Neural Networks

- An Artificial Neural Network is specified by:

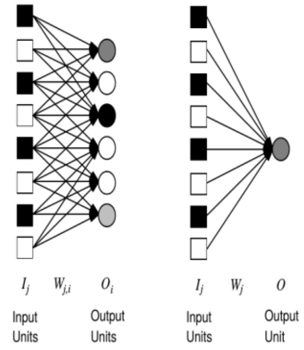
1. **Neuron Model:** The processing unit of the ANN which performs a linear combination of inputs.

2. **Architecture:** Just like the set of neurons in brain. The neurons are connected by links which have weight.

2. **Learning Algorithm:** Modifies the weight of links to model a specific task. The training relies heavily on the data fed to the neurons.

Perceptron

- Developed by Frank Rosenblatt in 1950-1960 inspired by McCulloh and Pitts.
- The initial models were able to classify linearly separable functions but failed to so in case of a non-linear decision boundary.
- Multi-layer perceptron – found as a “solution” to represent nonlinearly separable functions – 1950s.
- The perceptron algorithm couldn’t converge when there were multiple local optima.
- Throughout the 1950s it was believed that there exists no algorithm for multi-layer perceptron.
- Perceptron convergence theorem Rosenblatt 1962: Perceptron will learn to classify any linearly separable set of inputs.



Perceptron Network   Single Perceptron

Perceptrons and Neural Networks, Manuela Veloso

Neuron

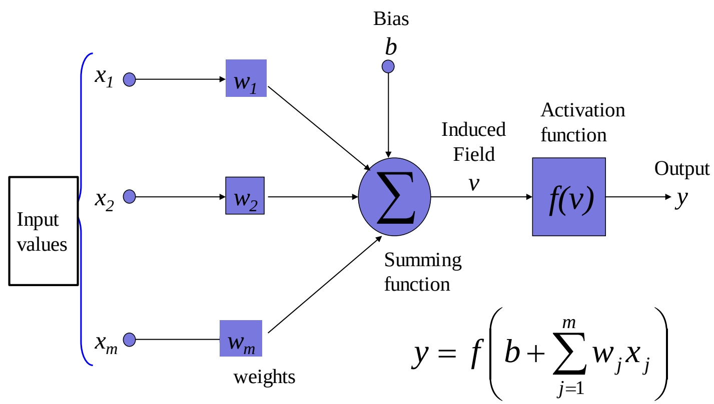
- The neuron provides a linear combination of the input provided to it and the applies a non-linear activation function to it.
- The weights of the links are represented as w_j and the inputs as x_j for the j^{th} input and neuron.

$$u = \sum_{j=1}^{m} w_j x_j$$

- Activation function: $y = f(u + b)$

  ‘b’ represents bias.

The Neuron Diagram



$$y = f\left(b + \sum_{j=1}^{m} w_j x_j\right)$$

Single Neuron as a Network

- x1 and x2 are normalized attribute value of data.
- y is the output of the neuron i.e. the class label.
- x1 and x2 values multiplied by weight values w1 a w2 are input to the neuron

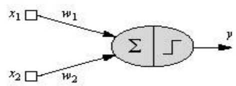


Fig1: an artificial neuron

- Given that
  - w1 = 0.25 and w2 = 0.75
  - Say value of x1 is 0.1 and value of x2 is 0.6,
  - So, weighted sum is :
  - Sum = w1 * x1 + w2 * x2 = 0.25 x 0.1 + 0.6 x 0.75 = 0.70

One Neuron as a Network

- The neuron receives the weighted sum as input and calculates the output as a function of input as follows :
- y = f(x) , where f(x) is defined as
- f(x) = 0 { when x< 0.5 }
- f(x) = 1 { when x >= 0.5 }
- For our example, weighted sum is 0.70, so y = 1 ,
- That means corresponding input attribute values are classified in class 1.
- If for another input values , sum = 0.45 , then f(x) = 0, so we could conclude that input values **are classified to class 0.**

Bias of a Neuron

- The bias ***b*** has the effect of applying a transformation to the weighted sum ***u***

  v* = *u* + *b
- The bias is an external parameter of the neuron. It can be modeled by adding an extra input.
- ***v*** is called **induced field** of the neuron

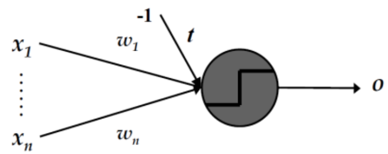
$$v = \sum_{j=0}^{m} w_j x_j$$

$$w_0 = b$$

Bias of a Neuron

- Really, the threshold t is just another weight (called the bias):

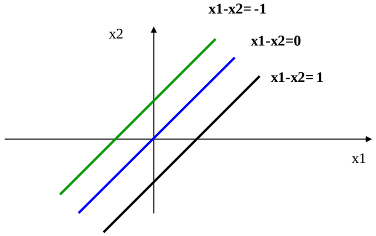
$$(w_1 \times x_1) + (w_2 \times x_2) + \ldots + (w_n \times x_n) \geq t$$
$$= (w_1 \times x_1) + (w_2 \times x_2) + \ldots + (w_n \times x_n) - t \geq 0$$
$$= (w_1 \times x_1) + (w_2 \times x_2) + \ldots + (w_n \times x_n) + (t \times \text{-}1) \geq 0$$



Bias of a Neuron : Geometric Interpretation

- The bias value added to the weighted sum $\sum_j w_j x_j$ so that we can transform it from the origin.

$v = \sum_j w_j x_j + b$, *here b is the bias*



Perceptron for Classification

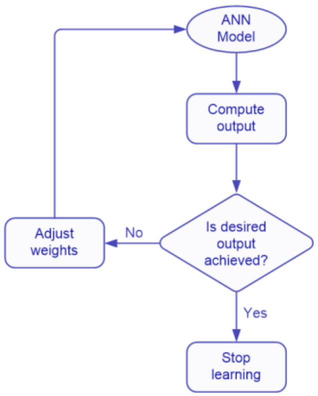
- The perceptron is used for binary classification.
- First train a perceptron for a classification task.
  - Find suitable weights in such a way that the training examples are correctly classified.
- *The perceptron* can only model *linearly separable classes.*
- Given training examples of classes C_1, C_2 train the perceptron in such a way that :
  - *If the output of the perceptron is +1 then the input is assigned to class C_1*
  - *If the output is -1 then the input is assigned to C_2*

Perceptron Training

Learning Procedure:

1. Randomly assign weights (between 0-1)
2. Present inputs from training data
3. Get output O, modify weights to gives results toward our desired output T
4. Repeat; stop when no errors, or enough epochs completed

A Supervised Learning Process



Three-step process:

1. Compute temporary outputs
2. Compare outputs with desired targets
3. Adjust the weights and repeat the process

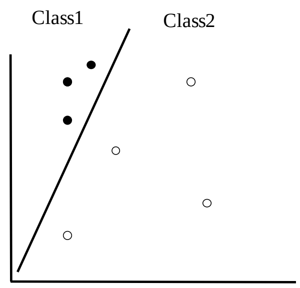
Perceptron Training

$\mathbf{w} \leftarrow \mathbf{0}$ (any initial values ok)
repeat
  for r=1 to R
    $\mathbf{w} \leftarrow \mathbf{w} + \eta(d^r - y^r)\mathbf{x}^r$
until no errors

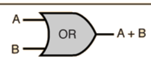
$\eta > 0$ is the learning rate
It can be taken to be 1 when inputs are 0 and 1

Perceptrons

- Essentially a linear discriminant
- Perceptron theorem: If a linear discriminant exists that can separate the classes without error, the training procedure is guaranteed to find that line or plane.



Implementing the OR Boolean Function



A	B	Out
0	0	0
0	1	1
1	0	1
1	1	1

Observation 1: The bias can't be positive.

Reason: From the diagram, the OR gate is 0 only if both inputs are 0. Hence according to the equation:

$= w_1x_1 + w_2x_2 + b$
$= w_1 * 0 + w_2 * 0 + b$
$= b$

Hence if b > 0 then the perceptron will classify this as 1 which is wrong.

NOTE: To classify it as class 0 we need the output as -1, let's set the bias b = -1

Implementing the OR Boolean Function

- Now with b = -1 let's check what the weights can be, for that let's check the 2nd row this time.
  $= w_1x_1 + w_2x_2 + b$
  $= w_1 * 0 + w_2 * 1 + -1$
  $= w_2 - 1 = 0$
  $\mathbf{w_2 = 1}$

- Similarly now let's check row 3 with new values
  $= w_1x_1 + w_2x_2 + b$
  $= w_1 * 1 + w_2 * 0 + -1$
  $= w_1 - 1 = 0$
  $\mathbf{w_1 = 1}$

So we have the values as w1= 1, w2 = 1 and b = -1

Let's see if this combination can classify the fourth row correctly

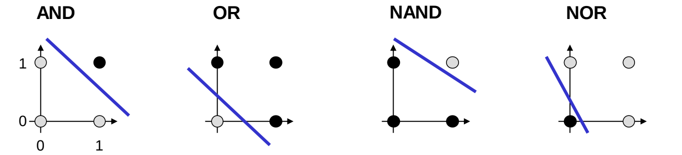
$= w_1x_1 + w_2x_2 + b$
$= 1 * 1 + 1 * 1 + -1$
$\mathbf{= 2 - 1 = 1}$

Convergence Theorem

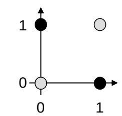
- Convergence theorem: For any linearly separable training data, the algorithm converges to a solution (as long as the learning rate is suitably small). But if the data is not linearly separable, the weights loop indefinitely.

Implementation of Boolean Functions

○ = 0 ● = 1



XOR



?

x_1	x_2	xor
0	0	0
0	1	1
1	0	1
1	1	0

$w_1 x_1 + w_2 x_2 \geq \theta$

) $0 < \theta$

) $w_2 \geq \theta$

) $w_1 \geq \theta$

) $w_1 + w_2 < \theta$

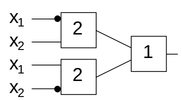
$\left.\begin{array}{l} \\ \\ \end{array}\right\}$ $w_1, w_2 \geq \theta > 0$

) $w_1 + w_2 \geq 2\theta$

contradiction!

Implementation of Boolean Functions: Leaving the dead end

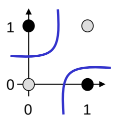
1. Multilayer Perceptrons:



) realizes XOR

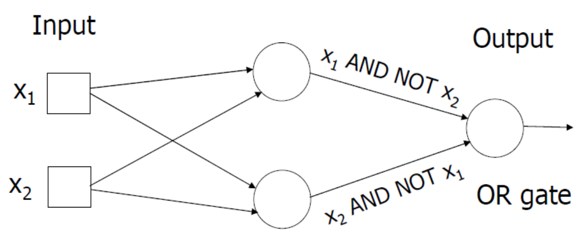
2. Nonlinear separating functions:

XOR $g(x_1, x_2) = 2x_1 + 2x_2 - 4x_1x_2 - 1$ with $\theta = 0$



$g(0,0) = -1$
$g(0,1) = +1$
$g(1,0) = +1$
$g(1,1) = -1$

Implementing XOR with simple perceptron units



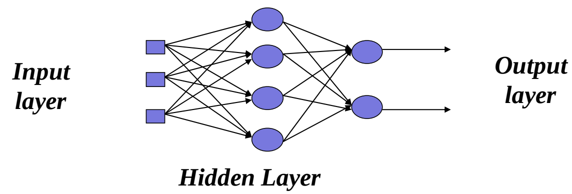
- Suffices to use one intermediate stage of simple perceptron units
- Approach generalizes to any Boolean function: write it in DNF, use one intermediate unit for each disjunct, then use an OR gate for output
- Proves that any Boolean function is realizable by a network of simple perceptron units

Different non linearly separable problems

Structure	*Types of Decision Regions*	*Exclusive-OR Problem*	*Classes with Meshed regions*	*Most General Region Shapes*
Single-Layer	*Half Plane Bounded By Hyperplane*	A B B A	B A	
Two-Layer	*Convex Open Or Closed Regions*	A B B A	B A	
Three-Layer	Abitrary (Complexity Limited by No. of Nodes)	A B B A	B A	

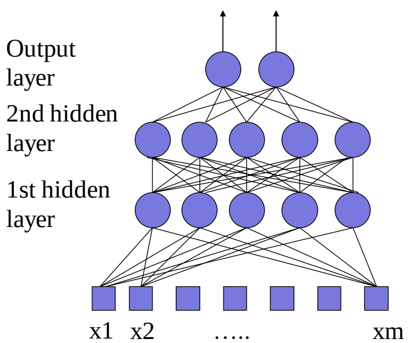
Multi layer feed-forward NN (FFNN)

- FFNN is a more general network architecture, where there are hidden layers between input and output layers.
- Hidden nodes do not directly receive inputs nor send outputs to the external environment.
- FFNNs overcome the limitation of single-layer NN.
- They can handle non-linearly separable learning tasks.



3-4-2 Network

Feed Forward Neural Networks

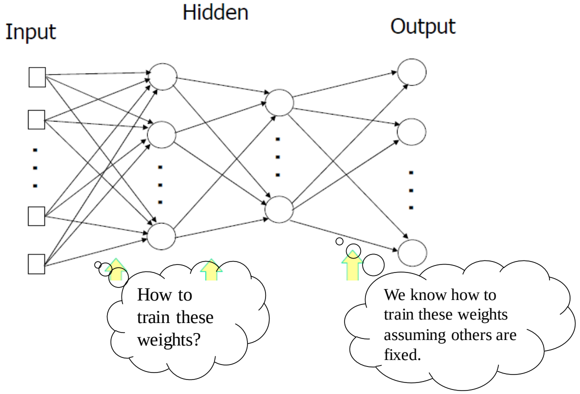


- The information is propagated from the inputs to the outputs
- Time has no role (NO cycle between outputs and inputs)

Hidden Layers

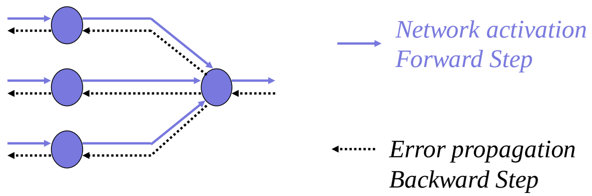
- In some cases, there may be many independencies among the input variables and adding an extra hidden layer can be helpful
- MLP with two hidden layers can approximate any non-continuous functions

Multilayer Networks



Backpropagation

- Back-propagation training algorithm



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

Backpropagation Algorithm

- The Backpropagation algorithm learns in the same way as single perceptron.
- It searches for weight values that minimize the total error of the network over the set of training examples (training set).

Given: set of input-output pairs
Task: compute weights for n-layer network to minimize the total error of the network

Backpropagation Algorithm

1.Determine the number of neurons required

2.Initialize weights to random values

3.Set activation values for threshold units

Backpropagation Algorithm

4. Choose an input-output pair and assign activation levels to input neurons

5. Propagate activations from input neurons to hidden layer neurons for each neuron
   $h_i = 1/(1 + e^{-\Sigma\, w1_{ij}x_j})$

6. Propagate activations from hidden layer neurons to output neurons for each neuron
   $o_k = 1/(1 + e^{-\Sigma\, w2_{ki}h_i})$

Backpropagation Algorithm

7. Compute error for output neurons by comparing pattern to actual
8. Compute error for neurons in hidden layer
9. Adjust weights in between hidden layer and output layer
10. Adjust weights between input layer and hidden layer
11. Go to step 4

Backprop learning algorithm (incremental-mode)

n=1;
initialize **weights** randomly;
while (stopping criterion not satisfied or n <max_iterations)
  for each example ($\boldsymbol{x}^r$)
  - 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;

Total Mean Squared Error

$$E[w] = \frac{1}{2}\sum(d_k^r - y_k^r)^2$$

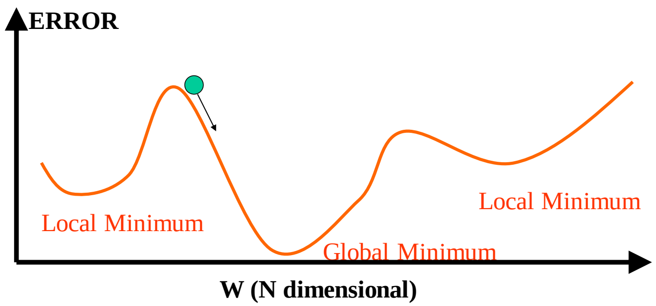
d_k^r and y_k^r are desired and actual output of k*th* unit for training example r.

Where E[w] is the sum of squared errors for the weight vector w, and r ranges over examples in the training set.

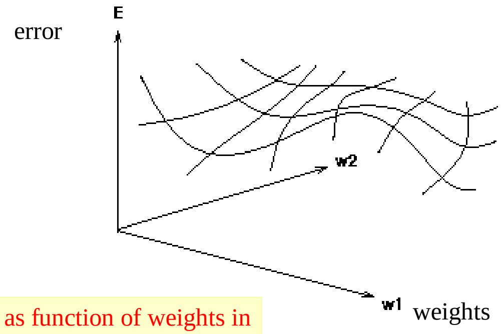
Derivation of Back-propagation

Training Process of the MLP

Training will be continued until the error (RMS) is minimized.



Error Surface



Error as function of weights in multidimensional space

Properties of Activation Function

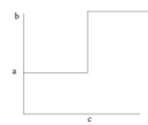
- Trying to make error decrease the fastest
- We need a derivative in activation function
- Activation function must be **continuous**, differentiable, non-decreasing, and easy to compute

Activation functions

- The choice of activation function φ determines the neuron model.

Examples:

- step function: $f(v)=\begin{cases} a & \text{if } v<c \\ b & \text{if } v>c \end{cases}$

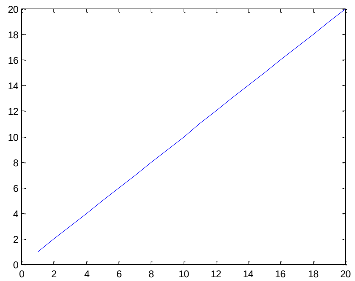


- Can be used for binary classification only
- Not differentiable, not suitable for backprop.

Activation functions

Linear : $y = v$

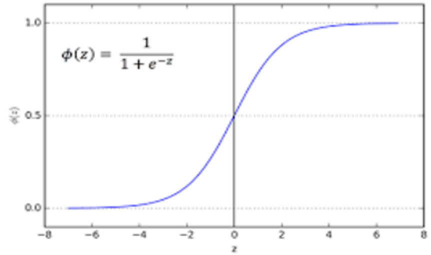
- Range= (-inf, inf)
- Can only learn linear boundaries.



Activation functions

- Logistic
(Sigmoid function)

$$f(v) = \frac{1}{1+\exp(-v)}$$



- Can learn non-linear complex boundaries.
- Squeezes the values to [0,1]
- Most popular activation function a decade ago.

Derivation of Logistic Function

- If the squashing function is the logistic function

$$g(s_i) = \frac{1}{1+e^{-s_i}}$$

the derivative has the convenient form

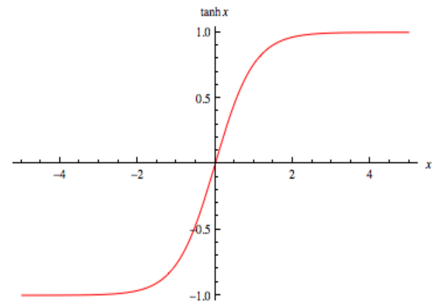
$$g'(s_i) = g(s_i)(1-g(s_i)) = y_i(1-y_i)$$

- Another popular choice of squashing function is tanh, which takes values in the range (-1,1) rather than (0,1)
  - requires plugging a different g′ into the algorithm

Activation functions

Hyperbolic tangent (TanH)

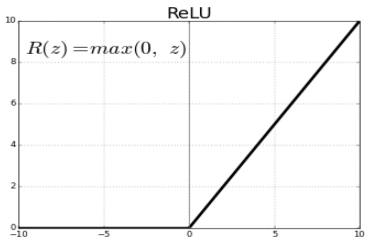
$$y = \frac{\exp(v)-\exp(-v)}{\exp(v)+\exp(-v)}$$



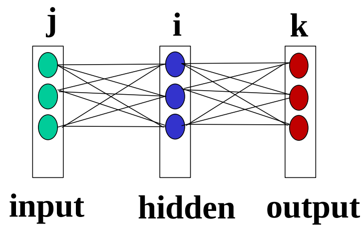
- Range: [-1,1]
- Trains faster than sigmoid.
- Still squeezes the higher and lower values to +1 or -1 resulting in loss of information.

Activation functions

- **ReLu(Rectified Linear Unit):** This function is a partwise linear function which will output the same input directly if it is positive else , it will output zero.
  - It has fast convergence to minimum loss.



Learning of MLP



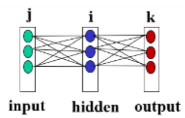
Error in r^th sample

$$E^r = \frac{1}{2}\sum(d_k^r - y_k^r)^2$$

d_k^r and y_k^r are desired and actual output of k*th* unit for training example r.

Overall error $E = \sum_r E^r$

Learning of MLP



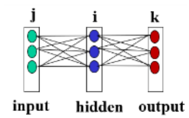
Overall error $E = \sum_r E^r$ $\quad \dfrac{\partial E}{\partial w_{ik}} = \sum_r \dfrac{\partial E^r}{\partial w_{ik}}$

$$-\frac{\partial E^r}{\partial w_{ik}} = -\frac{\partial E^r}{\partial S_k} \times \frac{\partial S_k}{\partial w_{ik}} = \delta_k \times x_i$$

$$S_k = \sum w_{ik} \times x_i$$

$$\frac{\partial S_k}{\partial w_{ik}} = \frac{\partial}{\partial w_{ik}}\left(\sum w_{ik} \times x_i\right) = x_i$$

Learning of MLP



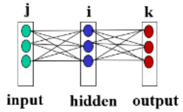
$$-\frac{\partial E^r}{\partial w_{ik}} = -\frac{\partial E^r}{\partial S_k} \times \frac{\partial S_k}{\partial w_{ik}} = \delta_k \times x_i$$

$$\delta_k = -\frac{\partial E^r}{\partial S_k} = -\frac{\partial E^r}{\partial y_k} \times \frac{\partial y_k}{\partial S_k} = \epsilon_k \times g'(S_k)$$

$$y_k = g(S_k)$$

$$\frac{\partial y_k}{\partial S_k} = g'(S_k)$$

Learning of MLP



$$-\frac{\partial E^r}{\partial w_{ik}} = -\frac{\partial E^r}{\partial S_k} \times \frac{\partial S_k}{\partial w_{ik}} = \delta_k \times x_i$$

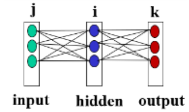
$$= \in_k \times g'(S_k) \times x_i \qquad \boxed{\delta_k = \in_k \times g'(S_k)}$$

At each output node,

$$\in_k = -\frac{\partial E^r}{\partial y_k} = -\frac{\partial}{\partial y_k} = [\frac{1}{2}\sum (d_k^r - y_k^r)^2 \ ] = d_k - y_k$$

Thus, $\delta_k = (d_k - y_k) \times g'(S_k)$

Learning of MLP



At each hidden node,

$$-\frac{\partial E^r}{\partial w_{ji}} = -\frac{\partial E^r}{\partial s_i} \times \frac{\partial s_i}{\partial w_{ji}} = \delta_i \times x_j$$

$$\delta_i = -\frac{\partial E^r}{\partial s_i} = -\frac{\partial E^r}{\partial y_i} \times \frac{\partial y_i}{\partial s_i} = \in_i \times g'(S_i)$$

WHY?
Output y_i of unit *i* is the input x_k of unit *k*.

$$\in_i = -\frac{\partial E^r}{\partial y_i} = -\frac{\partial E^r}{\partial x_k}$$

$$= -\frac{\partial E^r}{\partial S_k} \times \frac{\partial S_k}{\partial x_k} = \delta_k \times w_{ik}$$

$$\delta_i = \delta_k \times w_{ik} \times g'(S_i)$$

$$\boxed{\frac{\partial S_k}{\partial x_k} = \frac{\partial}{\partial x_k}(w_{ik} \times x_k) = w_{ik}}$$

Back Propagation Algorithm

1. Place input vector at input nodes and propagate forward.
2. At each output node *i*, compute $\delta_i = \in_i \times g'(S_i)$
   $= g'(S_i) \times (d_i - y_i)$
3. At each hidden node *i*, compute $\delta_i = g'(S_i) \times \sum \delta_k \times w_{ik}$
4. For each weight w_{ji} compute $\delta_i \times x_j$
   $$w_{ji} \leftarrow w_{ji} + \eta\, \delta_i^r \times x_j^r$$

We need derivative. Activation function must be continuous, differentiable, non-decreasing and easy to compute.