Multilayer Feedforward Neural Networks

Basic Structure

A multilayer feedforward network consists of a set of neurons that are logically arranged into two or more layers. There is an input layer and an output layer, each containing at least one neuron. There are usually one or more "hidden" layers sandwiched between the input and output layers. The term, **feedforward**, means that information flows in one direction only. The inputs to neurons in each layer come exclusively from the outputs of neurons in previous layers, and outputs from these neurons pass exclusively to neurons in following layers.

Basic Structure

The output of each neuron in the network is a function of that neuron's inputs. Given an input to such a network, the activations of all output-layer neurons can be computed in one deterministic pass; iteration is not required, and randomness does not play a role.

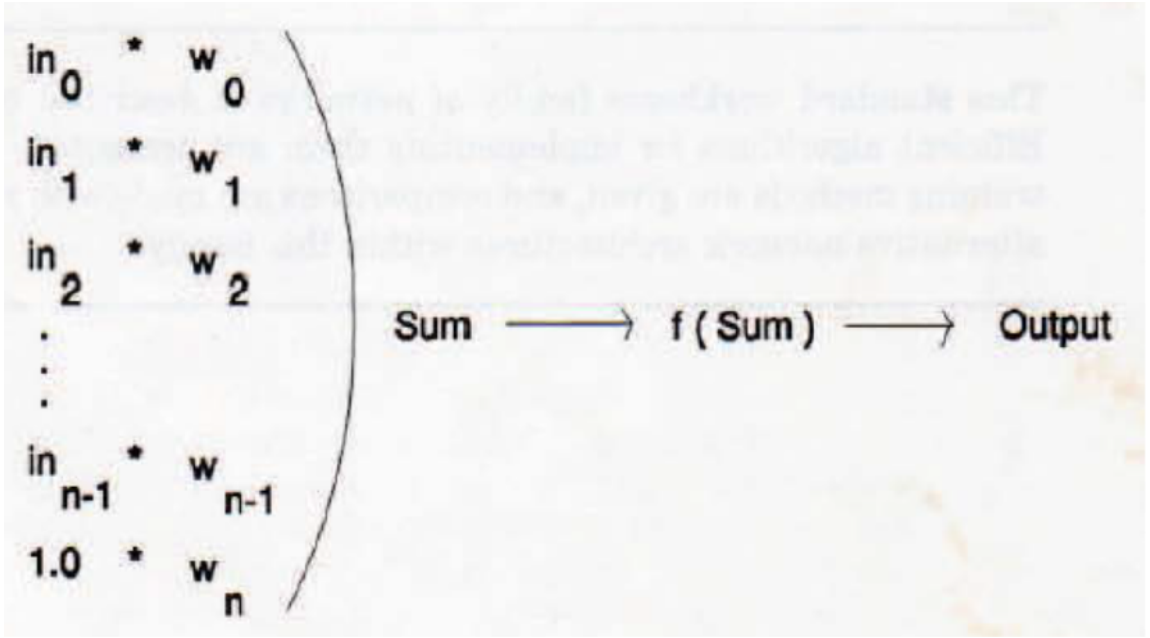
Some models allow connections to skip layers, in that outputs of neurons in one layer may connect directly to inputs of neurons in layers past the immediately following layer. Such networks will not be discussed in this course.

Basic Structure

A single neuron is shown schematically in figure. It has n inputs, labeled from 0 through n - 1. It also has one assumed input, called its bias, which is always equal to 1.0. The neuron is characterized by n + 1 weights which multiply each input, and an activation function which is applied to the weighted sum of inputs in order to produce the neuron's output. The weighted sum of inputs. Including the bias, is frequently called the net input. Thus, if our n inputs are

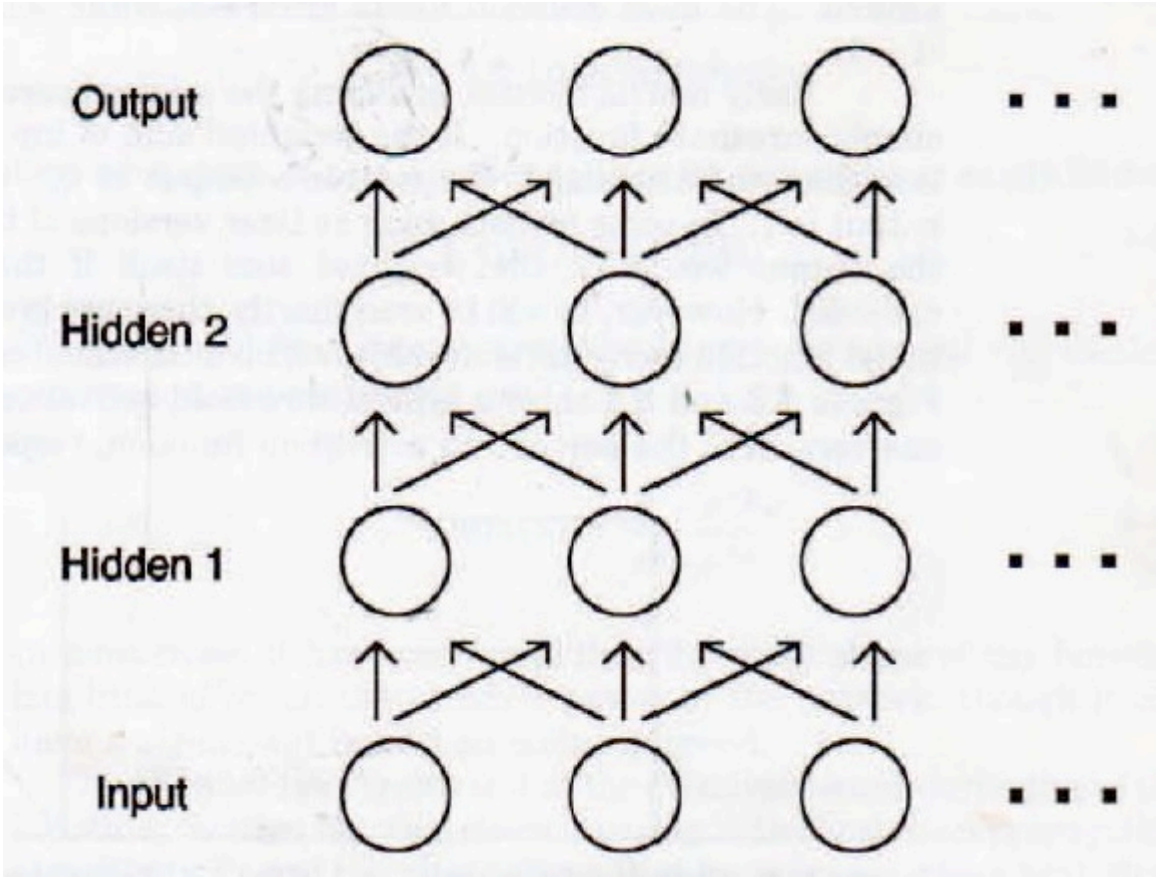
{xi, i = 0, ..., n-1}. the neuron's output is calculated as:

$$out = f(net) = f\left(\sum_{i=0}^{n-1} x_i w_i + w_n\right)$$



Basic Structure

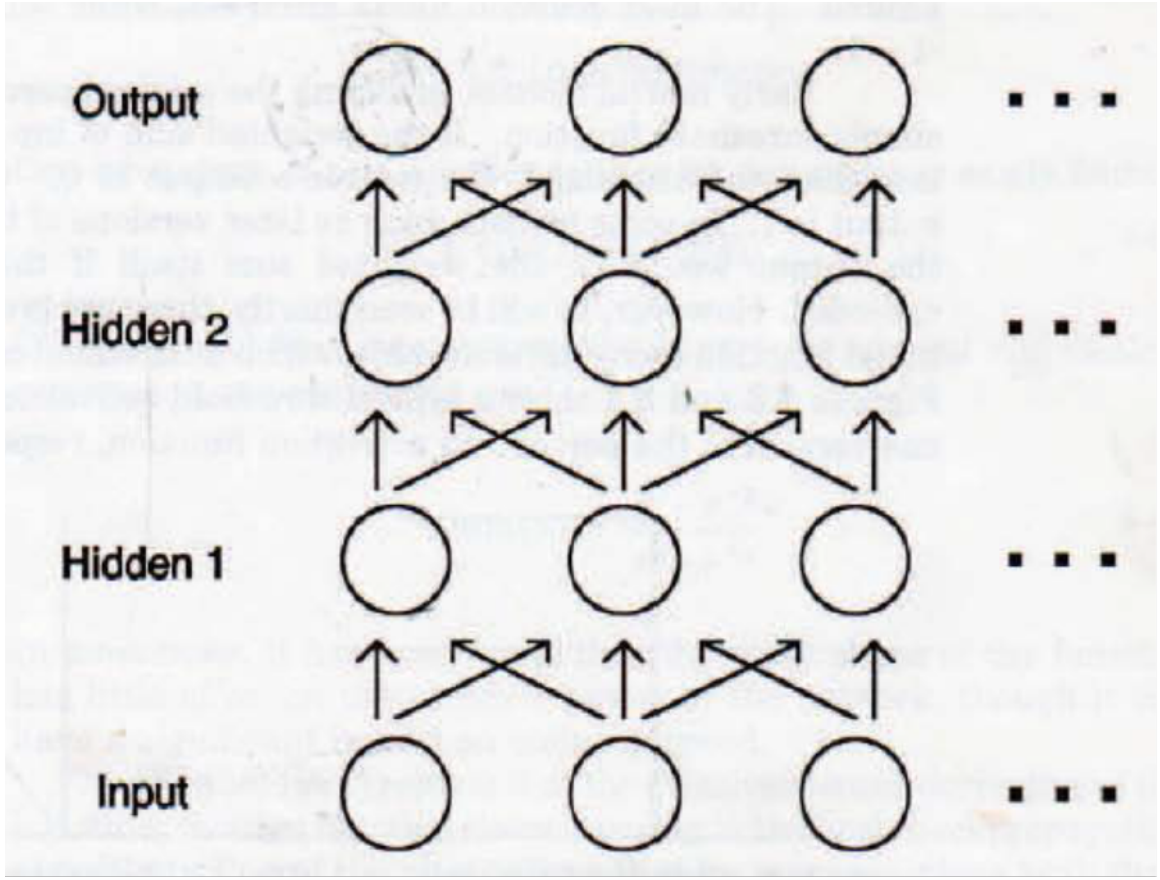
- The operational characteristics of this neuron are primarily controlled by the weights, wi. Although the activation function f(net) is also obviously important. it will be seen that in practice the neuron's operation is generally little affected by the exact nature of the activation function as long as some basic requirements are met.
- Training speed, on the other hand. may be strongly impacted by the activation function. This will be discussed later. Feedforward networks usually have a single layer of hidden neurons sandwiched between the input and output layers. Such a network is called a three-layer network. Rarely, two hidden layers will be needed. A four-layer network is shown schematically in the figure.



Basic Structure

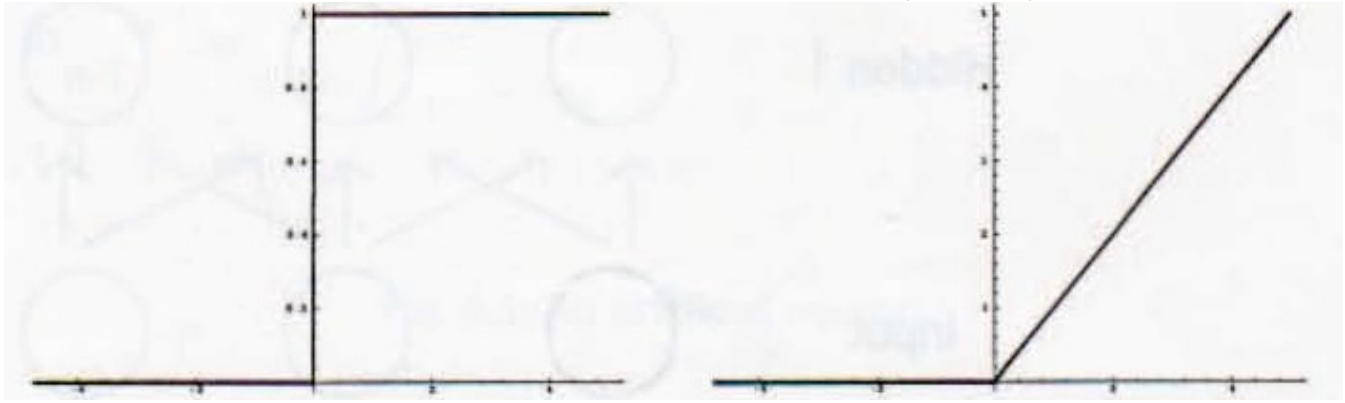
- Each neuron in a feedforward network. represented by circles in the figure, operates as shown in the below equation. Its inputs come from the previous layer. and its outputs go to the next layer. Nearly always, the same activation function is used for all neurons. It is the weights connecting neurons in one layer to those in the next which primarily determine the behavior of the network.

$$out = f(net) = f\left(\sum_{i=0}^{n-1} x_i w_i + w_n\right)$$



Activation Functions

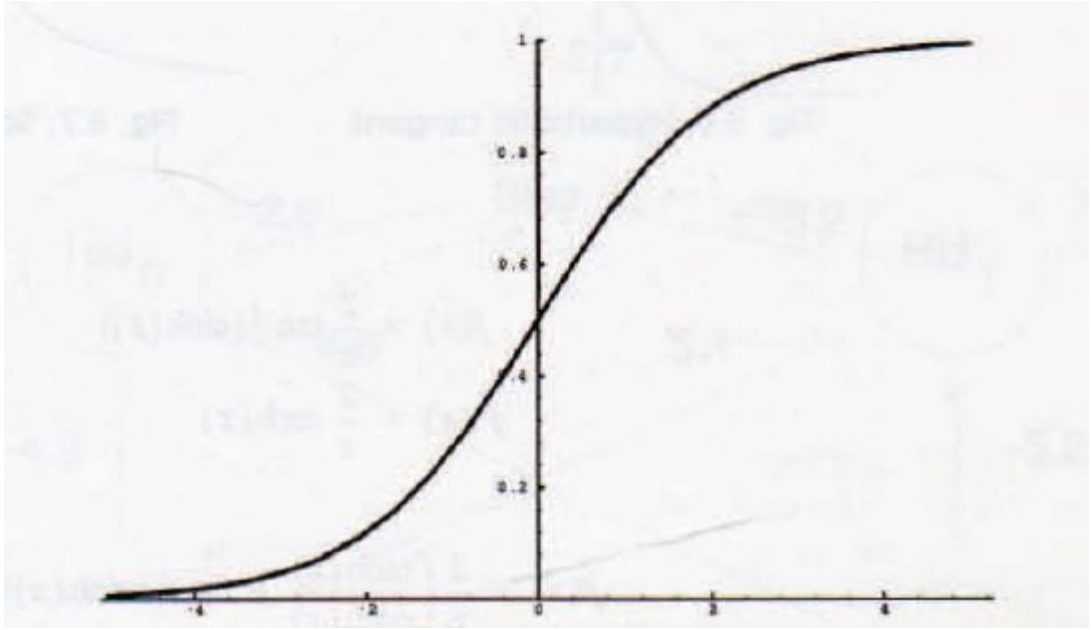
- The activation function is a nonlinear function that, when applied to the net input of a neuron, determines the output of that neuron. Its domain must generally be all real numbers, as there is no theoretical limit to what the net input can be. (In practice we can easily limit the net input by limiting the weights, and often do. Nevertheless, activation functions almost always have an unlimited domain.) The range of the activation function (values it can output) is usually limited. The most common limits are 0 to 1, while some range from -1 to +1.
- Figures show a typical threshold activation function and one version of the perceptron activation function, respectively.



Activation Functions

The majority of current models use a sigmoid(S-shaped) activation function. A sigmoid function may be loosely defined as a continuous, real-valued function whose domain is the reals, whose derivative is always positive, and whose range is bounded. The most commonly employed sigmoid function is the *logistic* function.

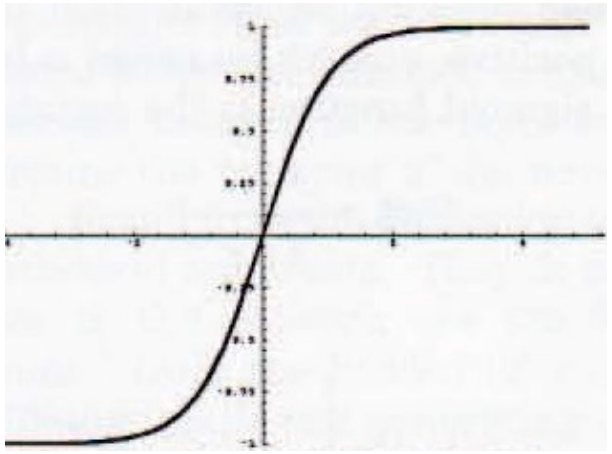
$$f(x) = \frac{1}{1+e^{-x}}$$



Activation Functions

One advantage of this function is that its derivative of ***f(x)*** is easily found:

f(x) (1-f(x))



Other sigmoid functions. such as the *hyperbolic tangent* and (scaled) *arctangent,* are sometimes used.

$$\tanh(x) = \frac{e^{x} - e^{-x}}{e^{x} + e^{-x}}$$

In most cases, it has been found that the exact shape of the function has little effect on the ultimate power of the network, though it can have a significant impact on training speed.

Linear Output Neurons

- In this course, it is assumed that the output neurons have the same activation function as the hidden neurons, typically the logistic function. This is traditional and is generally recommended. However, it must be emphasized that it is not required. In fact, in some cases, it may be detrimental. Many applications use the identity function, f(x) = x, as the activation function of the output-layer neurons. In other words, the output of each of these neurons is equal to its net input.
- The implication is that nonlinear optimization needs to be done only on hidden-layer weights. For any given set of hidden-layer weights, output· layer weights that deliver the global minimum of the mean square error can be explicitly computed. This is a significant savings, especially if there are many output neurons.
- There is one potentially serious drawback to linear activation functions. This concerns noise immunity. Although the squashing functions in the hidden layer provide a fair degree of buffering, the extra amount provided at the output layer can sometimes be valuable. Also, the problems of nonlinear output activation functions are rarely very serious. The best approach is usually to use squashing functions first, switching to linear functions only if there is a clear reason to do so.

Training the Network

- The training process starts by initializing all weights to small non-zero values. Often these are generated randomly. Then, a subset of the collection of training samples is presented to the network, one at a time. A measure of the error incurred by the network is made, and the weights are updated in such a way that the error is reduced. This process is repeated as necessary.
- One pass through this [present a subset of the training set ... measure error ... update weights] cycle is called an **epoch**. The size of the subset (number of training samples used per weight update) is called the **epoch size**. Often, the entire training set is used for the epoch. If smaller subsets are used, it is important that they be chosen randomly, or learning may be impaired.
- The most common error measure used is the **mean square error** in output activations. It is easily computed, has proven itself in practice, and perhaps most importantly, its partial derivative with respect to individual weights can be computed explicitly.

Training the Network

- The mean square error for a single presentation is found by squaring the difference between the attained and target activations for each output neuron, and averaging across them all. The epoch error is computed by averaging the errors of the training presentations within that epoch.
- If there are n output neurons, the error for the single presentation is:

$$E_p = \frac{1}{n} \sum_{j=0}^{n-1} (t_{pj} - o_{pj})^2$$

- If there are *m* presentations in the epoch, the error for that epoch is

$$E = \frac{1}{m} \sum_{p=0}^{m-1} E_p$$

Training the Network

- If we know the partial derivative of the error with respect to each weight, we know (at least on a local scale) which way the weights must move in order to reduce the error. Here we simply state that for a single presentation, the derivative of the output layer weight connecting previous layer neuron *i* to output neuron j is:

$$\frac{\partial E}{\partial w_{ji}} = -o_i \, f'(net_j) \, (t_j - o_j)$$

where **O*i*** is the output of previous layer neuron *i*, *netj* is the weighted sum coming into output layer neuron *j*, ***Oj*** is that neuron's obtained activation, and *tj* is the desired activation for that neuron. Note that we need to know the derivative of the activation function.

Training the Network

The formula is often written in two parts:

$$\delta_j = f'(net_j)\ (t_j - o_j)$$

$$\frac{\partial E}{\partial w_{ji}} = -\ o_i\ \delta_j$$

Partial derivatives with respect to hidden-layer weights may be computed if the delta values for the following layer are known. In the following formula, *wkj* is the weight connecting neuron j in this hidden layer to neuron *k* in the next layer. The delta shown with a subscript of k refers to the deltas for the layer following this hidden layer, while the delta having a subscript of j refers to the deltas being computed for this hidden layer.

$$\delta_j = f'(net_j) \sum_k (\delta_k\ w_{kj})$$

Training the Network

- It can be seen that evaluation of the derivatives occurs in the opposite order as executing the network. The output layer is done first. Its deltas are then used to compute derivatives for the next layer back, et cetera. It is this *backward propagation* of output errors which inspired the name for a popular training method.
- The above formulas are for a single presentation of an input pattern. To compute the gradient for an entire training epoch, the gradients for each sample are summed. Recall that the derivative of the sum of functions is the sum of their individual derivatives.

Gradient Descent Algorithm

- Backpropagation is what numerical analysts call a *gradient descent* algorithm. The gradient of a multivariate function is the direction that is most steeply "uphill". A tiny step in that direction will result in the maximum increase of the function compared to all other possible directions.
- By the same token, a tiny step in the opposite direction will effect the maximum possible decrease in the function. As discussed earlier, our function is the network's error for the training set. Thus, it seems to make sense to compute the gradient of this error function, take a step in the opposite direction (the direction of the negative gradient), and repeat as needed.
- Since we are always stepping in the optimal direction for reducing the error (at least locally), we would expect to descend to the minimum error location quite quickly. The exact distance to step, often called the learning rate, can be critical. If this distance is too small, convergence will be excessively slow. If it is too large, we will jump wildly and never converge.