J. Bangladesh Electron. 10 (1-2); 33-47, 2010

First-Order Optimization Method for Single and Multiple-Layer Feedforward Artificial Neural Networks

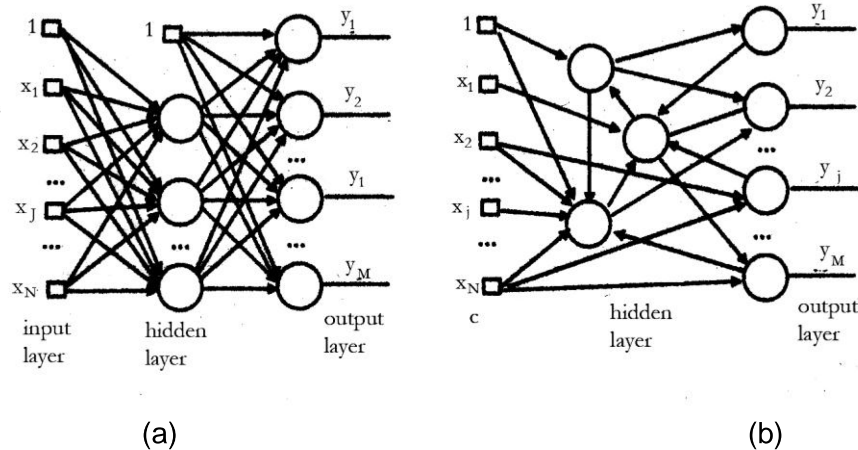
Muhammad Hanif[1] and Jamal Nazrul Islam[2]
[1]Assistant Professor , Department of Mathematics, Noakhali Science and Technology University, Noakhali 3802, Bangladesh.
[2]Emeritus Professor, Chittagong University, Chittagong 4331, Bangladesh.

Abstract

In this study, we apply the first-order optimization method in single and multiple-layer feedforward artificial neural networking problem and obtained some useful results. Trial and error basis has been applied in determining the internal parameters of the network considered as having a significant influence over its performance: the number of hidden layers, activation function, number of neurons in the hidden layers, training epochs, learning rate, and momentum term. Several practical examples were chosen to exemplify the use of the algorithm for finding the optimal parameters.

1. Introduction

The topology of artificial neural networks is defined by the number of layers, the number of units per layer, and the interconnection patterns between layers. They are generally divided into two categories based on the pattern of connections: feedforward and recurrent or feedback. In feedforward artificial neural networks, the neurons are organized in the form of layers. The neurons in a layer get input from the previous layer and feed their output to the next layer. In this kind of networks connections to the neurons in the same or previous layers are not permitted. The last layer of neurons is called the output layer and the layers between the input and output layers are called the hidden layers. The input layer is made up of special input neurons, transmitting only the applied external input to their outputs. In a network if there is only the layer of input nodes and a single layer of neurons constituting the output layer then they are called single layer network. If there are one or more hidden layers, such networks are called multilayer networks. Utilizing this design, nodes may be added together into a multilayered structure, in which outputs from previous layers are cumulatively weighted and added to form inputs for subsequent layers. The information is passed up through the layers, from input to output. In the multi-layer feedforward artificial neural networks, values are introduced at the input layer and are weighted then passed to a hidden layer of nodes, before being weighted again, prior to reaching the output layer. Hidden layers offer the benefit of not being the first or last layer within a network, and thus are never responsible for handling raw data or final output. In recurrent artificial neural network; some of its outputs are connected to its inputs, which contain feedback connections. Contrary to feed-forward artificial networks, the dynamical properties of the network are important. In some cases, the activation values of the units undergo a relaxation process such that the network will evolve to a stable state in which activation does not change further. In other applications in which the dynamical behavior constitutes the output of the network, the changes of the activation values of the output units are significant. Recurrent artificial neural networks are potentially more powerful than feedforward networks and can exhibit temporal behavior. Multi-layer feedforward and recurrent artificial neural network is shown in Figure 1.



(a) (b)

Fig. 1: (a) Feedforward artificial neural networks (b) Recurrent artificial neural network

This paper basically deals with the optimization of feedforward artificial neural networking problem. Our interest is to find the optimal solution of the problem by applying the first-order optimization method.

2. First-Order Optimization Method

The first-order techniques in the context of feedforward artificial neural networks is error backpropagation, a training algorithm for optimizing the connection weights of a feedforward ANN for a given problem. 'Backprop' can be viewed as an *unconstrained nonlinear optimization* scheme which combines the classical gradient descent *(or steepest descent)* and backsubstitution methods. It has been one of the most studied and used algorithms for neural networks learning ever since. This method is not only more general than the usual analytical derivations, which handle only the case of special network topologies, but also much easier to follow. It also shows how the algorithm can be efficiently implemented in computing systems in which only local information can be transported through the network. The inefficiency of steepest descent is due to the fact that the minimization directions and step sizes are poorly chosen: unless the first step is chosen such that it leads directly to the minimum, steepest descent will zig-zag with many steps.

The analytical calculation of the optimum weight vector for a problem is rather difficult in general. A better approach would be to let the Adaline Linear Combiner to find the optimum weights by itself through a search over the error surface. Instead of having a purely random search, some intelligence is added to the procedure such that the weight vector is changed by considering the gradient of e(**w**) iteratively. According to formula known as *delta rule* and following the *Method of Steepest Descent* [1]we write:

$$\Delta w = w(k+1) - w(k)$$

where

$$\Delta w = -\eta \nabla e(w(k))$$

In the above formula η is a small positive constant, determining the learning rate.

Note that for a real valued scalar function $E(\mathbf{w})$ on a vector space $\mathbf{w} \in \Re^N$, the gradient $\nabla E(\mathbf{w})$ gives the direction of the steepest upward slope, so the negative of the gradient is the direction of the steepest descent. This fact is demonstrated in Figure 2 for a parabolic error surface on two dimensions:

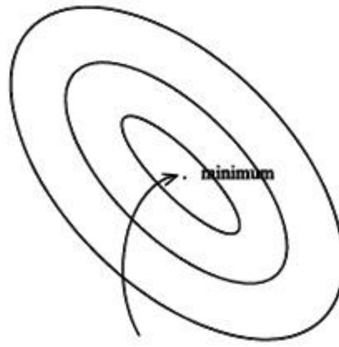


Fig. 2: Direction of the steepest gradient descent on the paraboloid error surface on two dimensional weight space. Only the equpotential curves of the error surface is shown instead of the 3D-error surface.

3. Statement of the Problem

The training process of feedforward artificial neural networks involves adjusting the weights till a desired input/output relationship is obtained. The mathematical characterization of a multilayer feedforward artificial neural network is that of a composite application of functions. Each of these functions represents a particular layer and may be specific to individual units in the layer, e.g. all the units in the layer are required to have same activation function. The overall mapping is thus characterized by a composite function relating feedforward network inputs to output.

Consider a feed-forward feedforward neural network with n input and m output units. It can consist of any number of hidden units and can exhibit any desired feed-forward connection pattern. We are also given a training set $\{(x_1, y_1), . . . , (x_d, y_d)\}$, where $\mathbf{x_d}$ **,** $\mathbf{y_d}$ consisting of d-ordered pairs of n- and m-dimensional vectors, which are called the input and output patterns. Let the primitive functions at each node of the network be continuous and differentiable. The weights of the edges are real numbers selected at random. Suppose that $y_d^k(t)$ denote the desired (target) outcome for the kth neuron at time t and the actual outcome of the neuron is $y^k(t)$. Suppose the outcome $y^k(t)$ was produced when x_i (input) applied to the network. If the actual response $y^k(t)$ is not same as $y_d^k(t)$, we may define an error signal as

$$e_k(t) = y_d^k(t) - y^k(t) \tag{1}$$

The error calculations used to train a neural network are very important. Typically error calculations are very different depending primarily on the network's application. The purpose of error calculations learning is to minimize an objective (cost) function based on the error signal $e_k(t)$. Once an objective or a cost function is selected, error calculations learning is strictly an *optimization problem*. The most popular error function is the sum-of-squared errors, or one of its scaled versions. This is analogous to using the minimum least squares optimization criterion in linear regression. Like least squares, the sum-of-squared errors is calculated by looking at the squared difference between what the network predicts for each training pattern and the target value, or observed value, for that pattern.

Formally, the equation is the same as one-half the traditional sum of squares error function:

$$E = \frac{1}{2}\sum_k e_x^2(t) = \frac{1}{2}\sum_k \left(y_d^k(t) - y^k(t)\right)^2 \tag{2}$$

Here summation runs over all k[th] neurons in the output layer of the network and the symbol *E* for the objective function to be *minimized.* i.e., the adjusting the weights of the network such that the actual output $y^k(t)$ generated by the network for the given input is as "close" to $y_d^k(t)$ as possible.

4. Implementation

In this section, we implement the method in section 2 to single and multiple-layer feedforward artificial neural networks.

4.1 Single-Layer Network

Consider a single layer multiple output network as shown in the Figure3. There are n inputs x_i, where $i = 1.....,n.$ We have m outputs y_s, $s = 1,...m.$ We may think of the neurons in the input layer as single-input-single-output linear elements, with each activation function f being the identity map. Here **w** = (*wij*) is the mxn order weighted matrix is used to denote the strength of the connection from the *i*th input to the *j*th processing element. Let y_{ds} be the desired output and e_s be the error observed at the output of the s neurons. Note that the activation function is a function from $\Re$ to $\Re$.

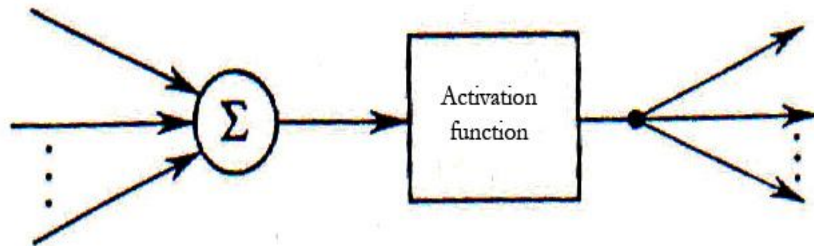


Fig. 3: Single layer with multiple neuron output network.

We have $y_s = f$ (**W X**) where **W** = (w_{ij}) = $\begin{pmatrix} w_{11} & w_{12} \ldots\ldots w_{1n} \\ w_{21} & w_{22} \ldots\ldots w_{2n} \\ \ldots\ldots & \ldots\ldots \quad \ldots\ldots \\ w_{m1} & w_{m2} \ldots\ldots w_{mn} \end{pmatrix}$, **X** = $\begin{pmatrix} x_1 \\ x_2 \\ \vdots \\ x_n \end{pmatrix}$ (3)

and

$$e_s = y_{ds} - y_s \tag{4}$$

when x_i is applied at the input.

If we define the total output error for input x_i as the sum of the square of the errors at each neuron output, that is:

$$E = \frac{1}{2}\sum_{s=1}^{m} e_s^2 = \frac{1}{2}\left[e_1^2 + e_2^2 + \ldots\ldots + e_m^2\right] \tag{5}$$

If we take partial derivative of (6) with respect to w_{ji} by applying the chain rule

$$\frac{\partial e_s}{\partial w_{ij}} = \frac{\partial e_s}{\partial y_s} \cdot \frac{\partial y_s}{\partial w_{ij}} \tag{6}$$

We have $\frac{\partial e_s}{\partial y_s} = -1, \frac{\partial y_s}{\partial w_{ij}} = f\ (WX).X.\frac{\partial W}{\partial w_{ij}} = X.f'(WX)$ $\left[\because \frac{\partial W}{\partial w_{ij}} = 1\right]$ (7)

From (6), we write $\frac{\partial e_s}{\partial w_{ij}} = -X.f'(WX)$ (8)

If we take partial derivative of (5) with respect to w_{ji}, when x_i is applied at the input by applying the chain rule

$$\frac{\partial E}{\partial w_{ij}} = \frac{\partial E}{\partial e_s} \cdot \frac{\partial e_s}{\partial w_{ij}} \tag{9}$$

We have $$\frac{\partial E}{\partial e_s} = [e_1, e_2, ..., e_s] = \mathbf{e_s} \tag{10}$$

From (8), (9) and (10), we write

$$\frac{\partial E}{\partial w_{ij}} = - e_s . X . f'(WX) \tag{11}$$

By defining

$$\delta = e_s \; f'(WX)$$

it can be reformulated as

$$\frac{\partial E}{\partial w_{ij}} = - \delta . X \tag{12}$$

In order to reach the minimum of the total error, we apply the delta rule in the same way explained for the steepest descent algorithm:

$$w_{ij}(k+1) - w_{ij}(k) = -\eta \frac{\partial E}{\partial w_{ij}} \tag{13}$$

where represents a learning constant η, i.e., a proportionality parameter which defines the step length of each iteration in the negative gradient direction.

From (12) and (13), we have

$$w_{ij}(k+1) - w_{ij}(k) = \eta . \delta . X$$

That is; $$w_{ij}(k+1) = w_{ij}(k) + \eta . \delta . X \tag{14}$$

Where $\delta = e_s \; f'(WX)$, $\mathbf{X} = \begin{pmatrix} x_1 \\ x_2 \\ \vdots \\ x_n \end{pmatrix}$, $\mathbf{W} = (w_{ij}) = \begin{pmatrix} w_{11} \; w_{12} \; .... w_{1n} \\ w_{21} \; w_{22} \; .... w_{2n} \\ ...... \quad ...... \quad ..... \\ w_{m1} \; w_{m2} \; ...... \; w_{mn} \end{pmatrix}$

4.2 Multiple-Layer Network

Consider the three layers, referred to as the input, hidden, and output layers. There are n inputs x_i, where $i = 1.....,n.$ We have m outputs y_s, $s = 1,...m.$ There are l neurons in the hidden layer. The outputs of the neurons in the hidden layer are z_j ,where $j = 1,....,l.$ The inputs $x_1,....,x_n$ are distributed to the neurons in the hidden layer. We may think of the neurons in the input layer as single-input-single-output linear elements, with each activation function being the identity map. In Figure 4, we do not explicitly depict the neurons in the input layer, instead, we illustrate the neurons as signal splitters. We denote the activation functions of the neurons in the each layer by f. Note that each activation function is a function from $\Re$ to $\Re$.

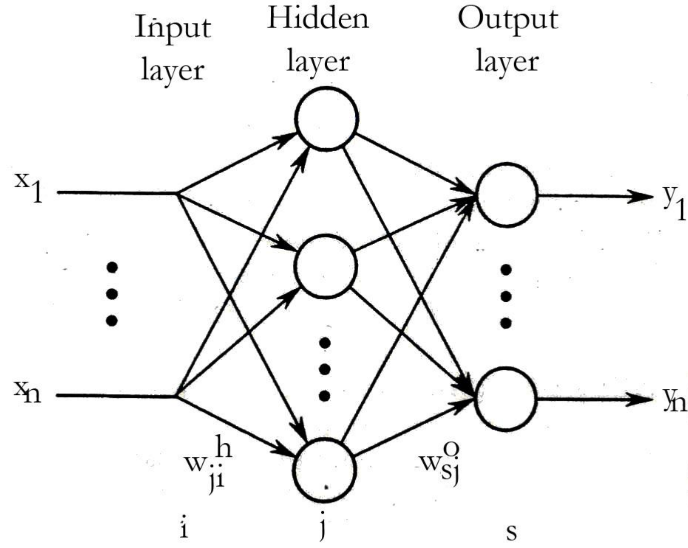


Fig. 4: A three-layered feedforward neural network

We denote the weights for inputs into the hidden layer by $w_{ji}^{h}, i = 1,....,n, j = 1,....,l.$ We denote the weights for inputs from the hidden layer into the output layer by $w_{sj}^{0}, j = 1,....,l, s = 1,.....,m.$ Given the weights w_{ji}^{h} and w_{sj}^{0}, the neural network implements a map from $\Re^{n}$ to $\Re^{m}$. To find an explicit formula for this map, let us denote the input to the jth neuron in the hidden layer by v_j, and the output of the jth neuron the hidden layer by z_j. Then, we have

$$v_j = \sum_{i=1}^{n} w_{ji}^{h} x_i,$$

$$z_j = f\left(\sum_{i=1}^{n} w_{ji}^{h} x_i\right).$$

The output from the sth neuron of the output layer is

$$y_i = f\left(\sum_{j=1}^{l} w_{sj}^{0} z_j\right).$$

Therefore, the relationship between the inputs $x_i = 1,....,n,$ and the sth output y_s is given by

$$y_s = f\left(\sum_{j=1}^{l} w_{sj}^{0} f(v_j)\right)$$

$$= f\left(\sum_{j=1}^{l} w_{sj}^{0} f\left(\sum_{i=1}^{n} w_{ji}^{h} x_i\right)\right)$$

$$= F_s(x_1,....,x_n).$$

The overall mapping that the neural network implements is therefore given by

$$\begin{bmatrix} y_1 \\ \vdots \\ y_m \end{bmatrix} = \begin{bmatrix} F_1(x_1,....,x_n) \\ \vdots \\ F_m(x_1,....,x_n) \end{bmatrix}.$$

We now consider the problem of training the neural network.The training of the neural network involves adjusting the weights of the network such that the output generated by the network for the given input $x_d = \left[x_{d1},....,x_{dn}\right]^T$ is as "close" to y_d as possible. Formally, this can be formulated as the following optimization problem:

$$\text{minimize } \frac{1}{2}\sum_{s=1}^{m}\left(y_{ds} - y_s\right)^2,$$

where $y_s, s = 1,....,m,$ are the actual outputs of the neural network in response to the inputs $x_{d1},....,x_{dn}$, as given by

$$y_s = f\left(\sum_{j=1}^{l} w_{sj}^{0} f\left(\sum_{i=1}^{n} w_{ji}^{h} x_i\right)\right).$$

The above minimization is taken over all $w_{ji}^{h}, w_{sj}^{0} \;\; i = 1,....,n, j = 1,....,l, s = 1,....,m.$ For simplicity of notation, we use the symbol w for the vector

$$w = \left\{w_{ji}^{h}, w_{sj}^{0} \;\; : i = 1,....,n,\; j = 1,....,l, s = 1....,m\right\},$$

and the symbol *E* for the objective function to be minimized, that is,

$$E(w) = \frac{1}{2}\sum_{s=1}^{m}\left(y_{ds} - y_s\right)^2$$

$$= \frac{1}{2}\sum_{s=1}^{m}\left(y_{ds} - f\left(\sum_{j=1}^{l} w_{sj}^{0} f\left(\sum_{i=1}^{n} w_{ji}^{h} x_{di}\right)\right)\right)^2.$$

To solve the above optimization problem, we use a gradient algorithm with fixed step size. To formulate the algorithm, we will need to compute the partial derivatives of *E* with respect to each component of *w.* For this, let us first fix the indices *i*, *j*, and *s.* We first compute the partial derivative of *E* with respect to w_{sj}^{0}.

For this, we write

$$E(w) = \frac{1}{2}\sum_{p=1}^{m}\left(y_{dp} - f\left(\sum_{q=1}^{l} w_{pq}^{0} z_q\right)\right)^2,$$

where, for each $q = 1,....,l,$

$$z_q = f\left(\sum_{i=1}^{n} w_{qi}^{h} x_{di}\right).$$

Using the chain rule, we obtain

$$\frac{\partial E}{\partial w_{sj}^{0}}(w) = -\left(y_{ds} - y_s\right) f\left(\sum_{q=1}^{l} w_{sq}^{0} z_q\right) z_j,$$

where $f^{/} : \Re \rightarrow \Re$ is the derivative of f. For simplicity of notation, we write

$$\delta_s = \left(y_{ds} - y_s\right) f\left(\sum_{q=1}^{l} w_{sq}^{0} z_q\right).$$

We can think of each δ_s as a scaled output error, since it is the difference between the actual output y_s of the neural network and the desired output y_{ds}, scaled by $f^{/}\left(\sum_{q=1}^{l} w_{sq}^{0} z_q\right)$. Using the δ_s notation, we have

$$\frac{\partial E}{\partial w_{sj}^{0}}(w) = -\delta_s z_j. \qquad (15)$$

We next compute the partial derivative of *E* with respect to w_{ji}^{h}. We start with the equation

$$E(w) = \frac{1}{2}\sum_{p=1}^{m}\left(y_{dp} - f\left(\sum_{q=1}^{l} w_{pq}^{0} f\left(\sum_{r=1}^{n} w_{qr}^{h} x_{dr}\right)\right)\right)^2.$$

Using the chain rule once again, we get

$$\frac{\partial E}{\partial w_{ji}^{h}}(w) = -\sum_{p=1}^{m}\left(y_{dp} - y_p\right) f^{/}\left(\sum_{q=1}^{l} \delta_{pq} z_q\right) w_{pj}^{o} f^{/}\left(\sum_{r=1}^{n} w_{jr}^{h} x_{dr}\right) x_{di},$$

where $f^{/} : \Re \to \Re$ is the derivative of f. Simplifying the above yields

$$\frac{\partial E}{\partial w_{ji}^{h}}(w) = -\left(\sum_{p=1}^{m} \delta_p w_{pj}^{o}\right) f^{/}\left(v_j\right) x_{di}. \qquad (16)$$

We are now ready to update the weights of the neural network.

Therefore, from the Steepest descent (Gradient descent) method [1], we write the each weight updated equation (for w_1, w_2, ….,w_l weights in the network) using the increment

$$\Delta w_i = -\eta \frac{\partial E}{\partial w_i}$$

i.e.

$$w_{i+1} - w_i = -\eta \frac{\partial E}{\partial w_i} \quad i = 1, 2, \ldots, l \qquad (17)$$

where represents a learning constant η, i.e., a proportionality parameter which defines the step length of each iteration in the negative gradient direction.

From equation (15), (16) and (17), we write the update equations for the two sets of weights w_{sj}^{0} and w_{ji}^{h} separately. We have

$$w_{sj}^{o(k+1)} = w_{sj}^{o(k)} + \eta \delta_s^{(k)} z_j^{(k)} \qquad (18)$$

$$w_{ji}^{h(k+1)} = w_{ji}^{h(k)} + \eta\left(\sum_{p=1}^{m} \delta_s^{(k)} w_{pj}^{o(k)}\right) f^{/}\left(v_j^{(k)}\right) x_{di}, \qquad (19)$$

where η is the (fixed) step size and

$$v_j^{(k)} = \sum_{i=1}^{n} w_{ji}^{h(k)} x_{di}$$

$$z_j^{(k)} = f\left(v_j^{(k)}\right)$$

$$y_s^{(k)} = f\left(\sum_{q=1}^{l} w_{sq}^{o(k)} z_q^{(k)}\right)$$

$$\delta_s^{(k)} = \left(y_{ds} - y_s^{(k)}\right) f'\left(\sum_{q=1}^{l} w_{sq}^{o(k)} z_q^{(k)}\right).$$

The update equation for the weights w_{sj}^{o} of the output layer neurons is illustrated in Figure 5, whereas the update equation for the weights w_{ji}^{h} of the hidden layer neurons is illustrated in Figure 6.

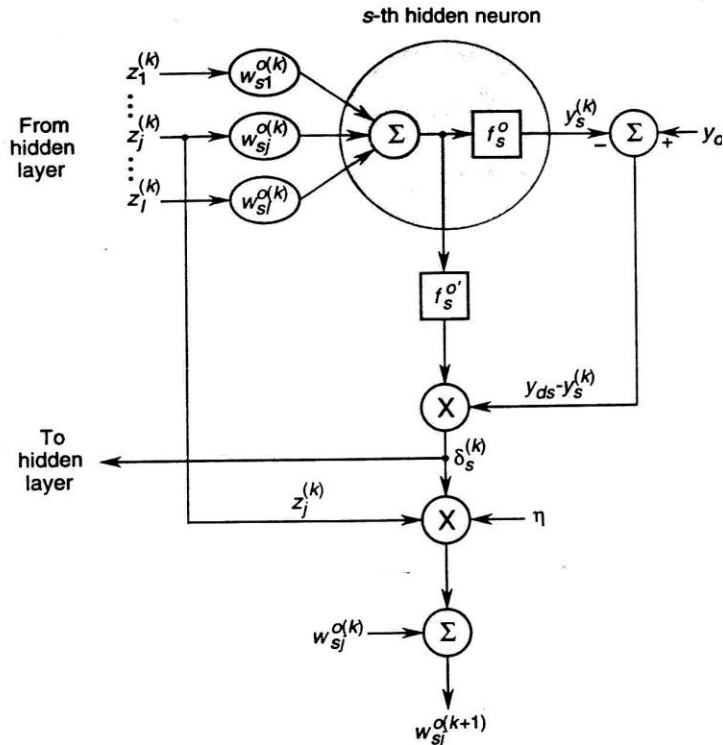


Fig. 5: Illustration of update equation for the output layer weights

The above update equations are referred to in the literature as the *backpropagation algorithm.* The reason for the name "backprogagation" is that the output errors $\delta_1^{(k)},....,\delta_m^{(k)}$ are "propagated back" from the output layer to the hidden layer, and are used in the update equation for the hidden layer weights, as illustrated in Figure 6. In the above discussion, we assumed only a single hidden layer. In general, we may have multiple hidden layers- in this case, the update equations for the weights will resemble the equations derived above. In the general case, the output errors are propagated backward from layer to layer and are used to update the weights at each layer.

We summarize the backpropagation algorithm qualitatively as follows. Using the inputs x_{di} and the current set of weights, we first compute the quantities $v_j^{(k)}$, $z_j^{(k)}$, $y_s^{(k)}$, and $\delta_s^{(k)}$, in turn. This is called the *forward pass* of the algorithm, because it involves propagating the input layer to the output layer. Next, we compute the updated weights using the quantities computed in the

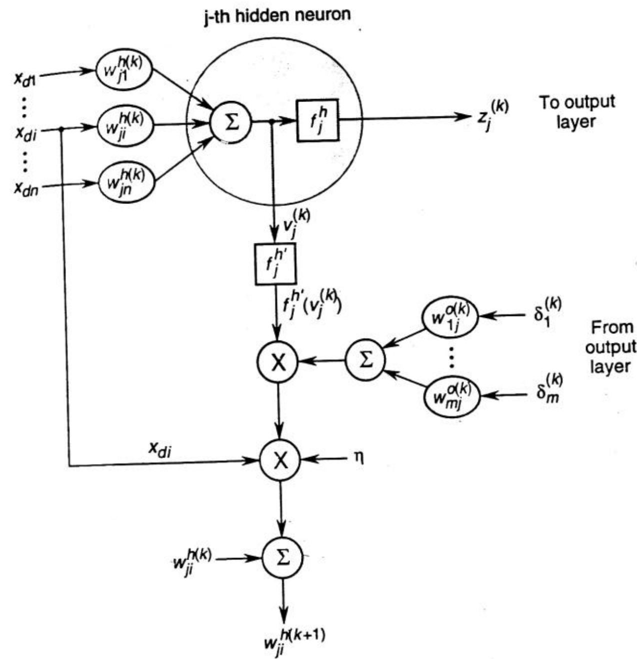


Fig. 6: Illustration of the update equation for the hidden layer weights

forward pass. This is called the *reverse pass* of the algorithm, because it involves propagating the computed output errors $\delta_s^{(k)}$ backward through the network.

5 Numerical Examples

Example 1. Consider the simple feedforward neural network shown in Figure 7. The activation function for all the neurons are given by $f(v)=1/(1+e^{-v})$. This particular activation function has the convenient property that $f^{\prime}(v)=f(v)(1-f(v))$. Therefore, using this property, we can write

$$\begin{aligned}\delta_1 &= (y_d - y_1)f'\left(\sum_{q=1}^{2} w_{1q}^0 z_q\right)\\ &= (y_d - y_1)f\left(\sum_{q=1}^{2} w_{1q}^0 z_q\right)\left(1-f\left(\sum_{q=1}^{2} w_{1q} z_q\right)\right)\\ &= (y_d - y_1)y_1(1-y_1).\end{aligned}$$

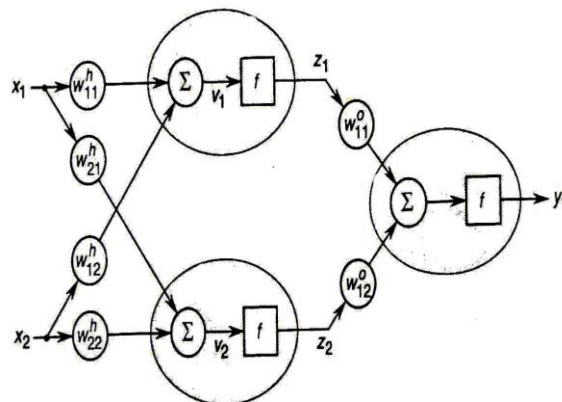


Fig. 7: Simple feedforwarded Neural network

Suppose the initial weights are $w_{11}^{h(0)} = 0.1, w_{12}^{h(0)} = 0.3, w_{21}^{h(0)} = 0.3, w_{22}^{h(0)} = 0.4, w_{11}^{o(0)} = 0.4,$ and $w_{12}^{o(0)} = 0.6.$ Let $x_d = [0.2, 0.6]^T$ and $y_d = 0.7.$ Perform one iteration of the backpropagation algorithm to update the weights of the network. Use a step size of $\eta = 10.$

To proceed, we first compute

$$\begin{aligned} v_1^{(0)} &= w_{11}^{h(0)} x_{d1} + w_{12}^{h(0)} x_{d2} = 0.2 \\ v_2^{(0)} &= w_{21}^{h(0)} x_{d1} + w_{22}^{h(0)} x_{d2} = 0.3 \end{aligned}$$

Next, we compute

$$\begin{aligned} z_1^{(0)} &= f\left(v_1^{(0)}\right) = \frac{1}{1 + e^{-0.2}} = 0.5498 \\ z_2^{(0)} &= f\left(v_2^{(0)}\right) = \frac{1}{1 + e^{-0.3}} = 0.5744 \end{aligned}$$

We then compute

$$y_1^{(0)} = f\left(w_{11}^{o(0)} z_1^{(0)} + w_{12}^{o(0)} z_2^{(0)}\right) = f(0.5646) = 0.6375,$$

which gives an output error of

$$\delta_1^{(0)} = \left(y_d - y_1^{(0)}\right) y_1^{(0)} \left(1 - y_1^{(0)}\right) = 0.01444.$$

This completes the forward pass.

To update the weights, we use

$$\begin{aligned} w_{11}^{o(1)} &= w_{11}^{o(0)} + \eta \delta_1^{(0)} z_1^{(0)} = 0.4794 \\ w_{12}^{o(1)} &= w_{12}^{o(0)} + \eta \delta_1^{(0)} z_2^{(0)} = 0.6830, \end{aligned}$$

and, using the fact that $f'\left(v_j^{(0)}\right) = f\left(v_j^{(0)}\right)\left(1 - f\left(v_j^{(0)}\right)\right) = z_j^{(0)}\left(1 - z_j^{(0)}\right)$, we get

$$\begin{aligned} w_{11}^{h(1)} &= w_{11}^{h(0)} + \eta \delta_1^{(0)} w_{11}^{o(0)} z_1^{(0)} \left(1 - z_1^{(0)}\right) x_{d1} = 0.1029 \\ w_{12}^{h(1)} &= w_{12}^{h(0)} + \eta \delta_1^{(0)} w_{11}^{o(0)} z_1^{(0)} \left(1 - z_1^{(0)}\right) x_{d2} = 0.3086 \\ w_{21}^{h(1)} &= w_{21}^{h(0)} + \eta \delta_1^{(0)} w_{12}^{o(0)} z_2^{(0)} \left(1 - z_2^{(0)}\right) x_{d1} = 0.3042 \\ w_{22}^{h(1)} &= w_{22}^{h(0)} + \eta \delta_1^{(0)} w_{12}^{o(0)} z_2^{(0)} \left(1 - z_2^{(0)}\right) x_{d2} = 0.4127 \end{aligned}$$

Now

$$\begin{aligned} v_1^{(1)} &= w_{11}^{h(1)} x_{d1} + w_{12}^{h(1)} x_{d2} = 0.3881 \\ v_2^{(1)} &= w_{21}^{h(1)} x_{d1} + w_{22}^{h(1)} x_{d2} = 0.4256 \end{aligned}$$

Next, we compute

$$\begin{aligned} z_1^{(1)} &= f\left(v_1^{(1)}\right) = \frac{1}{1 + e^{-0.3881}} = 0.5948 \\ z_2^{(1)} &= f\left(v_2^{(1)}\right) = \frac{1}{1 + e^{-0.4256}} = 0.6048 \end{aligned}$$

We then compute

$$y_1^{(1)} = f\left(w_{11}^{o(1)} z_1^{(1)} + w_{12}^{o(1)} z_2^{(1)}\right) = f(0.9202) = 0.7150,$$

which gives an output error of

$$\delta_1^{(1)} = \left(y_d - y_1^{(1)}\right) y_1^{(1)} \left(1 - y_1^{(1)}\right) = 0.02$$

Finally, we have $y_d - y_1^{(0)} = 0.12$, $y_d - y_1^{(1)} = 0.09$ and hence $\left|y_d - y_1^{(1)}\right| < \left|y_d - y_1^{(0)}\right|$

After 15 iterations of the backpropagation algorithm, we get

$$w_{11}^{0(15)} = 0.6365$$
$$w_{12}^{0(15)} = 0.8474$$
$$w_{11}^{h(15)} = 0.1105$$
$$w_{12}^{h(15)} = 0.3315$$
$$w_{21}^{h(15)} = 0.3146$$
$$w_{22}^{h(15)} = 0.4439$$

The resulting value of the output corresponding to the input x_d = [0.2,0.6] is $y_1^{(15)} = 0.6997$

In the above example, we considered an activation function of the form

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

The above function is called *sigmoid* (previously discussed), and is a popular activation function used in practice. It is possible to use a general version of the sigmoid function, of the form

$$g(v) = \frac{\beta}{1+e^{-(v-b)}}$$

The parameters β and b represent scale and shift parameters, respectively. The parameter b is often interpreted as a bias. If such activation function is used in a neural network, we would also want to adjust the values of the parameters β and b, which also effect the value of the objective function to be minimized. However, it turns out that these parameters can be incorporated into the backpropagation algorithm simply by treating them as additional weights in the network. Specifically, we can represent a neuron with activation function g as one with activation function *f* with the addition of two extra weights as shown in Figure 8.

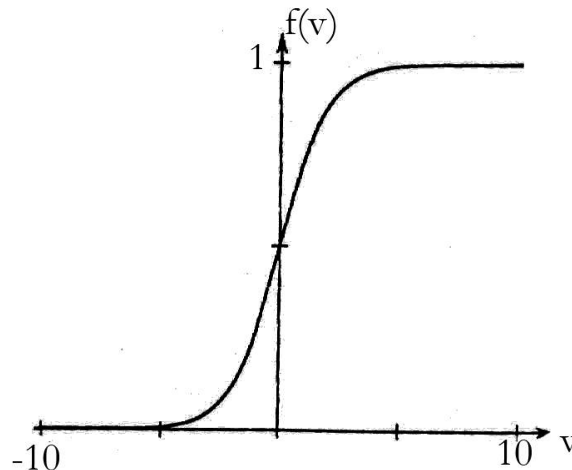


Fig. 8: The sigmoid function

Example 2. Consider the same neural network as shown in Example 1. We introduce shift parameters b_1, b_2, and b_3 to the activation functions in the neurons. Using the configuration illustrated in Figure 9, we can incorporate the shift parameters into the backpropagation algorithm. We have

$$v_1 = w_{11}^h x_{d1} + w_{12}^h x_{d2} - b_1$$
$$v_2 = w_{21}^h x_{d1} + w_{22}^h x_{d2} - b_2$$
$$z_1 = f(v_1)$$
$$z_2 = f(v_2)$$
$$y_1 = f(w_{11}^o z_1 + w_{12}^o z_2 - b_3)$$
$$\delta_1 = (y_d - y_1) y_1 (1 - y_1),$$

where f is the sigmoid function

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

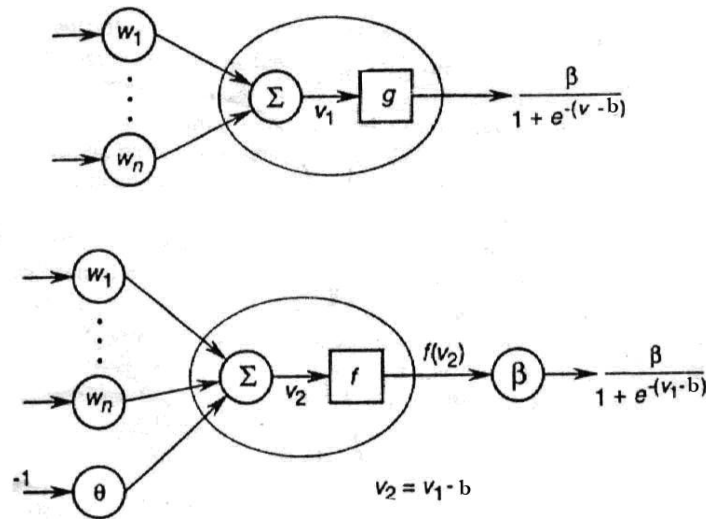


Fig. 9: These two configurations are equivalent

The components of the gradient of the objective function E with respect to the shift parameters are

$$\frac{\partial E}{\partial b_1}(w) = \delta_1 w_{11}^0 z_1 (1 - z_1)$$
$$\frac{\partial E}{\partial b_2}(w) = \delta_1 w_{12}^0 z_2 (1 - z_2)$$
$$\frac{\partial E}{\partial b_3}(w) = \delta_1.$$

In the next example, we apply the network discussed in Example 2 to solve the celebrated Exclusive OR (XOR) problem.

Example 3. Consider the neural network of Example 2. We wish to train the neural network to approximate the Exclusive OR (XOR) function, defined in Table 1. Note that the XOR function has two inputs and one output. To train the neural network, we use the following training pairs:

$$x_{d,0} = [0, 0]^T, \quad y_{d,0} = 0$$

$$x_{d,1} = [0, 1]^T, \quad y_{d,1} = 1$$

$$x_{d,3} = [1, 0]^T, \quad y_{d,2} = 1$$

$$x_{d,4} = [1, 1]^T, \quad y_{d,3} = 0.$$

Table 1. Truth Table for XOR Function

x_1	x_2	$F(x_1,x_2)$
0	1	0
0	1	1
1	0	1
1	1	0

We now apply the backpropagation algorithm to train the network using the above training pairs. To do this, we apply the above pairs one per iteration, in cycle fashion. In other words, in the kth iteration of the algorithm, we apply the pair $(x_{d,R(k)}, y_{d,R(k)})$, where, as in Kaczmarz's algorithm, R(k) is the unique integer in {0, 1, 2,3} satisfying k = 4l + R(k) for some integer l, that is, R(k) is the remainder that results if we divide k by 4.

The experiment yields the following weights

$$\begin{aligned} w_{11}^{0} &= -11.01 \\ w_{12}^{0} &= 10.92 \\ w_{11}^{h} &= -7.777 \\ w_{12}^{h} &= -8.403 \\ w_{21}^{h} &= -5.593 \\ w_{22}^{h} &= -5.638 \\ b_1 &= -3.277 \\ b_2 &= -8.357 \\ b_3 &= 5.261. \end{aligned}$$

Table 2. shows the output of the network with the above weights corresponding to the training input data.

Table 2 Response of the Trained Network for Example 3.

x_1	x_2	y
0	0	0.007
0	1	0.99
1	0	0.99
1	1	0.009

6 Results

A simple but effective description on feedforward artificial neural networks has been made in here giving emphasize on the backpropagation algorithm, since it is widely used and many other

algorithms are derived from it. We have implemented the first-order optimization method in single and multiple-layer feedforward artificial neural networking problem and set-up the numerical test examples in three different cases. The convergence behaviors of our examples for weight, scale and shift parameters $w, \beta, \text{and } b$, respectively in few iterations shows that the results of the actual network output is as “close” to our desired(target) output.

Acknowledgement

The University of Chittagong, for providing the financial support and a stimulating environment for research.

References

[1] D.P. Bertsekas (1982),” Constrained Optiization and Lagrange Multiplier Methods”, Academy Press, New York.

[2] D. Rumelhart, G. Hinton and R. J. Williams (1986), “Learning internal representations by error propagation”, in Parallel Distributed Processing, MIT Press, Cambridge.

[3] Hornik, K., Stinchcombe, M. and White, H.(1989), “Multilayer Feedforward Networks are Universal Approximators,” *Neural Networks*, **2**, 359-366.

[4] Hanson, S. J. and Pratt, L(1989)., “Comparing biases for minimal network construction with back-propagation”, *Advances in NIPS*, **1** , 177–185.

[5] Karnin, E. D. (1990), “A simple procedure for pruning back-propagation trained neural networks”, *IEEE ICNN’90* , **1** , 239–242.

[6] K. Hornik, (1991), “Approximation capabilities of multilayer feedforward networks,” *Neural Networks,* **4**, 251—257.

[7] Chen X.- H.Y.G.- A. (1992), “Efficient backpropagation learning using optimal learning rate and momentum.”, *Neural Network*., **10**(3), 517–527.

[8] R. Battiti (1992), “First and second-order methods for learning: Between steepest descent and newton's methods," *Neural Computation* **4**, 141-166.

[9] Gori, M.& Tesi, A. (1992), “On the problem of local minima in backpropagation”, *IEEE Trans. Patterns Analysis and Machine Intelligence*, **14**, 76-85.

[10] Azimi-Sadjadi M.R. and Liou R.J. (1992), “ Fast learning process of multi-layer neural network using recursive least squares method.”, *IEEE Trans. Signal Process*, **40**(2), 443–446.

[11] Karayiannis and A. N. Venetsanopoulos, (1993), “Artificial neural networks: learning algorithms, performance evaluation, and applications”, Kluver Academic, Boston, MA.

[12] Karayiannis N.B. and Venetsanopoulos A.N. (1993), “Efficient Learning Algorithms for Neural Networks (ELEANNE)”., *IEEE Trans. Syst. Man Cybern.,* **23**(5), 1372–1383.

[13] Bilski J. (1995), “Fast learning procedures for neural networks.”, Ph.D. Thesis, AGH University of Science and Technology, (in Polish).

[14] O.S.P. and Stodolski M. (1996), “*Fast second-order learning algorithm for feedforward multilayer neural networks and its application”*, *Neural Network*., **9**(9), 1583–1596.

[15] Shang, Y.& Wah, B.W. (1996), “Global optimization for neural network training”, *IEEE Computer*, **29**(3), 45-54

[16] S. Tamura and M. Tateishi, (1997). “Capabilities of a four-layered feedforward neural network: Four layers versus three”, *IEEE Transactions on Neural Networks*, **8**(2), 251—255.

[17] Bilski J. and Rutkowski L. (1998)., “A fast training algorithm for neural networks”, *IEEE Trans. Circuits Syst. II*, Vol. **45**(6), 749–753.

[18] Abid S., Fnaiech F. and Najim M. (2001), “A fast feedforward training algorithm using a modified form of the standard backpropagation algorithm”, *IEEE Trans. Neural Network*, **12** (2), 424–434.

[19] Leung Ch.S., Tsoi Ah.Ch. and Chan L. W. (2001), “Two regularizers for recursive least squared algorithms in feedforward multilayered neural networks”, *IEEE Trans. Neural Netw.* **12**(6), 1314–1332.

[20] Ampazis N. and Perantonis J. (2002), “Two highly efficient second-order algorithms for training feedforward networks”, *IEEE Trans. Neural Network*, **13** (5), 1064–1074.

[21] G.-B. Huang (2003), “Learning capability and storage capacity of two-hidden-layer feedforward networks,” *IEEE Transactions on Neural Networks,* **14**(2), 274—281.