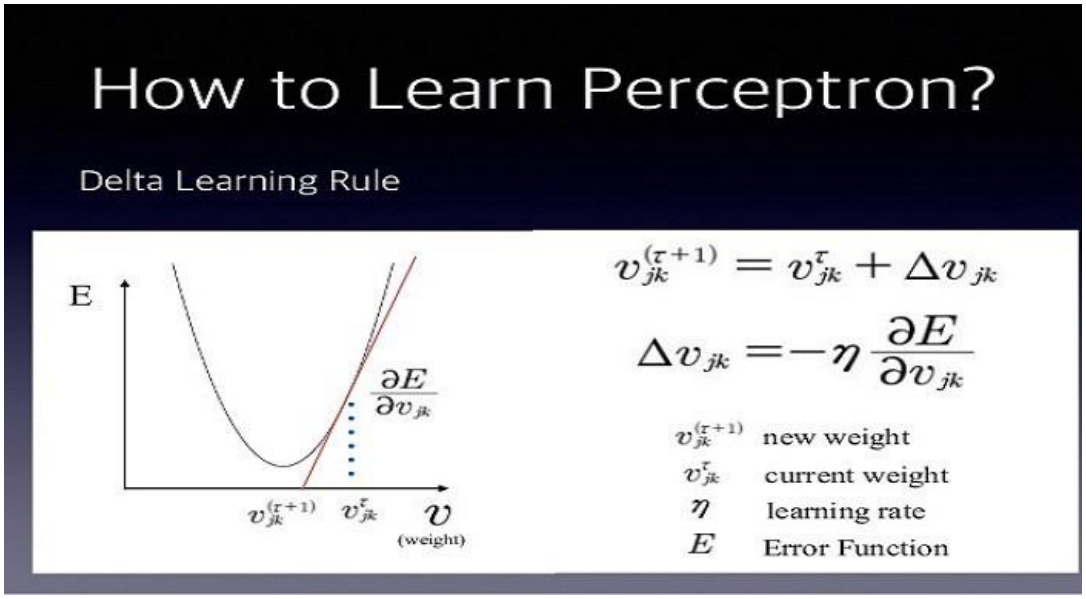
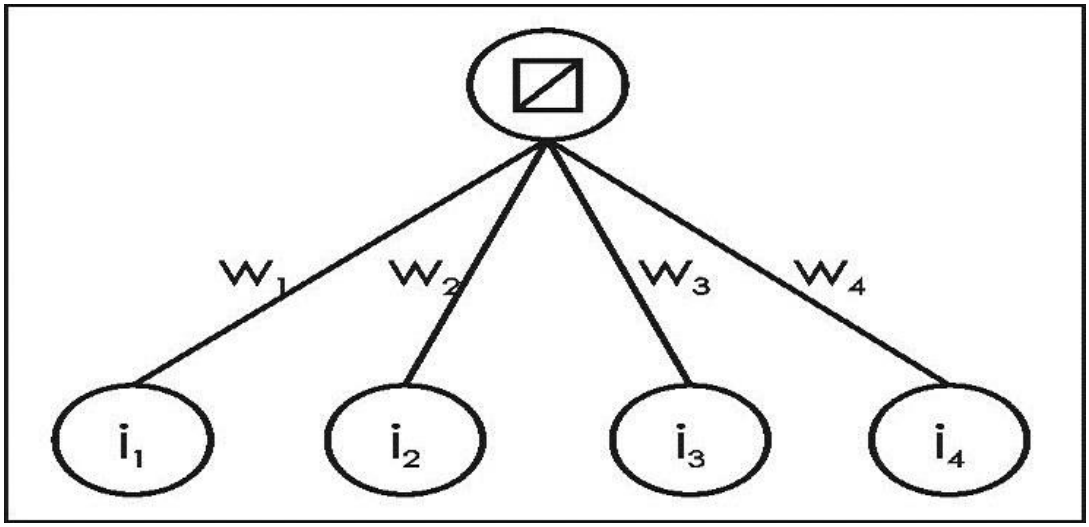
Delta Learning Rule & Gradient Descent | Neural Networks

The development of the perceptron was a big step towards the goal of creating useful connectionist networks capable of learning complex relations between inputs and outputs. In the late 1950's, the connectionist community understood that what was needed for further development of connectionist models was a mathematically-derived (and thus potentially more flexible and powerful) rule for learning. By early 1960's, the Delta Rule [also known as the *Widrow & Hoff Learning rule* or the *Least Mean Square* (LMS) rule] was invented by *Widrow and Hoff*. This rule is similar to the perceptron learning rule by *McClelland & Rumelhart, 1988*, but is also characterized by a mathematical utility and elegance missing in the perceptron and other early learning rules.



The **Delta Rule** uses the difference between *target activation* (i.e., target output values) and *obtained activation* to drive learning. For reasons discussed below, the use of a *threshold activation function* (as used in both the *McCulloch-Pitts* network and the perceptron) is dropped & instead a linear sum of products is used to calculate the activation of the output neuron (alternative activation functions can also be applied). Thus, the activation function is called a *Linear Activation function*, in which the output node's activation is simply equal to the sum of the network's respective input/weight products. The strength of network connections (i.e., the values of the weights) are adjusted to reduce the difference between *target* and *actual* output activation (i.e., error). A graphical depiction of a simple two-layer network capable of deploying the Delta Rule is given in the figure below (Such a network is not limited to having only one output node):



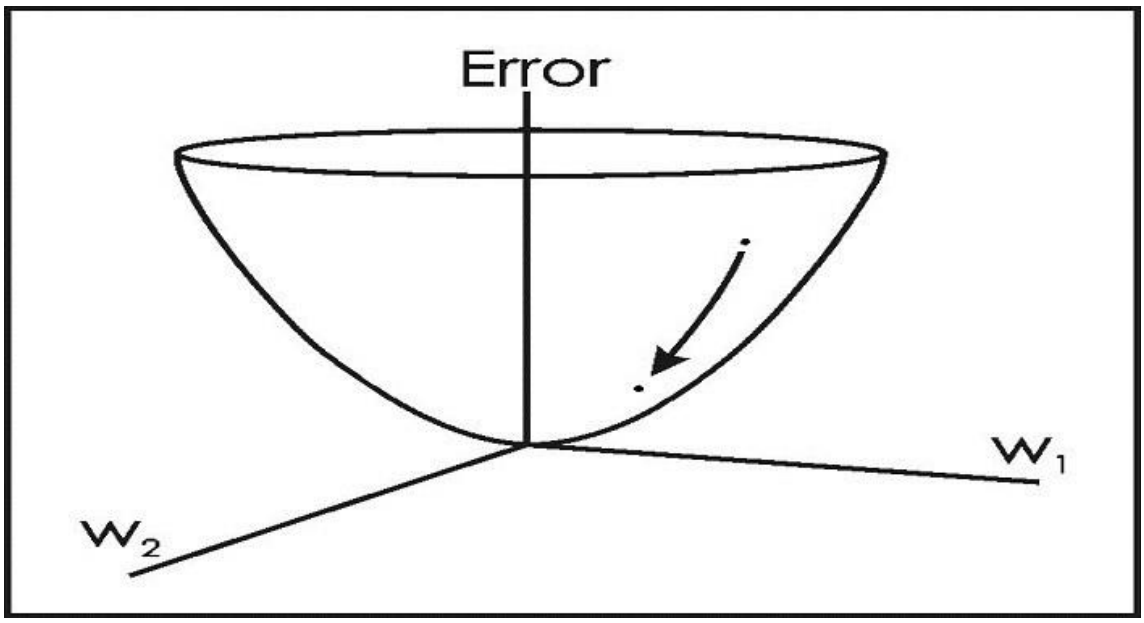
During **forward propagation** through a network, the output (activation) of a given node is a function of its inputs. The inputs to a node, which are simply the products of the output of preceding nodes with their associated weights, are summed and then passed through an activation function before being sent out from the node. Thus, we have the following:

$$S_j = \sum_i w_{ij} a_i$$

and

$$a_j = f(S_i)$$

where '*Sj'* is the sum of all relevant products of weights and outputs from the previous layer *i*, '*wij'* represents the relevant weights connecting layer *i* with layer *j*, '*ai'* represents the activation of nodes in the previous layer *i*, '*aj'* is the activation of the node at hand, and '*f*' is the activation function.



Error function with just 2 weights w*1* and w*2*

For any given set of input data and weights, there will be an associated magnitude of error, which is measured by an error function (also known as a cost function) (e.g., *Oh, 1997; Yam and Chow, 1997*). The Delta Rule employs the error function for what is known as **Gradient Descent learning**, which involves the '*modification of weights along the most direct path in weight-space to minimize error', so* change applied to a given weight is proportional to the negative of the derivative of the error with respect to that weight (*McClelland and Rumelhart 1988, pp.126–130*). The **Error/Cost function** is commonly given as the sum of the squares of the differences between all target and actual node activation for the output layer. For a particular training pattern (i.e., training case), error is thus given by:

$$E_p = \frac{1}{2}\sum_n (t_{j_n} - a_{j_n})^2$$

where '**E*p***' is total error over the training pattern, ½ is a value applied to simplify the function's derivative, 'n' represents all output nodes for a given training pattern, 't*j*' sub *n* represents the *Target value* for node *n* in output layer *j*, and 'a*j*' sub *n* represents the actual activation for the same node. This particular error measure is attractive because its derivative, whose value is needed in the employment of the Delta Rule, and is easily calculated. Error over an entire set of training patterns (i.e., over one iteration, or epoch) is calculated by summing all '**E*p***':

$$E = \sum_p E_p = \frac{1}{2} \sum_p \sum_n (t_{j_n} - a_{j_n})^2$$

Error/Cost Function

where '**E**' is total error, and '*p'* represents all training patterns. An equivalent term for **E** in earlier equation is **Sum-of-squares error**. A normalized version of this equation is given by the **Mean Squared Error (MSE)** equation:

$$MSE = \frac{1}{2PN} \sum_p \sum_n (t_{j_n} - a_{j_n})^2$$

where '**P**' and 'N' are the *total number of training patterns and output nodes*, respectively. It is the error of both previous equations, that gradient descent attempts to minimize (not strictly true if weights are changed after each input pattern is submitted to the network (*Rumelhart et al., 1986: v1, p.324; Reed and Marks, 1999: pp. 57–62*). Error over a given training pattern is commonly expressed in terms of the Total Sum of Squares ('**tss**') error, which is simply equal to the sum of all squared errors over all output nodes and all training patterns. '*The negative of the derivative of the error function is required in order to perform Gradient Descent Learning'*. The derivative of our equation(which measures error for a given pattern '*p'*) above, with respect to a particular weight 'w*ij'* sub '*x*', is given by the chain rule as:

$$\frac{\delta E_p}{\delta w_{ij_x}} = \frac{\delta E_p}{\delta a_{j_z}} \frac{\delta a_{j_z}}{\delta w_{ij_x}}$$

where 'a*j'* sub '*z'* is activation of the node in the output layer that corresponds to weight 'w*ij'* sub *x* (*subscripts* refer to particular layers of nodes or weights, and the 'sub-subscripts' simply refer to individual weights and nodes within these layers). It follows that:

$$\frac{\delta E_p}{\delta a_{j_z}} = (2)(\frac{1}{2})(t_{j_z} - a_{j_z})(-1) = -(t_{j_z} - a_{j_z})$$

and

$$\frac{\delta a_{j_z}}{\delta w_{ij_x}} = \frac{\delta}{\delta w_{ij_x}} \sum_n (w_{ij_n} a_{i_n}) = \frac{\delta}{\delta w_{ij_x}} (w_{ij_0} a_{i_0} + w_{ij_1} a_{1_1} \cdots w_{ij_n} a_{i_n}$$

Thus, the derivative of the error over an individual training pattern is given by the product of the derivatives of our prior equation:

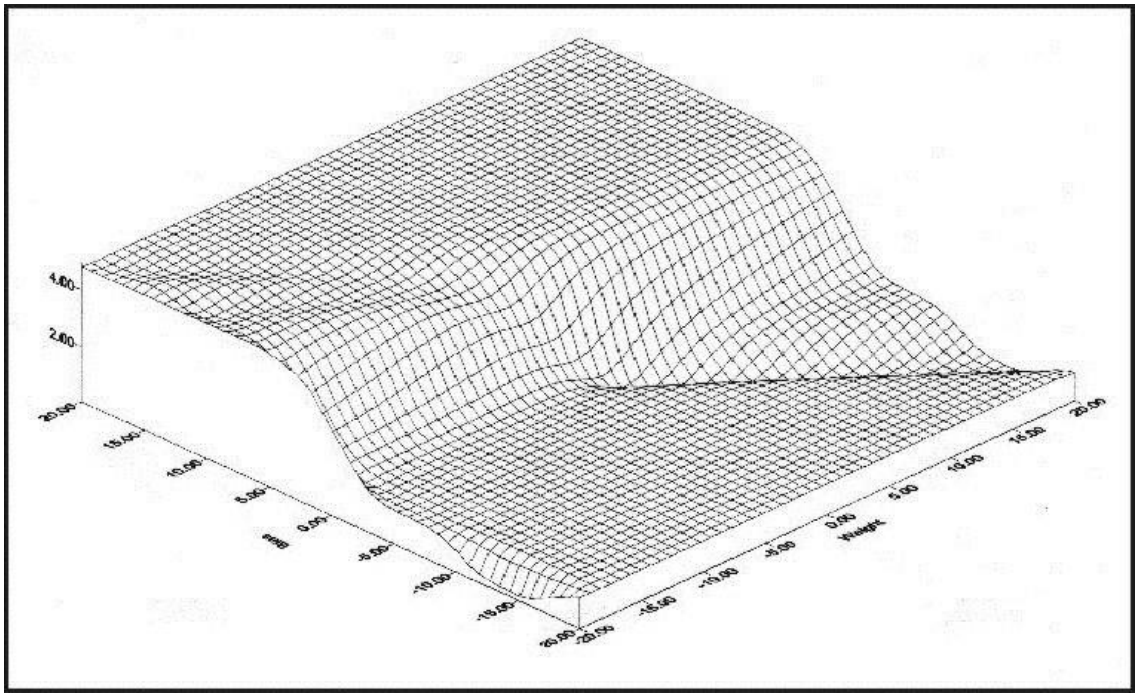
$$\frac{\delta E_p}{\delta w_{ij_x}} = -(t_{j_z} - a_{j_z})(a_{i_x})$$

Because Gradient Descent learning requires that any change in a particular weight be proportional to the negative of the derivative of the error, the change in a given weight must be proportional to the negative of our prior equation . Replacing the difference between the target and actual activation of the relevant output node by **d**, and introducing a learning rate epsilon, that equation can be re-written in the final form of the Delta Rule:

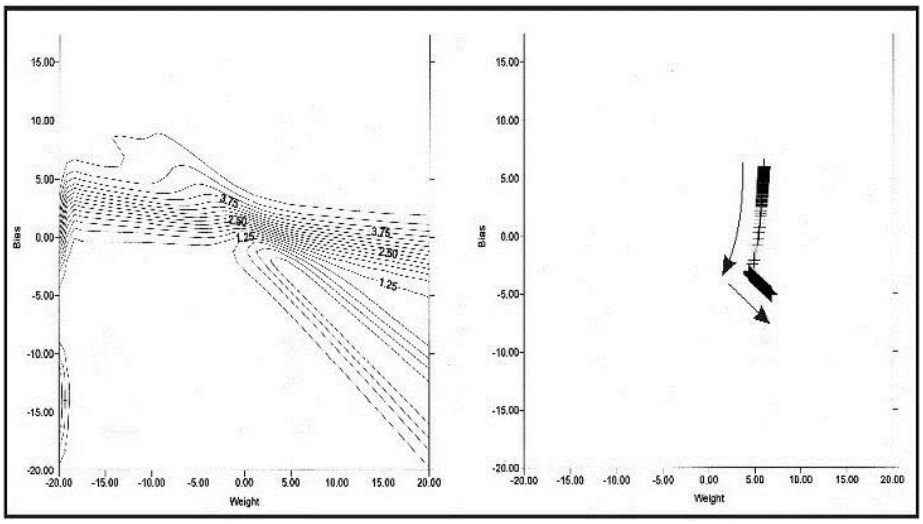
$$\Delta w_{ij_x} = -\varepsilon \frac{\delta E}{\delta w_{ij}} = \varepsilon \delta a_{i_x}$$

Delta Rule for Perceptrons

The reasoning behind the use of a Linear Activation function here instead of a Threshold Activation function can now be justified: Threshold activation function that characterizes both the McColloch and Pitts network and the perceptron is not differentiable at the transition between the activations of **0**and **1** (*slope = infinity*), and its derivative is **0** over the remainder of the function.
Hence, **Threshold activation function cannot be used in Gradient Descent learning**. Whereas a Linear Activation function (or any other function that is differential) allows the derivative of the error to be calculated.



Three-dimensional depiction of an Actual error surface (Leverington, 2001)



Two-dimensional depiction of the error surface