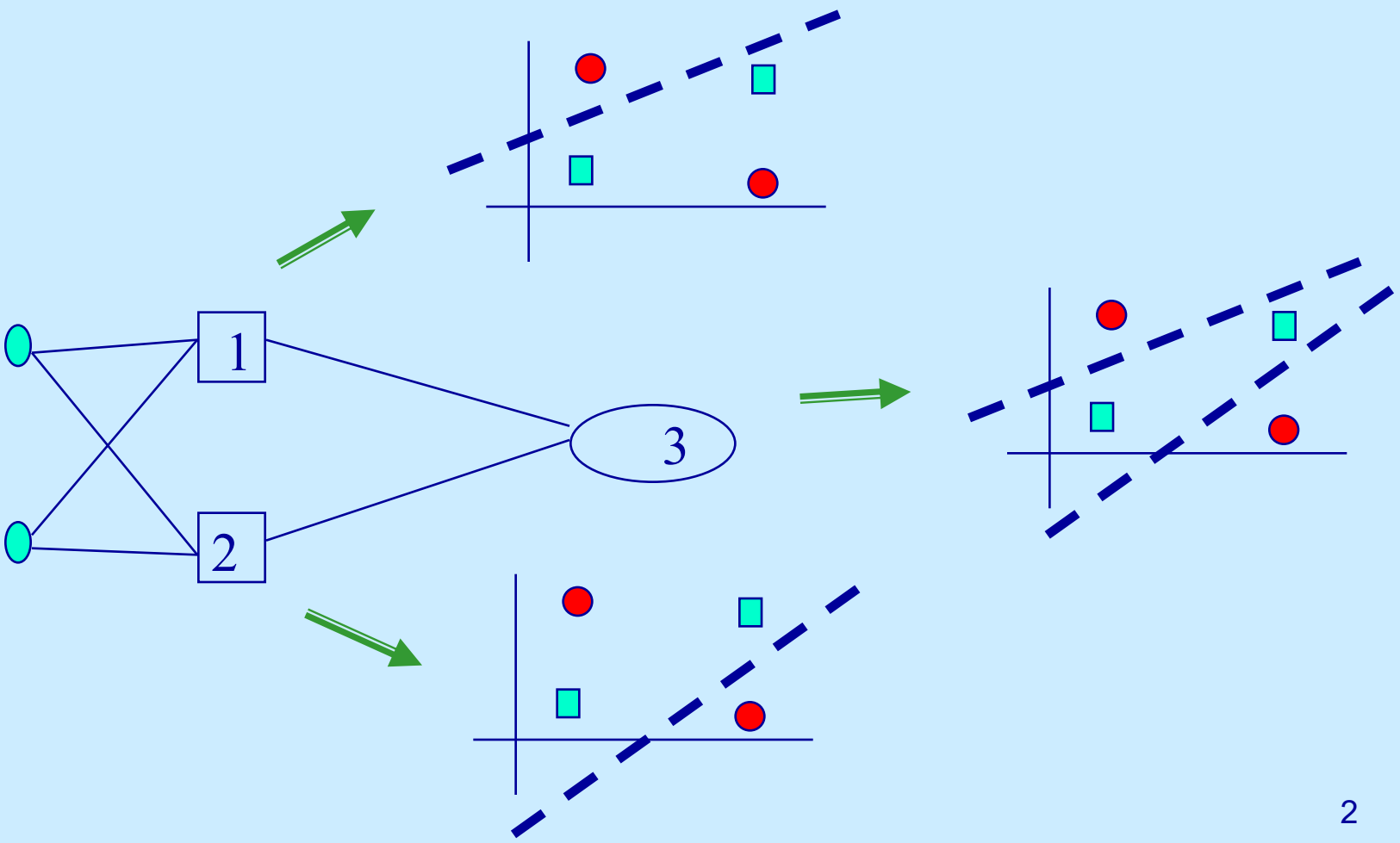
هوش محاسباتی:
شبکه های عصبی مصنوعی

دانشکده مهندسي کامپيوتر
دانشگاه علم و صنعت ایران
mozayani@iust.ac.ir

مدل پرسپترون چند لایه

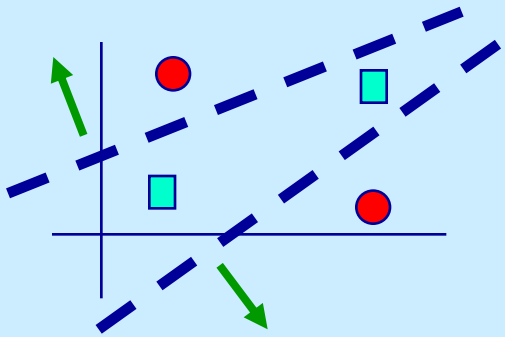
Solution to XOR problem

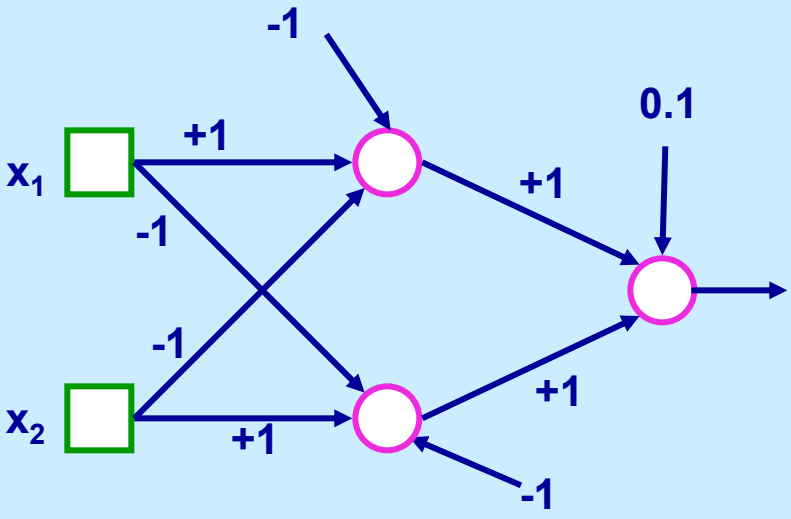
Minsky & Papert (1969) offered solution to XOR problem by combining perceptron unit responses using a second layer of units



XOR problem

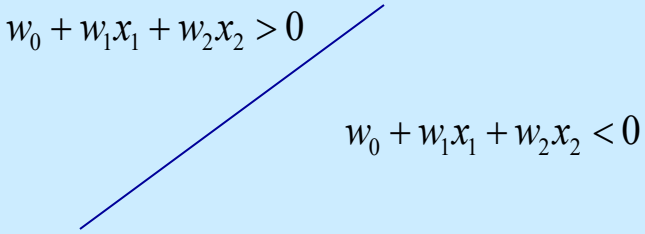
In this graph of the XOR, input pairs giving output equal to 1 and -1 are depicted with green and red points.
These two classes cannot be separated using a line.
We have to use two lines.
The following NN with two hidden nodes realizes this non-linear separation, where each hidden node is a perceptron and describes one of the two blue lines.

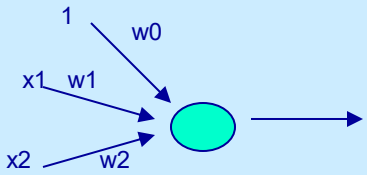




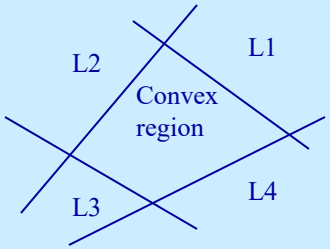
This NN uses the sign activation function. The two green arrows indicate the directions of the weight vectors of the two hidden nodes, (1,-1) and (-1,1). They indicate the regions where the network output will be 1. The output node is used to combine the outputs of the two hidden nodes.

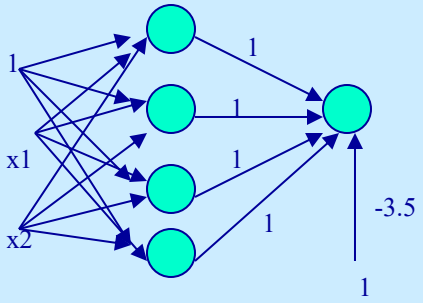
Types of decision regions



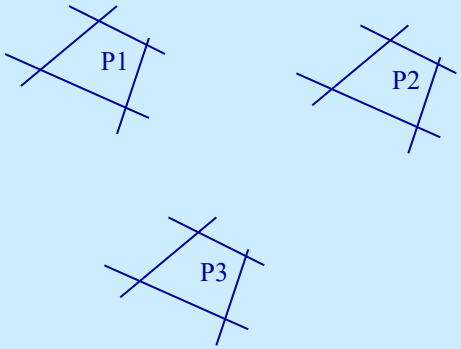


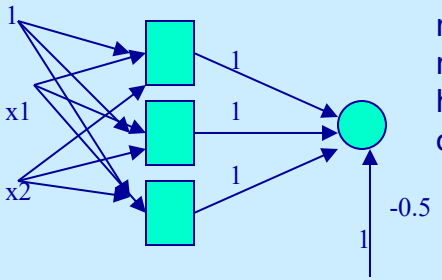
Network with a single node





One-hidden layer network that realizes the convex region: each hidden node realizes one of the lines bounding the convex region

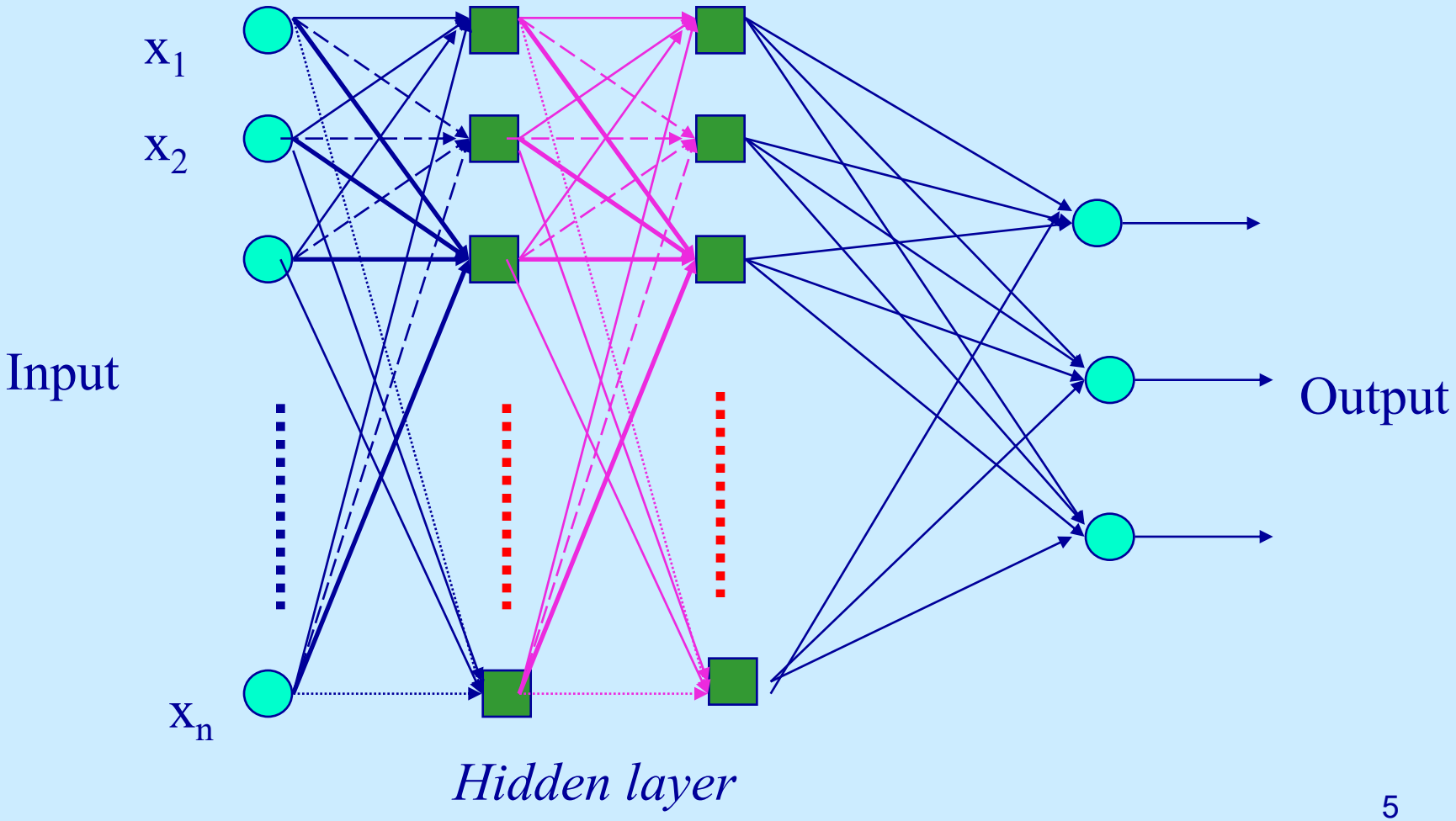




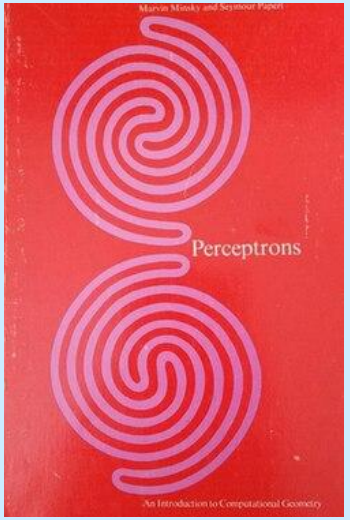
two-hidden layer network that realizes the union of three convex regions: each box represents a one hidden layer network realizing one convex region

Multi-layer perceptron

A four-layer network:



MLP problem



- Minsky and Papert (1969)
- Discusses two-layer perceptrons with threshold logic units
- *"...our intuitive judgement [is] that the extension [to MLPs with hidden layers] is sterile."*
- Drove wedge between numerics and symbolics researchers
- Neural network funding dried up

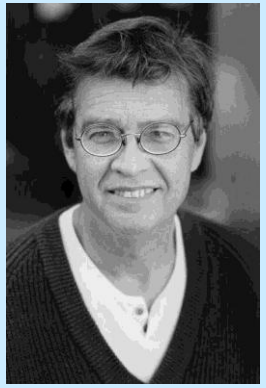
Training: Backpropagation

- Main problem:

  the error for hidden neurons is not known !

- 1985 : Yann Le Cun

- 1986 : Rumelhart, Hinton & Williams



David E. Rumelhart, 1942-2011

Training: Backpropagation

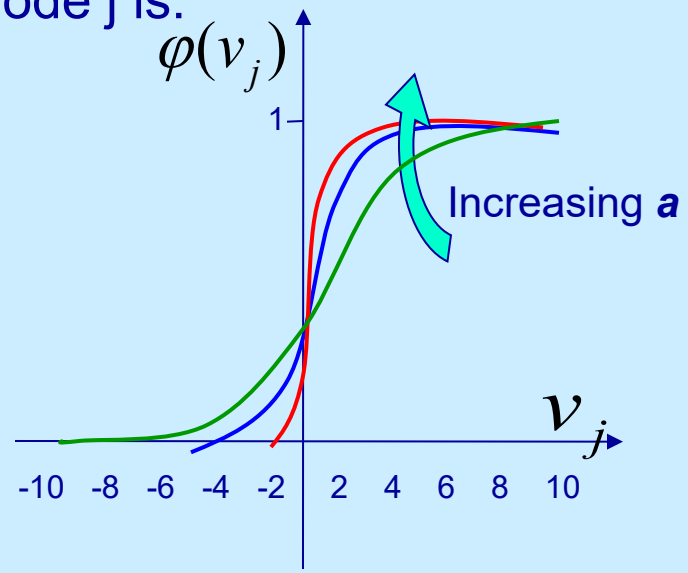
- The classical learning algorithm of MLP is based on the gradient descent method.
- For this reason the activation function are continuous functions of the weights, differentiable everywhere.
- A typical activation function that can be viewed as a continuous approximation of the step (threshold) function is the Sigmoid Function. The activation function for node j is:

$$\varphi(\mathrm{v_j}) = \frac{1}{1+e^{-av_j}} \quad \text{with } a > 0$$

where $\mathrm{v_j} = \sum_i w_{\mathrm{ji}} y_{\mathrm{i}}$

with w_{ji} weight of link from node i

to node j and y_{i} output of node i



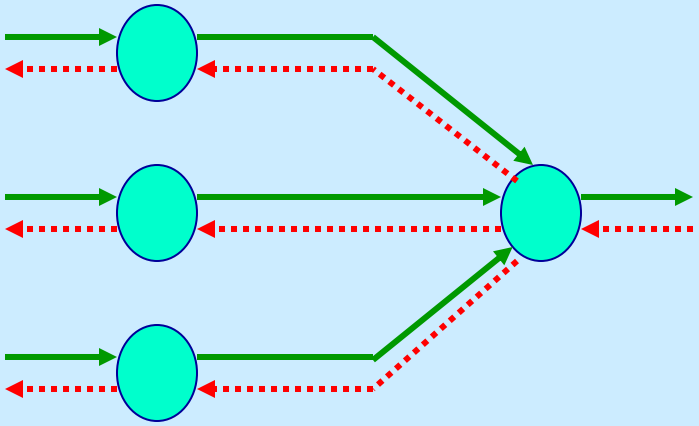
- when a tends to infinity then φ becomes the step function

Training: Backprop algorithm

- The BP algorithm searches for weight values that minimize the total error of the network over the set of training set.
- BP consists of the repeated application of the following two passes:
  - **Forward pass**: in this step the network is activated on one example and the error of (each neuron of) the output layer is computed.
  - **Backward pass**: in this step the network error is used for updating the weights (credit assignment problem). This process is more complex than the LMS algorithm for Adaline, because hidden nodes are linked to the error not directly but by means of the nodes of the next layer. Therefore, starting at the output layer, the error is propagated backwards through the network, layer by layer. This is done by recursively computing the local gradient of each neuron.

BP

- Back-propagation training algorithm



Network activation
Forward Step

Error propagation
Backward Step

- BP adjusts the weights of the network in order to minimize the total mean squared error.

Total Mean Squared Error

- The error of output neuron *j* after the activation of the network on the n^{th} training example $(X_{(n)}, d_{(n)})$ is:

$$e_j(n) = d_j(n) - y_j(n)$$

- The network error is the sum of the squared errors of the output neurons:

$$E(n) = \frac{1}{2} \sum_{j \in \text{outputnode}} e_j^2(n)$$

- *The total mean squared error is the average of the network errors of the training examples.*

$$E_{AV} = \frac{1}{N} \sum_{n=1}^{N} E(n)$$

Weight Update Rule

The BP weight update rule is based on the gradient descent method:

Take a step in the direction yielding the maximum decrease of the network error E.

This direction is the opposite of the gradient of E.

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

$$\Delta \mathrm{w}_{\mathrm{ji}} = -\eta \frac{\partial E}{\partial \mathrm{w}_{\mathrm{ji}}}$$

Weight Update Rule

Input of neuron j is:

$$v_j = \sum_{i=0,\ldots,m} w_{ji} y_i$$

Using the chain rule we can write:

$$\frac{\partial E}{\partial w_{ji}} = \frac{\partial E}{\partial v_j} \frac{\partial v_j}{\partial w_{ji}}$$

(local gradient of neuron j) $= -\delta_j$ $\qquad$ y_i

Therefore:

$$\Delta w_{ji} = \eta \delta_j y_i$$

Weight update of output neuron

In order to compute the weight change Δw_{ji} we need to know the local gradient of neuron j δ_j
There are two cases, depending whether j is an output or a hidden neuron.
If j is an output neuron then using the chain rule we obtain:

$$-\frac{\partial E}{\partial \mathrm{v_j}} = -\frac{\partial E}{\partial \mathrm{e_j}} \frac{\partial \mathrm{e_j}}{\partial \mathrm{y_j}} \frac{\partial \mathrm{y_j}}{\partial \mathrm{v_j}} = -\mathrm{e}_j(-1)\varphi'(\mathrm{v_j})$$

Knowing: $\mathrm{e_j} = \mathrm{d_j} - \mathrm{y_j}$ and $y_j = \varphi(\mathrm{v_j})$ and $\mathrm{E(n)} = \frac{1}{2}\sum \mathrm{e_j^2(n)}$

So **if j is an output node** then the weight w_{ji} from neuron i to neuron j is updated of:

$$\Delta w_{ji} = \eta(\mathrm{d_j} - \mathrm{y_j})\varphi'(\mathrm{v}_j)y_i$$

Weight update of hidden neuron

If j is a hidden neuron then its local gradient δ_j is computed using the local gradients of all the neurons of the next layer.

Using the chain rule we have: $\delta_j = -\frac{\partial E}{\partial v_j} = -\frac{\partial E}{\partial y_j}\frac{\partial y_j}{\partial v_j}$

$$-\frac{\partial E}{\partial y_j} = -\sum_{k\in C} e_k \frac{\partial e_k}{\partial y_j} = \sum_{k\in C} e_k \left[\frac{-\partial e_k}{\partial v_k}\right]\frac{\partial v_k}{\partial y_j}$$

Knowing: $-\frac{\partial e_k}{\partial v_k} = \varphi'(v_k), \qquad e_k\varphi'(v_k) = \delta_k, \qquad \frac{\partial v_k}{\partial y_j} = w_{kj},$

Then $-\frac{\partial E}{\partial y_j} = \sum_{k \text{ in next layer}} \delta_k w_{kj}$

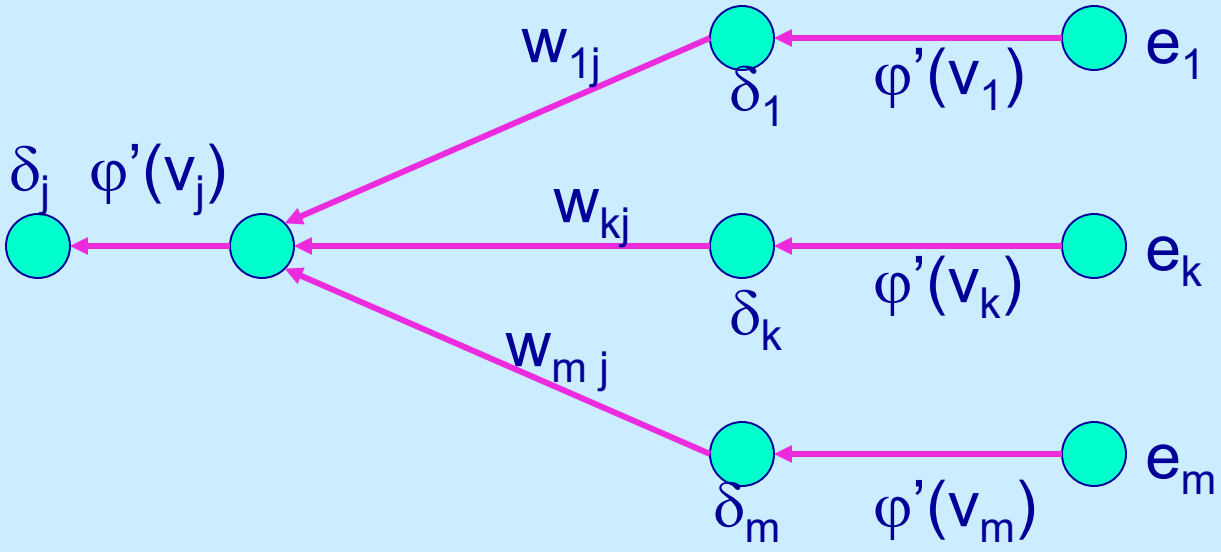
Moreover $\frac{\partial y_j}{\partial v_j} = \varphi'(v_j)$

So **if j is a hidden node** then the weight w_{ji} from neuron i to neuron j is updated of:

$$\Delta w_{ji} = \eta y_i \varphi'(v_j) \sum_{k \text{ in next layer}} \delta_k w_{kj}$$

Error backpropagation

The flow-graph below illustrates how errors are back-propagated to hidden neuron ***j***



Summary: Delta Rule

- **Delta rule** $\Delta w_{ji} = \eta \delta_j y_i$

$$\delta_j = \begin{cases} \varphi'(v_j)(d_j - y_j) & \text{IF j output node} \\ \varphi'(v_j) \sum_{k \text{ of next layer}} \delta_k w_{kj} & \text{IF j hidden node} \end{cases}$$

Where $\varphi'(v_j)$ is the derivative of activation function

UNIVERSAL APPROXIMATION THEORY

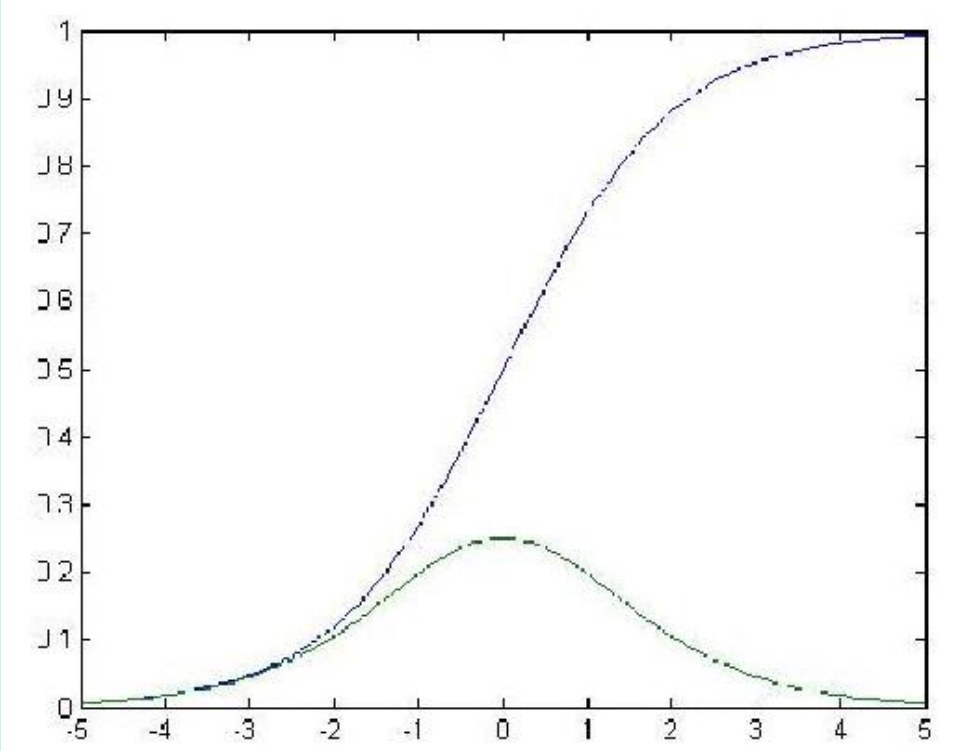
- $\Phi(.)$ a nonconstant, bounded, monotone-increasing continuous function
- I_{n0} denote the n0-dimensional unit hypercube $[0,1]^{n0}$
- given any function $f \in C(I_{n0})$ (space of continuous functions on I_{n0}) and $\varepsilon>0$, there exist an integer n1 and sets of real constants μ^i and b_i, and w_{ij}, where i=1;2;...;n1 and j=1;2;...;n0 such that we may define:

$$F(x_1, x_2, \ldots, x_{n_0}) = \sum_{i=1}^{n_1} \mu_i \phi \left(\sum_{j=1}^{n_0} w_{ij} x_j + b_i \right)$$

As an approximate realization of the function f(.); that is:

$$\left| F(x_1, x_2, \ldots, x_{n_0}) - f(x_1, x_2, \ldots, x_{n_0}) \right| < \varepsilon$$

Logistic activation function and its derivative



Since the amount of change in a given weight is proportional to this derivative, weights will be changed most for those units that are near their midrange and, in some sense, not yet committed to being either on or off.

Weight decay

it sometimes happens that hidden units have strong learned input weights that 'pin' their activation against 1 or 0, and in that case it becomes effectively impossible to propagate error back through these units.

Different solutions to this problem have been proposed:

One is to use a small amount of weight decay to prevent weights from growing too large.

Another is to add a small constant to the derivative of the activation function of the hidden unit.

η adaptation

- If η is small then the algorithm learns the weights very slowly, while if η is large then the large changes of the weights may cause an unstable behavior with oscillations of the weight values.

- A heuristics for accelerating the convergence of the back-prop algorithm through η adaptation:
  - Heuristic 1: Every weight has its own η.
  - Heuristic 2: Every η is allowed to vary from one iteration to the next.

Generalized delta rule

- A technique for tackling the problem of η is the introduction of **a momentum term** in the delta rule which takes into account previous updates. We obtain the following **generalized Delta rule** (Rumelhart method):

$$\Delta w_{ji}(n) = \alpha \Delta w_{ji}(n-1) + \eta \delta_j(n) y_i(n)$$

α is the momentum constant $\quad 0 \le \alpha < 1$

the momentum accelerates the descent in steady downhill directions.
the momentum has a stabilizing effect in directions that oscillate in time.

BP algorithm (incremental-mode)

n=1;

initialize **w**(n) randomly;

while (stopping criterion not satisfied and n<max_iterations)

for each example (**x**,*d*)

- run the network with input x and compute the output y
- update the weights in backward order starting from those of the output layer:

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

with Δw_{ji} computed using the (generalized) Delta rule

end-for

n = n+1;

end-while;

BP algorithm (Batch mode)

- In the batch-mode the weights are updated only after all examples have been processed, using the formula

$$w_{ji} = w_{ji} + \sum_{x \text{ training example}} \Delta w_{ji}^{x}$$

-
- In the incremental mode from one epoch to the next choose a randomized ordering for selecting the examples in the training set in order to avoid poor performance.

Stopping criterions

- Sensible stopping criterions:
  - total mean squared error change:

    BP is considered to have converged when the absolute rate of change in the average squared error per epoch is sufficiently small (in the range [0.1, 0.01]).
  - generalization based criterion:

    After each epoch the MLP is tested for generalization. If the generalization performance is adequate then stop. If this stopping criterion is used then the part of the training set used for testing the network generalization will not used for updating the weights.

MLP Network Design

- Data representation
- Network Topology
- Network Parameters
- Training
- Validation

Data Representation

- Data representation depends on the problem. In general MLPs work on continuous (real valued) inputs. So symbolic data are encoded into continuous ones.

- Inputs of different types may have different ranges of values which affect the training process. Normalization may be used, like the following one which scales each inputs to assume values between 0 and 1.

$$x_i = \frac{x_i - \min_i}{\max_i - \min_i}$$

Network Topology

- The number of layers and of neurons depend on the application. In practice this issue is solved by trial and error.

- Two types of adaptive algorithms can be used:
  - Network Pruning: start from a large network and successively remove some neurons and links until network performance degrades.
  - Network Growing: begin with a small network and introduce new neurons until performance is satisfactory.

Network parameters

- How are the weights initialized?
- How is the learning rate chosen?
- How many hidden layers and how many neurons?
- How many examples in the training set?

Initialization of weights

- In general, initial weights are randomly chosen, with typical values between -1.0 and 1.0 or -0.5 and 0.5.

- If some inputs are much larger than others, random initialization may bias the network to give much more importance to larger inputs. In such a case, weights can be initialized as follows:
  - input (first) layer:

$$w_{ji} = \pm \frac{1}{2N} \sum_{i=1,\ldots,N} \frac{1}{|x_i|}$$

  - Other layers:

$$w_{kj} = \pm \frac{1}{2N} \sum_{i=1,\ldots,N} \frac{1}{\varphi(\sum w_{ji} x_i)}$$

learning rate

- The right value of η *depends on the application*. Values between 0.1 and 0.9 have been used in many applications.
- Other heuristics adapt η during the training as described in previous slides.

Adaptive learning rate

The learning rate parameter changes as the algorithm progresses

- Bold Driver Method
  - modifies the learning rate based on change in E
    - If E increases, then learning rate is reduced
    - If E decreases, then learning rate is increased
- Self-adaptive
  - Each weight has its own learning parameter
    - increase, if the sign of δ doesn't change
    - decrease, if the sign of δ changes

Training

- Rule of thumb:
  - the number of training examples should be at least five to ten times the number of weights of the network.
- Other rule:

$$N > \frac{|W|}{(1 - a)}$$

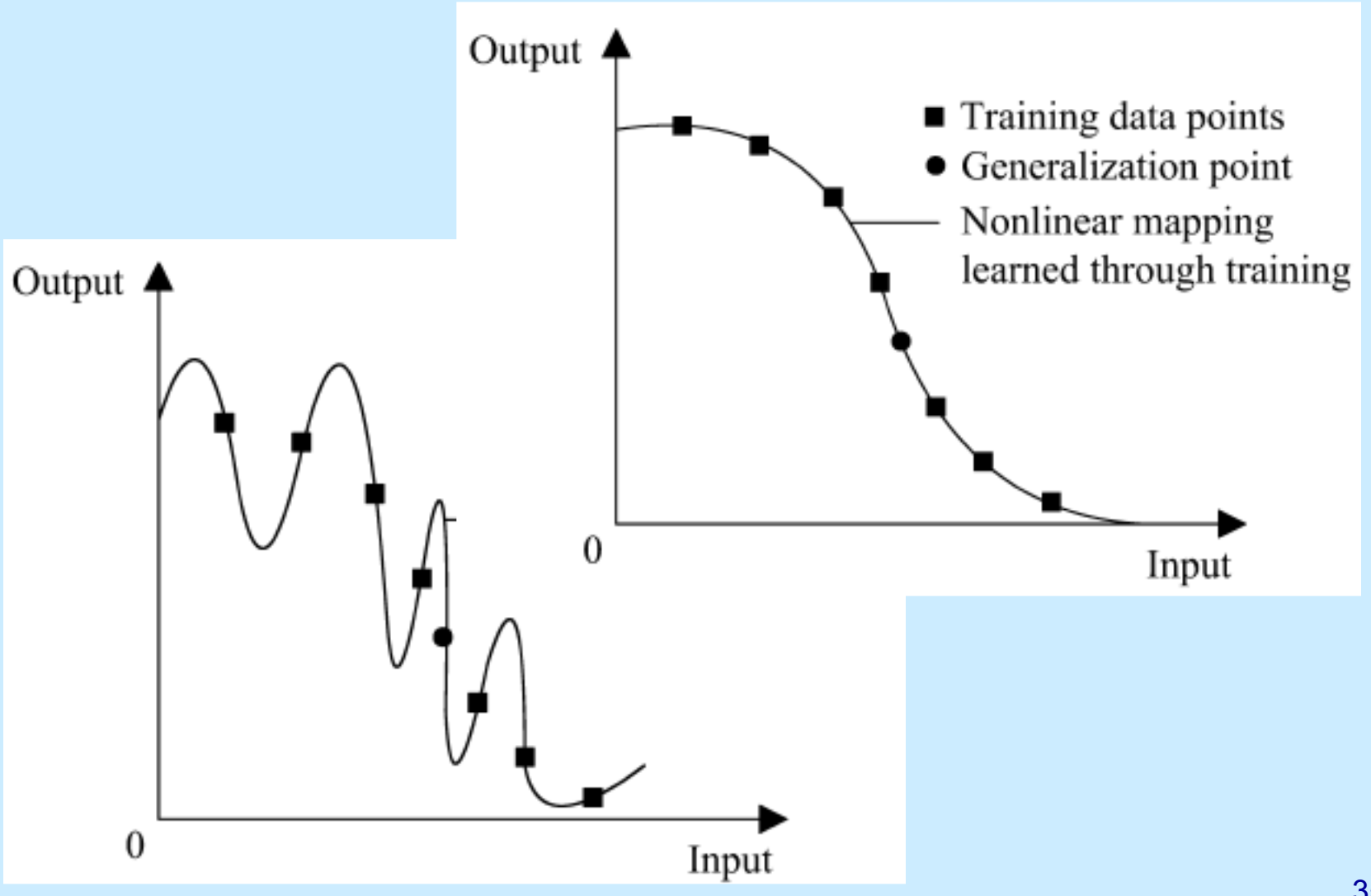
|W|= number of weights
a=expected accuracy on test set

Over-fitting

- **Over-fitting** = tuning to fit the idiosyncrasies of the training examples that are **not representative** of the general data.

- An ANN's ability to learn its training data is proportional to the number of hidden neurons, but so is overfitting!

Over-fitting



Over-fitting

Remedies:

a) Start with just a few neurons in the hidden layer
b) Use a test set as well as a training set
c) Decay weights by small fraction after each iteration

- Keep the set of best weights so far (on the training set), and the current set of weights. Stop training when the test set error goes up a lot. Use the best weight set as the result.

Over-fitting

- Over-fitting is a major problem when data sets are small. If small, then can split the total data into k different Distributions of training and test sets. Take the average over the k distributions.

- OR split the m test examples into k parts, and average the test error values (using the k-1 parts as part of the training set).

Some remarks

- Often, the more weights in the network, the easier to slip out of a local minimum via another dimension (weight), but local minima can cause difficulties (theory here is weak).

- Can train multiple nets with the same data, by initializing with different small weights. Choose the network with the best test set performance. Or, take the average output over several networks ("committee" network members).

Applicability of MLP

- Every boolean function can be represented by a network with a single hidden layer
- but it might require exponential (in the number of inputs) hidden neurons.

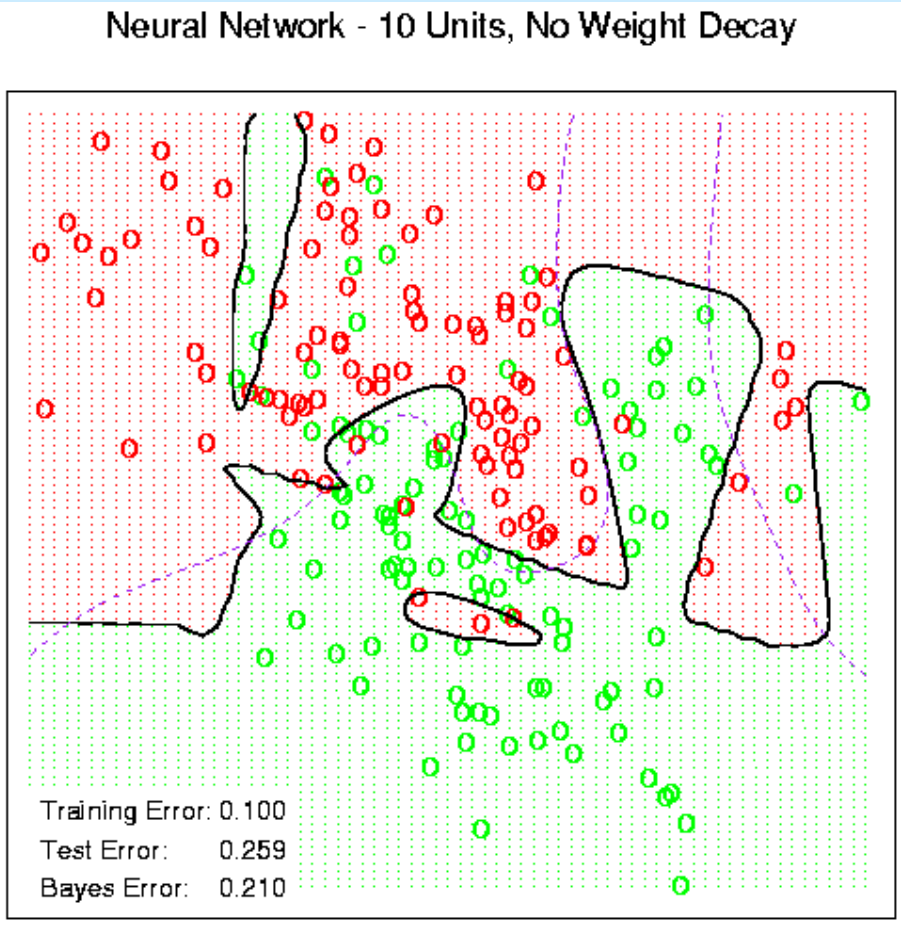
- Every bounded piece-wise continuous function can be approximated with arbitrarily small error by a network with one hidden layer.
- Any continuous function can be approximated to arbitrary accuracy by a network with two hidden layers.

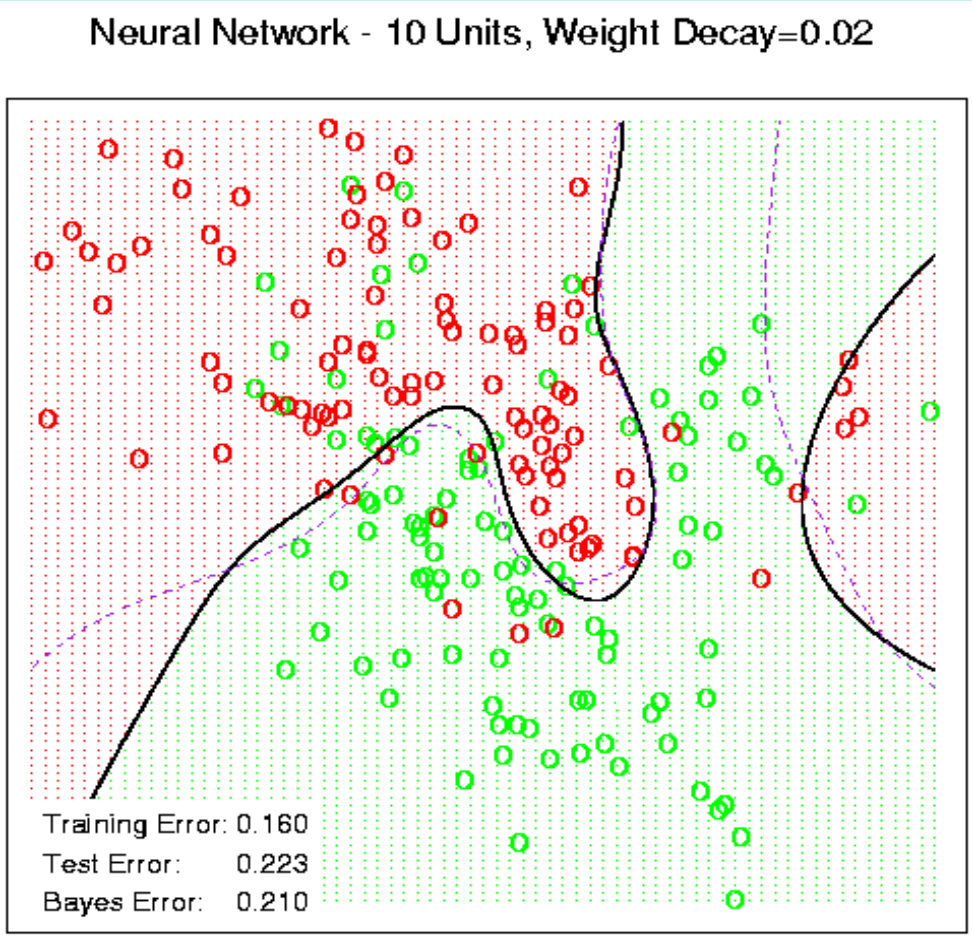
Applications of MLP

- Classification
- Function approximation

- ➔ Generalization

Example


Neural Network - 10 Units, No Weight Decay
Training Error: 0.100
Test Error: 0.259
Bayes Error: 0.210


Neural Network - 10 Units, Weight Decay=0.02
Training Error: 0.160
Test Error: 0.223
Bayes Error: 0.210

Hidden layer interpretation

- A hidden layer can give information about the nature of the input data, e.g. send in an 8 digit binary signal with 8 input neurons, where 7 are 0, one is 1, and the targets are the same as the inputs. Then the 3 neuron hidden layer outputs take (nearly) the binary code (001, 010, 011..) for the 8 possible inputs. Hidden layers can discover useful data representations!

- More layers, more complex representations!