Classification by backpropagation

- Backpropagation is a neural network learning algorithm.
- A neural network is a set of connected input/output units in which each connection has a weight associated with it.
- During the learning phase, the network learns by adjusting the weights so as to be able to predict the correct class label of the input tuples.
- Neural network learning is also referred to as connectionist learning due to the connections between units.
- Neural networks involve long training times and are therefore more suitable for applications where this is feasible.
- Backpropagation learns by iteratively processing a data set of training tuples, comparing the network's prediction for each tuple with the actual known target value.
- The target value may be the known class label of the training tuple (for classification problems) or a continuous value (for prediction).
- For each training tuple, the weights are modified so as to minimize the mean squared error between the network's prediction and the actual target value. These modifications are made in the ―backwards‖ direction, that is, from the output layer, through each hidden layer down to the first hidden layer hence the name is backpropagation.
- Although it is not guaranteed, in general the weights will eventually converge, and the learning process stops.

Advantages:

- It include their high tolerance of noisy data as well as their ability to classify patterns on which they have not been trained.
- They can be used when you may have little knowledge of the relationships between attributes and classes.
- They are well-suited for continuous-valued inputs and outputs, unlike most decision tree algorithms.

- They have been successful on a wide array of real-world data, including handwritten character recognition, pathology and laboratory medicine, and training a computer to pronounce English text.
- Neural network algorithms are inherently parallel; parallelization techniques can be used to speed up the computation process.

Process:

Initialize the weights:

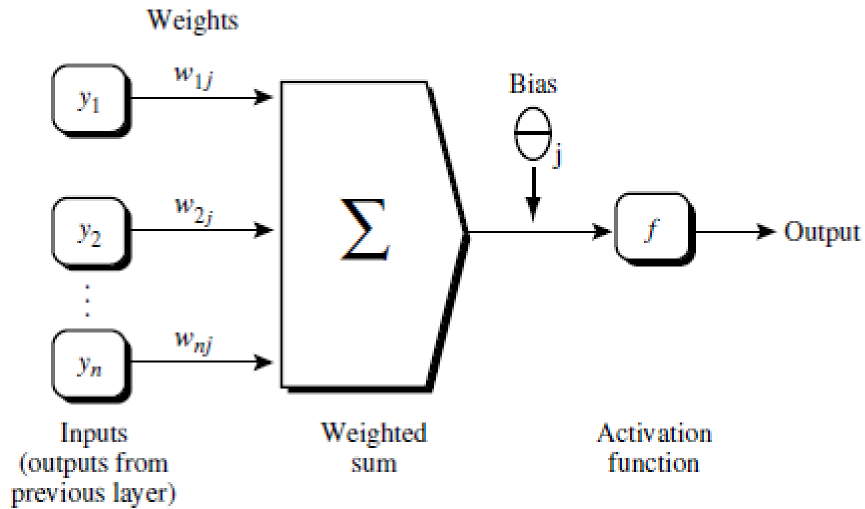
The weights in the network are initialized to small random numbers ranging from-1.0 to 1.0, or -0.5 to 0.5. Each unit has a *bias* associated with it. The biases are similarly initialized to small random numbers. Each training tuple, ***X***, is processed by the following steps.

Propagate the inputs forward:

First, the training tuple is fed to the input layer of thenetwork. The inputs pass through the input units, unchanged. That is, for an input unit*j*, its output, *Oj*, is equal to its input value, *Ij*. Next, the net input and output of eachunit in the hidden and output layers are computed. The net input to a unit in the hiddenor output layers is computed as a linear combination of its inputs. Each such unit has anumber of inputs to it that are, in fact, the outputs of the units connected to it in theprevious layer. Each connection has a weight. To compute the net input to the unit, each input connected to the unit ismultiplied by its correspondingweight, and this is summed.

$$I_j = \sum_i w_{ij} O_i + \theta_j,$$

Where $w_{i,j}$ is the weight of the connection from unit i in the previous layer to unit j; O_i is the output of unit i from the previous layer θ_j is the bias of the unit & it acts as a threshold in that it serves to vary the activity of the unit. Each unit in the hidden and output layers takes its net input and then applies an activation function to it.



Back propagate the error:

The error is propagated backward by updating the weights and biases to reflect the error of the network's prediction. For a unit *j* in the output layer, the error Err_j is computed by

$$Err_j = O_j(1 - O_j)(T_j - O_j)$$

Where Oj is the actual output of unit j, and Tj is the known target value of the given training tuple. The error of a hidden layer unit *j* is

$$Err_j = O_j(1 - O_j)\sum_k Err_k w_{jk};$$

Where wjk is the weight of the connection from unit j to a unit k in the next higher layer, and Errk is the error of unit k. Weights are updated by the following equations, where D*wi j* is the change in weight *wi j*:

$$\Delta w_{ij} = (l)Err_j O_i$$

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

Biases are updated by the following equations below

$$\Delta\theta_j = (l)Err_j$$

$$\theta_j = \theta_j + \Delta\theta_j$$

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 θ_j in *network* {
(19) $\Delta\theta_j = (l) Err_j$; // bias increment
(20) $\theta_j = \theta_j + \Delta\theta_j$; } // bias update
(21) } }