Feed-forward Neural Networks

1 Introduction

The development of layered feed-forward networks began in the late 1950's, represented by Rosenblatt's **perceptron** and Widrow's ADaptive LINear Element **(ADLINE)**

Both the perceptron and ADLINE are single layer networks and are often referred to as single layer perceptrons.

Single layer perceptrons can only solve **linearly separable** problems.

The limitations of the single layer network has led to the development of multi-layer feed-forward networks with one or more hidden layers, called multi-layer perceptron (MLP) networks. MLP networks overcome many of the limitations of single layer perceptrons, and can be trained using the backpropagation algorithm. The backpropagation technique was invented independently several times.

In 1974, Werbos developed a backpropagation training algorithm. However, Werbos' work remained almost unknown in the scientific community, and in 1985, Parker rediscovered the technique. Soon after Parker published his findings, Rumelhart, Hinton and Williams also rediscovered the technique. It is the efforts of Rumelhart and the other members of the Parallel Distributed Processing (PDP) group, that make the backpropagation technique a mainstay of neurocomputing.

To date, backpropagation networks are the most popular neural network model and have attracted most research interest among all the existing models.

2 Single Layer Perceptron

The single layer perceptron was first devised by Rosenblatt in the late 1950's and early 1960's. The basic model of a perceptron capable of classifying a pattern into one of two classes is shown in Fig. 1.

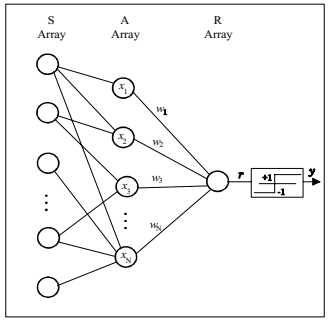


Figure 1 A basic perceptron model

The machine consists of an array *S* of *sensory units* which are randomly connected to a second array *A* of *associative units*. Each of these units produces an output only if enough of the sensory units which are connected to it are activated, that is, the output signals of the associative units are binary.

The sensory units can be viewed as the means by which the machine receives stimuli from its external environment. The outputs of the associative units are the input to the perceptron.

The response of the machine is proportional to the weighted sum of the outputs of the associative units; i.e., if x_i denotes the output signal of the *i*th associative unit and w_i the corresponding weight, the response is given by

$$r = \sum_{i=1}^{N} w_i x_i \qquad (1)$$

and this response signal is passed through a hard limiting non-linearity to produce the output of the machine.

$$y = \begin{cases} +1 & if \quad r \geq 0 \\ -1 & if \quad r < 0 \end{cases}$$

An effective technique for analysing the behaviour of the perceptron network shown in Fig. 1 is to plot a map of the decision regions created in the multidimensional space spanned by the input variables. The perceptron network of Fig. 1 forms the decision regions separated by a hyperplane defined by

$$\sum_{i=1}^{N} w_i x_i = 0 \qquad (2)$$

As can be seen from (2), the decision boundary is determined by the connection weights of the network.

Figure 2 shows an example of the decision regions created by the perceptron network for two-dimensional input vectors. In this case, the hyperplane is a line. These decision regions classify the input patterns as belonging to one of the two classes. In Fig. 2, inputs above the boundary line lead to a class 1 ($y >= 0$) response, and input patterns below the boundary line lead to a class 2 ($y < 0$) response.

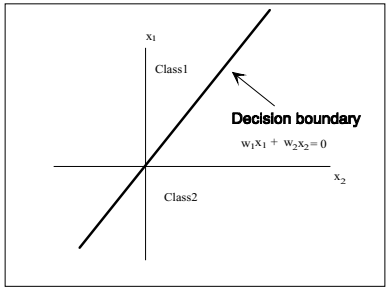


Figure 2. Decision regions formed by the perceptron network

2.1 The Perceptron Training Algorithm

The training algorithm for the perceptron network of Fig. 1 is a simple scheme for the iterative determination of the weight vector *W*. This scheme, known as the perceptron convergence procedure, can be summarised as follows.

The initial connection weights are set to small random non-zero values. A new input pattern is then applied and the output is computed as

$$y(n) = f\left(\sum_{i=1}^{i=N} w_i(n) x_i(n)\right) \qquad (3)$$

where $f(x) = \begin{cases} +1 & if \quad x \geq 0 \\ -1 & if \quad x < 0 \end{cases}$ is the hard limiting non-linearity and n is the iteration index.

Connection weights are updated according to

$$w_i(n+1) = w_i(n) + \eta\big(d(n) - y(n)\big)x_i(n)\,,\ i = 1,2,\ldots,N \qquad (4)$$

where η is a positive gain factor less than 1

$$d(n) = \begin{cases} +1 & \text{if input is from class 1} \\ -1 & \text{if input is from class 2} \end{cases}$$

The perceptron convergence procedure does not adapt the weights if the output decision is correct.

If the output decision disagrees with the binary desired response $d(n)$, however, adaptation is effected by adding the weighted input vector to the weight vector when the error is positive, or subtracting the weighted input vector from the weight vector when the error is negative.

The perceptron convergence procedure is terminated when the training patterns are correctly separated.

Figure 3 shows an example of the use of the perceptron convergence procedure. Samples from class 1 are represented by circles in the figure, and samples from class 2 are represented by crosses. Samples from class 1 and class 2 were presented alternately. The four lines show the four decision boundaries after the weights had been adapted following errors on iterations 0, 2, 4, and 80. In this example it can be seen that the classes were well separated after only four iterations.

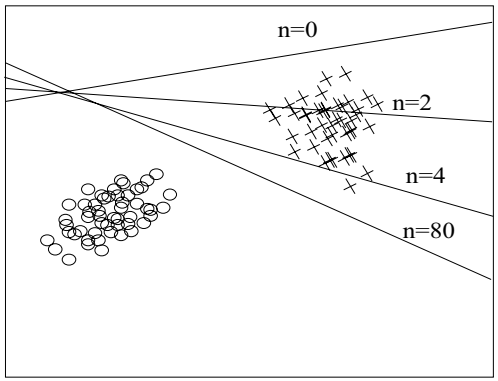


Figure 3 An example of the perceptron convergence process

Rosenblatt proved that if the inputs presented are separable into two classes, the perceptron convergence procedure converges and positions the decision hyperplane between those two classes. One problem with the perceptron convergence procedure is that the decision boundaries may oscillate continuously when inputs are not separable and distributions overlap.

3 The ADLINE and Widrow-Hoff Algorithm

The adaptive linear element (ADLINE) is a simple type of processing element that has a real vector X as its input and a real number y as its output (Fig. 4) and uses the Widrow-Hoff training algorithm.

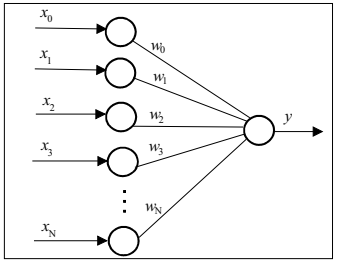


Figure 4 Adaptive Linear Element (ADLINE)

The input to the ADLINE is $X = (x_0, x_1, \ldots, x_N)$, where x_0 is the bias input, set to a constant value (usually x_0=1). The output of the ADLINE is the inner product of the input vector X and the weight vector $W = (w_0, w_1, w_2, \cdots, w_N)$, that is

$$y = x_0 w_0 + x_1 w_1 + \cdots + x_n w_N$$

The weight vector W defines a hyperplane in the N-dimensional space of input vectors X, as shown in Fig. 5.

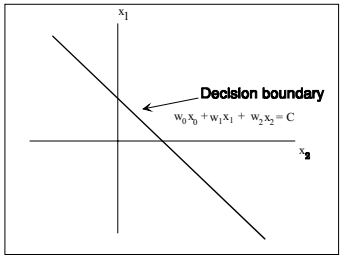


Figure 5 ADLINE decision boundary in the input space.For all input vectors X that fall in the area above the decision hyperplane, the ADLINE output $y > C$. For all input vectors X that fall in the area below the hyperplane, the ADLINE output $y < C$.

The hyperplane is given by

$$x_0 w_0 + x_1 w_1 + \cdots + x_n w_N = C$$

where C is a constant number.

Cost Function of the ADLINE

For each input vector $X(n)$ to the ADLINE, there exists a corresponding target output (desired output) $d(n)$. The cost function of the ADLINE is defined as

$$E = \frac{1}{2}\sum_{n=1}^{P}(d(n) - y(n))^2 \qquad (5)$$

where P is the number of training vectors, and $y(n)$ is the actual output of ADLINE for the nth training vector $X(n)$

The objective of training is to find a weight vector W^* that minimises the cost function

3.1 The Widrow-Hoff Training Algorithm

In 1959, Bernard Widrow, along with his student Macron E. Hoff, developed an algorithm for finding the weight vector W^*. This algorithm is the well known Widrow-Hoff algorithm, also known as the **LMS law** and the **delta rule**.

The method for finding W^* is to start from an initial value of W and then 'slide down' the ADLINE cost function surface until the bottom of the surface is reached. Since the cost function is a quadratic function of the weights, the surface is convex and has a unique (global) minimum.

The basic principle of the Widrow-Hoff algorithm is a gradient descent technique and has the form of

$$w_i(n+1) = w_i(n) - \eta \frac{\partial E}{\partial w_i} \tag{6}$$

we have

$$\begin{aligned} \frac{\partial E}{\partial w_i} &= \frac{1}{2}\sum_{n=1}^{P} 2\big(d(n) - y(n)\big)\left(-\frac{\partial y(n)}{\partial w_i}\right) \\ &= \sum_{n=1}^{P} \big(d(n) - y(n)\big)\big(-x_i(n)\big) \\ &= -\sum_{n=1}^{P} \delta(n) x_i(n) \end{aligned} \tag{7}$$

where $\delta(n) = d(n) - y(n)$.

Instead of computing the true gradient using equation (7), the Widrow-Hoff algorithm uses the instantaneous gradient which is readily available from a single input data sample, and the Widrow-Hoff training algorithm is given by

$$w_i(n+1) = w_i(n) + \eta \delta(n) x_i(n) \tag{8}$$

where η is the training constant.

The training constant η determines the stability and convergence rate, and is usually chosen by trial and error. If η is too large, the weight vector will not converge; if η is too small, the rate of convergence will be slow.

4 Multi-layer Feed-forward Networks and the Backpropagation Training Algorithm

In the previous two sections, networks with only input and output units were described. These networks have proved useful in a wide variety of applications. The essential character of such networks is that they map similar input patterns to similar output patterns. This is why such networks can do a relatively good job in dealing with patterns that have never been presented to the networks. However the constraint that similar input patterns lead to similar outputs is also a limitation of such networks. For many practical problems, very similar input patterns may have very different output requirements. In such cases, the networks described in sections 2.2 and 2.3 may not be able to perform the necessary mappings.

Minsky and Papert pointed out that such networks cannot even solve the *exclusive-or* (XOR) problem illustrated in the table below.

XOR TRUTH TABLE

Input patterns	outputs
0 0	0
0 1	1
1 0	1
1 1	0

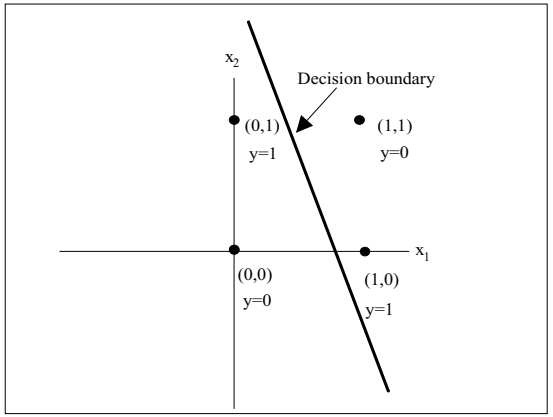


Figure 6 Single layer feed-forward network is incapable of solving the XOR problem

To see this in a more straightforward way, we recall from the above sections that single layer networks form a hyperplane that separates the *N*-dimensional Euclidean space of input vectors. In the case of the XOR problem, the input vectors are two-dimensional and the hyperplane (which is determined by the weights of the network) is a straight line. As can be seen from Fig. 6, this line should divide the space such that the points (0,0) and (1,1) lie on one side and the points (0,1) and (1,0) lie on the other side of the line. This is clearly impossible for a single layer network.

To overcome the limitations of single layer networks, multi-layer feed-forward networks can be used, which not only have input and output units, but also have hidden units that are neither input nor output units. A three layer feed-forward network with one hidden layer is shown in Fig. 7.

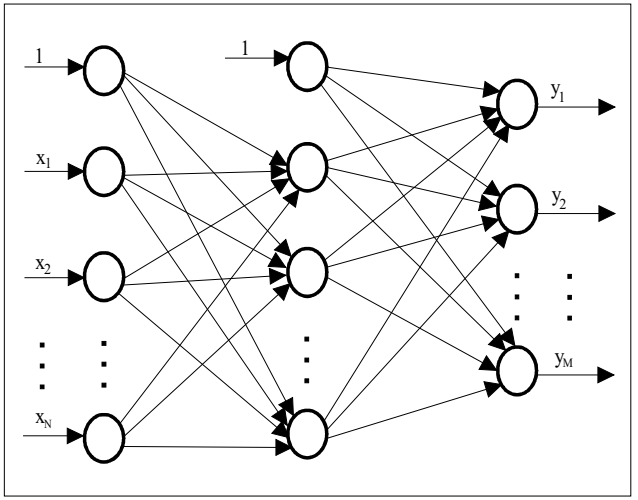


Figure 7 A three layer feed-forward network

Multi-layer networks overcome many of the limitations of single layer networks, but were generally not used in the past (before mid 1980s) because an effective training algorithm was not available. With the publication of the backpropagation training algorithm by Rumelhart, Hinton and Williams in the mid-1980's, multi-layer feed-forward networks, some times called multi-layer perceptron (MLP) networks have become a mainstay of neural network research. In September 1992, Professor B. Widrow from Stanford University told the delegates of the 1992 European Neural Network Conference that "three quarters of the neural network researchers in the USA work on backpropagation networks".

The capabilities of multi-layer networks stem from the non-linearities used with the units. Each neuron in the network receives inputs from other neurons in the network, or receives inputs from the outside world. The outputs of the neurons are connected to other neurons or to the outside world. Each input is connected to the neurons by a

weight. The neuron calculates the weighted sum of the inputs (called the activation), which is passed through a non-linear transfer function to produce the actual output for the neuron. The most popular non-linear transfer function is of the sigmoidal type.

A typical sigmoid function has the form:

$$f(x) = \frac{1}{1 + e^{-gx}} \qquad (4.1)$$

When g become large, the sigmoid function become a signum function as shown in Fig. 8

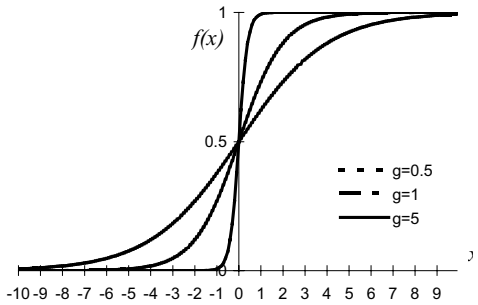


Figure 8 Sigmoid function used in multi-layer feed-forward networks

The introduction of one hidden layer allows the network to represent an arbitrary Boolean function, and the use of two hidden layers allows the network to represent an arbitrary decision space. The hidden units also enable networks to automatically represent geometrical invariance such as translation invariance.

A theorem which states that the backpropagation network is able to implement any function of practical interest to any desired degree of accuracy was proven by Hecht-Nielsen and is expressed below.

Backpropagation Network Function Approximation Theorem: *Given any* $\varepsilon > 0$ *and any* L_2 *function* f: $[0,1]^N \rightarrow R^M$, *there exists a three-layer* (*with two hidden layers*) *backpropagation network that can approximate* f *to within* ε *mean squared error accuracy.*

The above theorem guarantees the ability of a multi-layer network with the correct weights to accurately implement an arbitrary L_2 function. It ***does not*** state how the weights should be selected or even whether these weights can be found using existing network learning algorithms.

Notwithstanding the fact that the backpropagation networks are not guaranteed to be able to find the *correct* weights for any given task, they have found numerous applications in a variety of problems. Sejnowski and Rosenberg have demonstrated that backpropagation networks can learn to convert text to realistic speech. Burr has shown that backpropagation networks can be used for recognition of spoken digits and hand-written characters with excellent performance. Backpropagation networks can also be used for data compression by forcing the output to reproduce the input for a network with a lower-dimensional hidden layer, and many many more …

4.1 Backpropagation Training Algorithm

The backpropagation training algorithm is an extension of the Widrow-Hoff algorithm. It uses a gradient descent technique to minimise a cost function equal to the mean squared difference between the desired and the actual network outputs. The backpropagation training algorithm proposed by Rumelhart *et al*. involves the presentation of a set of pairs of input and output patterns. The network first uses the input vector to produce its own output vector (actual network output) and then compares this actual output with the desired output, or target vector. If there is no

difference, no training takes place, otherwise the weights of the network are changed to reduce the difference between actual and desired outputs.

Cost Function of Backpropagation Network: The cost function that the backpropagation network tries to minimise is the squared difference between the actual and desired output value summed over the output units and all pairs of input and output vectors.

Let $$E(n)=\frac{1}{2}\sum_{j=1}^{M}(d_j(n)-o_j(n))^2 \qquad (4.2)$$

be a measure of error on input/output pattern *n*

where $d_j(n)$ is the desired output for the *j*th component of the output vector for input pattern *n*, *M* is the number of output units,and $o_j(n)$ is the *j*th element of the actual output vector produced by the presentation of input pattern *n*.

Let $$s_{hj}(n)=\sum_{i=1}^{N}w_{hji}x_i(n) \qquad (4.3)$$

be the weighted sum input to unit *j* in the hidden layer produced by the presentation of input pattern *n*,

where w_{hji} is the weight connecting input unit *i* and hidden unit *j,* and *N* is the number of input units.

Similarly, let $$s_{oj}(n)=\sum_{i=1}^{H}w_{oji}o_{hi}(n) \qquad (4.4)$$

be the weighted sum input to unit *j* in the output layer produced by the presentation of input pattern *n*, where *H* is the number of hidden units, $o_{hi}(n)$ is the output of hidden

unit *i* produced by the presentation of input pattern *n*,and w_{oji} is the weight connecting hidden unit *i* and output unit *j.*

The outputs of the hidden units and output units are, respectively,

$$o_{hj}(n)=f(s_{hj}(n)) \qquad (4.5)$$

$$o_j(n)=f(s_{oj}(n)) \qquad (4.6)$$

where f is a differentiable and non-decreasing non-linear transfer function.

Let $$E=\sum_{n=1}^{P}E(n) \qquad (4.7)$$

be the overall measure of error, where P is the total number of the training samples. E is called the ***Cost Function*** of the backpropagation network. The backpropagation algorithm is the technique which finds the weights that minimise the cost function E.

Gradient Descent Technique: The computational method used with the backpropagation training algorithm in attempting to find the *correct* weight values is a gradient descent technique. Common to all gradient search techniques is the use of the gradient, in this case, the gradient of the cost function $\nabla E=\left(\partial E/\partial w_1,\partial E/\partial w_2,\cdots,\partial E/\partial w_L\right)$, where L is the total number of weights in the network. In the process of training the network, only the discrete approximation to the true gradient of E can be obtained and used.

Following the presentation of each pattern to the network, the weights can be updated according to

$$w_{ij}(n+1) = w_{ij}(n) - \eta \frac{\partial E(n)}{\partial w_{ij}} \tag{4.8}$$

Alternatively, following the presentation of a complete cycle of patterns the weights can be updated according to

$$w_{ij}(n+1) = w_{ij}(n) - \eta \frac{\partial E}{\partial w_{ij}} \tag{4.9}$$

where $w_{ij}(n)$ is the value of w_{ij} before updating, $w_{ij}(n+1)$ is the value of w_{ij} after updating, and η is the learning rate which determines the convergence rate and stability of the training process.

The above two Equations are is called the *on-line* training mode and *batch* training mode. In training the networks, either equation may be used to achieve almost the same results.

The Chain Rule for Calculating $\frac{\partial E(n)}{\partial w_{oji}}$ and $\frac{\partial E(n)}{\partial w_{hji}}$

First we compute $\frac{\partial E(n)}{\partial w_{oji}}$

We can write

$$\frac{\partial E(n)}{\partial w_{oji}} = \frac{\partial E(n)}{\partial s_{oj}(n)} \frac{\partial s_{oj}(n)}{\partial w_{oji}} \tag{4.10}$$

According to equation(4.4) we have

$$\frac{\partial s_{oj}}{\partial w_{oji}} = \frac{\partial \left(\sum_{i=1}^{H} w_{oji} o_{hi} \right)}{\partial w_{oji}} = o_{hi}(n) \tag{4.11}$$

Now let us define

$$\delta_{oj}(n) = -\frac{\partial E(n)}{\partial s_{oj}(n)} \tag{4.12}$$

To compute $\delta_{oj}(n)$, we apply the chain rule to write the partial derivative as the product of two factors. Thus

$$\delta_{oj}(n) = -\frac{\partial E(n)}{\partial s_{oj}(n)} = -\left(\frac{\partial E(n)}{\partial o_j(n)} \right) \left(\frac{\partial o_j(n)}{\partial s_{oj}(n)} \right) \tag{4.13}$$

From equation (4.6) we have

$$\frac{\partial o_j(n)}{\partial s_{oj}(n)} = f'(s_{oj}(n)) \tag{4.14}$$

To compute the first factor in equation (4.13), following the definition of $E(n)$ in equation (4.2), we have

$$\frac{\partial E(n)}{\partial o_j(n)} = -(d_j(n) - o_j(n)) \tag{4.15}$$

Substituting for the two factors in equation (4.13), we have

$$\delta_{oj}(n) = (d_j(n) - o_j(n)) f'(s_{oj}(n)) \tag{4.16}$$

combining equations (4.10), (4.11) and (4.16), we have

$$\frac{\partial E(n)}{\partial w_{oji}} = -o_j(n)(d_j(n) - o_j(n)) f'(s_{oj}(n)) \tag{4.17}$$

Now to compute $\frac{\partial E(n)}{\partial w_{hji}}$, we can write

$$\frac{\partial E(n)}{\partial w_{hji}} = \frac{\partial W(n)}{\partial s_{hj}(n)} \frac{\partial s_{hj}(n)}{\partial w_{hji}} \tag{4.18}$$

According to equation (4.5) we have

$$\frac{\partial s_{hj}(n)}{\partial w_{hji}} = \frac{\partial \left(\sum_{i=1}^{N} w_{hji} x_i(n) \right)}{\partial w_{hji}} = x_i(n) \tag{4.19}$$

Define $$\delta_{hj}(n) = -\frac{\partial E(n)}{\partial s_{hj}(n)} \tag{4.20}$$

Using the chain rule, we have

$$\delta_{hj}(n) = f'(s_{hj}(n)) \sum_{k=1}^{M} \delta_{ok}(n) w_{okj} \tag{4.21}$$

From equations (4.19), (4.20) and (4.21), we have

$$\frac{\partial E(n)}{\partial w_{hji}} = -x_i(n) f'(s_{hj}(n)) \sum_{k=1}^{M} \delta_{ok}(n) w_{okj} \tag{4.22}$$

From equation (4.7), we have

$$\frac{\partial E}{\partial w_{oij}} = \sum_{n=1}^{P} \frac{\partial E(n)}{\partial w_{oij}} \tag{4.23}$$

and

$$\frac{\partial E}{\partial w_{hij}} = \sum_{n=1}^{P} \frac{\partial E(n)}{\partial w_{hij}} \tag{4.24}$$

Depending on whether the units are in the output layer or the hidden layer, the partial derivative is calculated according to equations (4.17) and (4.24), or equations (4.22) and (4.23), respectively. The backpropagation training can be summarised in the following steps, which are executed iteratively until the cost function *E* has decreased to an acceptable value:

1. Initialise weights to small random values
2. Present input and desired output
3. Calculated the partial derivative of the weights
4. Adapt weights according to equation (4.8) or (4.9)
5. Repeat by going to 2

BACKPROPAGATION TRAINING ALGORITHM

Backpro_proc() Begin

```
////////////////////////////////////////////////////////////////////
//Define Input, output, error, gradient, bias, and weight vectors  //

float w[2][64][64],e[64],eh[64],h[64],o[64],y[64],
H[64],b[2][64], input[64];

//    w[0][[i][j]       ith inpu to jth hidden neuron
//    w[1][i][j]        ith hidden to jth output neuron
//    b[0][i]           Bias of ith hidden neuron
//    b[1][i]           Bias of the ith output neuron
//    e[i]              Error information for ith output neuron
//    eh[i]             Error information for ith hidden neuron
//    h[i]              weighted sum of ith hidden neuron
//    H[i]              Output of the ith hidden neuron
//    o[i]              weighted sum of the ith output neuron
//    y[i]              Output of the ith output neuron
//    input[i]          ith component of the input vector

////////////////////////////////////////////////////////////////////

//Step 1//
//Initial wights//

      srand(seed);
      for(l=0;l<2;l++)
         {
         for(i=0;i<64;i++)
            {
            for(j=0;j<64;j++)
             {
                  w[l][i][j]=(float)(random(2400)-1200.0)/5000;
             }
            }
         }

      for(l=0;l<2;l++)
       {
       for(i=0;i<64;i++)
          {
            b[l][i]=(float)(random(2400)-1200.0)/10000.0;
          }
       }

////////////////////////////////////////////////////////////////////

//Step 2
//hidden layer output//

      for(k=0;k<h_node;k++) //h_node = #of hidden nodes//
        {
        h[k]=0;
        for(i=0;i<INPUT_VECTOR_SIZE;i++)
                                  //INPUT_VECTOR_SIZE #input nodes
           {
           h[k]+=input[i]*w[0][i][k];
           }
        h[k]+=b[0][k];
        H[k]=sigmoid(h[k]);
        }
```



```
//output layer output//

      for(k=0;k<OUTPUT_VECTOR_SIZE;k++)
                              //OUTPUT_VECTOR_SIZE # of output nodes
         {
         o[k]=0;
         for(i=0;i<h_node;i++)
            {
            o[k]+=H[i]*w[1][i][k];
            }
         o[k]+=b[1][k];
         y[k]=sigmoid(o[k]);
         }

////////////////////////////////////////////////////////////////////

Step 3

//error information//

//output layer//

   for(i=0;i<OUTPUT_VECTOR_SIZE;i++)
       {
       e[i]=grad(o[i])*(output[i]-y[i]);
       }

//hidden layer//

     for(i=0;i<h_node;i++)
       {
       eh[i]=0;
       for(k=0;k<OUTPUT_VECTOR_SIZE;k++)
         {
         eh[i]+=e[k]*w[1][i][k];
         }
       eh[i]=grad1(h[i])*eh[i];
       }

////////////////////////////////////////////////////////////////////

Step 4
//update weights//


//output layer//

      for(i=0;i<h_node;i++)
         {
         for(j=0;j<OUTPUT_VECTOR_SIZE;j++)
            {
            w[1][i][j]=w[1][i][j]+A*H[i]*e[j]; //A = training rate, η
            }
         }

      for(j=0;j<OUTPUT_VECTOR_SIZE;j++)
        {
         b[1][j]=b[1][j]+A*e[j];
        }
```

```
//hidden layer//

     for(i=0;i<INPUT_VECTOR_SIZE;i++)
      {
      for(j=0;j<h_node;j++)
         {
         }
      }

     for(j=0;j<h_node;j++)
      {
      b[0][j]=b[0][j]+A*eh[j];
      }
////////////////////////////////////////////////////////////////

//Repeat By going to Step 2//
```

Backpro_proc() End

```
//First order derivertive of the Sigmoid function//

      float grad(x)
      float x;
      {
      float h;
      h=2*exp(-x)/pow((1+exp(-x)),2.0);
      return(h);
      }

//Sigmoid Function//

      float sigmoid(x)
      float x;
      {
      float h;
      h=(1-exp(-x))/(1+exp(-x));
      return(h);
      }

//First order derivertive of liner transfer function//

      float grad(x)
      float x;
      {
      float h;
      h=1;
      return(h);
      }

//Sigmoid Function//

      float sigmoid(x)
      float x;
      {
      float h;
      h=x;
      return(h);
      }
```

Some traning examples

1. Exclusive-Or (XOR) Task

The network consists of two input units, two hidden units, and one output unit.

2. 8-3-8 Encoder Task :

The network consists of eight input units, three hidden units, and eight output units.

3 10-5-10 Encoder Task:

The network consists of ten input units, five hidden units, and ten output units.

4 10-5-10 Complement Encoder Task :

8-3-8 ENCODER

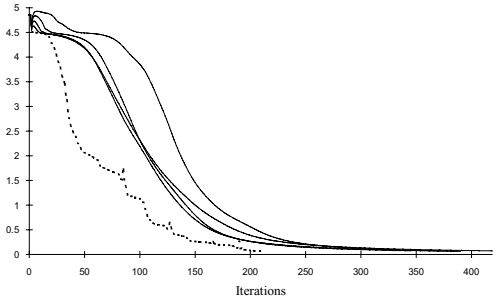
input	output
10000000	10000000
01000000	01000000
00100000	00100000
00010000	00010000
00001000	00001000
00000100	00000100
00000010	00000010
00000001	00000001

10-5-10 ENCODER

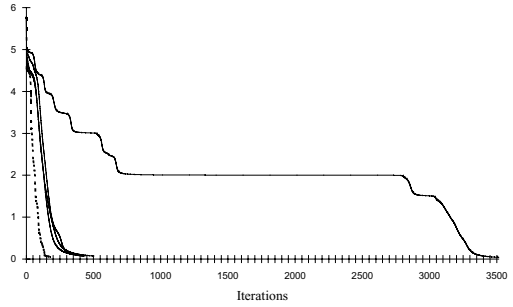
input	output
1000000000	1000000000
0100000000	0100000000
0010000000	0010000000
0001000000	0001000000
0000100000	0000100000
0000010000	0000010000
0000001000	0000001000
0000000100	0000000100
0000000010	0000000010
0000000001	0000000001

10-5-10 COMPLEMENT ENCODER

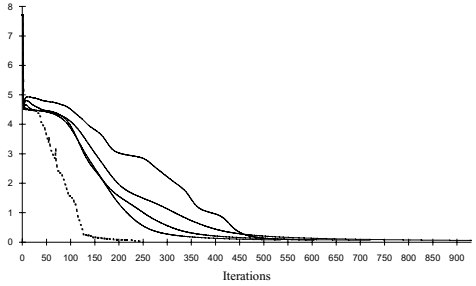
input	output
0111111111	0111111111
1011111111	1011111111
1101111111	1101111111
1110111111	1110111111
1111011111	1111011111
1111101111	1111101111
1111110111	1111110111
1111111011	1111111011
1111111101	1111111101
1111111110	1111111110



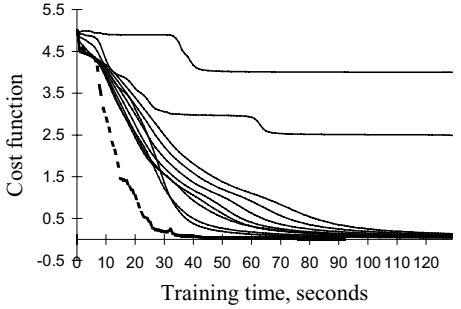
Training of 10-5-10 complement encoder task



Training of 10-5-10 complement encoder task



Training of 10-5-10 complement encoder task



Training history for 10-5-10 complement encoder benchmark task