4

Factors Affecting the Performance of Artificial Neural Network Models

Artificial neural network is widely used in various fields like system's modelling, forecasting, control, image processing and recognition, and many more. The development of multi-layered ANN model for a particular application involves many issues which affect its performance. ANN performance depends mainly upon the following factors:

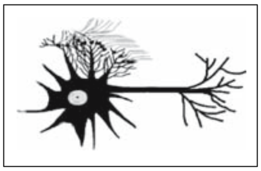
1. Network
2. Problem complexity
3. Learning Complexity.

4.1 Network Complexity

Network complexity broadly depends on

a. Neuron complexity
b. Number of neurons in each layer
c. Number of layers
d. Number and type of interconnecting weights.

4.1.1 Neuron Complexity



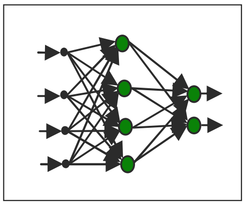
Mainly the neuron complexity could be viewed at two levels; firstly at aggregation function level and secondly at activation function level. There are two types of aggregations functions used for neuron modelling such as summation or product functions, but some researchers used combination of both summation and product aggregation function such as compensatory operators (Chaturvedi et al. 1997, 1999). The threshold functions used in neuron may be discrete like hard limiter used by McCulloch and Pitts (1943) in their neuron model or continuous functions like linear or non-linear function like sigmoid, Gaussian functions, etc.

The activation for a neuron can be thought of as the amount by which the neuron is affected by the input it receives. One could picture a neuron vibrating degrees depending on how excited it has become, and different neurons will be excited, or depressed the matter, by different stimuli and by differing degrees. Actually defining this state of activation for each unit within a model, and assigning a value to it, is a tricky process because the precision of the model depends on the reaction of the individual units.

Some models use a set of discrete values, that is, one of a finite set of possible values. These are often taken to be 0, 1 or -1. On the other hand, a model may take any value between two limits. This termed a continuous set of values, because for any two numbers there is always one that you can find that lies between them. In some cases, the model may have no upper or lower limit for the continuous values, but this presents problems, values can grow to an unmanageable size very quickly.

In this section, the effect of various activation functions on ANN model are considered for dc motor current prediction problem and found that the tan sigmoid function at hidden layer and pure linear function at output layer in a three layer network, where input layer is simply distributing the inputs in various hidden layer and no processing takes place there, requires least number of training epochs (i.e. 104). The comparisons of the results obtained for different activation functions are shown in bar chart, Fig. 4.1. From bar chart it is quite clear that the other functions takes more training epochs then also the model cannot be trained to the desired error level for some functions. The functions pure linear and pure linear in the model at hidden and output layers respectively also requires same number of training epochs but the results predicted for the non-linear problems are not so good. The function pair log sigmoid and log sigmoid is also able to train the model upto the desired error level but training epochs required is very large (in this case it requires 2,175). Remaining all other function pairs can not train the model up to the desired level when trained up to 2,200 epochs.

4.1.2 Number of Layers



While developing ANN model, two layers are fixed, namely input layer and output layer. Generally, at the input layer, the inputs are distributed to other neurons in the next layer and no processing takes place at this layer. Unlike the input layer, at output layer processing is done. Therefore, in a two layer network there is only one processing layer and this type of ANN can be used for linearly separable problems. Most of the real life problems are not linearly separable in nature and hence this type of two layer network could not be used. In the literature

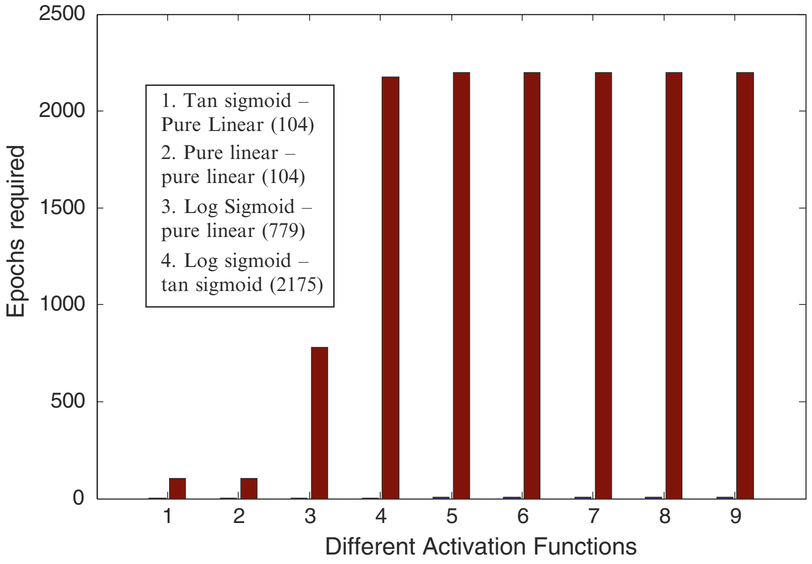


Fig. 4.1. Effect of different activation functions at different layer

it is mentioned that the three layer network is a universal approximator and could handle most of the problems. Then also for complex problems, it is difficult to train ANN with three layers network structure. Hence, most of the time the ANN developer uses trial and error method to select the number of layers in the ANN structure.

There are two ways to deal with this problem. Firstly, one can start with three layers network and then during training the number of layers and neurons may be increased till the satisfactory performance is obtained. Second method to handle this situation is, one could begin with large number of layers and then start deleting the layers and neuron, till the ANN size is optimal.

4.1.3 Number of Neurons in Each Layer

The number of neurons at input layer and output layer are equal to the number of input and output variables, but the problem lies with the number of neurons at hidden layers. It is mentioned that the number of neuron in the hidden layer is the average of number of neurons at input and output layers. But it is not hard and fast rule.

4.1.4 Type and Number of Interconnecting Weights

Generally every neuron in ANN is interconnected with its as previous layer neurons and each interconnection has some weight (signal gain), which modifies the input signal in one way or the other. The weights in the neural network could be deterministic or fuzzy in nature. Normally, ANN weights are deterministic and can be determined by some learning rule. It is well proven and logical also that it is not necessary to connect every neuron with the other neuron in the next layer. We can remove some of the connections to reduce the complexity of the ANN and ultimately the training time of it.

To select the optimal size of the network, there are two techniques generally adapted; either one could start with large number of neurons in each layer of the network and during training remove the connections till its performance is not optimal or we can start with minimal size of network and then insert the neurons and layers to achieve the optimal size of the network as mentioned earlier.

4.2 Problem Complexity

The performance of ANN models does not depend only on the size of the neural network that is chosen for the problem in hand, but it also depends on the problem complexity. The problem complexity depends on the type of functional mapping, accurate and sufficient training data acquired and their effective way of presentation to ANN during training. During the training phase of ANN, unknown neural network weights are to be determined. If the unknown network weights are more than the training data, then they could not be determined. Therefore, the training data must always be more in number than unknown weights, otherwise network will not train perfectly (means the error will never reach to global minima).

The training performance also depends on the effective way of presentation of data, in which following points have to be considered.

4.2.1 Range of Normalization of Training Data

Normalisation has a major role in the training and testing of neural networks. It is necessary to normalize the input and output in the same order of magnitude. Normalization is very critical issue in ANN. If the input and the output variables are not of the same order of magnitude, some variables may appear to have more significance than they actually do. The training algorithm has to compensate for order-of-magnitude differences by adjusting the network weights, which is not very effective in many of the training algorithms such as back propagation algorithm. For example, if one input variable has a value of thousands and other input variable has a value in tens, the assigned weight for the second variable entering a node of hidden layer 1 must be much greater than that for the first. In addition, typical transfer functions, such as a sigmoid function, or a hyperbolic tangent function, cannot distinguish between two values of xi when both are very large, because both yield identical threshold output values of 1.0.

Whenever we do normalisation of training and testing data, we need to determine minimum and maximum value of the given data. The problem is that these maximum and minimum values restrict the operating range of the network (Welstead 1994). A network that has been trained to predict a maximum change in output say 1% cannot possibly predict a change of 2%, even if the input data warrants it. This creates problems in trying to model volatile change in data. The remedy for this situation is somewhat by expanding the maximum and minimum values. First of all determine actual max-min values and then new maximum values is computed by adding 10% to the previous maximum value and a new minimum value is computed by subtracting 10% to the previous minimum value. The network can now handle values that fall within this expanded range and finally train the neural network model for these normalised data. Note that normalised data is something that is of interest only to the network. The user wants to get ANN output in the range of the actual data. For this reason, it is necessary to convert back the output of neural network into the actual range by denormalizing the ANN output.

Too large a range in relation to the actual data value has the effect of compressing the data so that it all looks the same to the network during training. If the range is too short then the neural network model could not predict the value outside that range and it will give absurd results. Hence, the selections of suitable range (i.e. max–min values) is of great importance, because it will affect the results of neural network model during testing.

The neural network is trained for different normalisation ranges and found very encouraging results. The authors have seen that if the input data of neural network model is normalised in the range of -0.9 to $+0.9$ and output data in the range of 0.1 to 0.9 then model took least number of epochs to train when threshold functions at hidden layer is tan sigmoid and at output layer is pure linear. The comparison of various normalisation ranges during and testing have been studied and the results given in Table 4.1, and Table 4.2 for modelling and simulation of dc motor using neural network. The ANN model was also developed for short term electrical load forecasting problem and the effect of different normalization range had been studied. The simulation results representing training and testing performance are complied in Tables 4.3–4.5 and shown in Fig. 4.2.

Generally it is found that the two layer neural network with tan sigmoid threshold functions at hidden layer and pure linear threshold function at output layer can train for any set of non-linear data and the performance will improve if the normalisation range taken between -0.9 to $+0.9$ for input and 0.1 to 0.9 for output.

4.2.2 Type of Functional Mapping

There are four possibilities in preparing training patterns (input and output vectors) for ANN models as shown in Fig. 4.3.

Table 4.1. DC motor current simulations with different normalisation range (Tolerable error = 10^{-3}, mapping actual input and actual output (X–Y) Activation functions – tan sigmoid at hidden layer and pure linear at output layer.)

Normalization	X (0.1−2.5)	X (0.1−0.9)	X (0.1−2.5)	X (0.1−0.9)	X (−0.9 to +0.9)
	Y (0.1−2.5)	Y (0.1−2.5)	Y (0.1−0.9)	Y (0.1−0.9)	Y (0.1−0.9)
Epochs	1500 (NT)	93	81	104	86
Test	2.1725	2.1816	2.1779	2.1951	2.2030
Results	1.9809	1.9340	1.9403	1.9518	1.9569
	1.7008	1.6976	1.7091	1.7136	1.7153
	1.4955	1.4883	1.5018	1.5014	1.5006
	1.3186	1.3096	1.3232	1.3201	1.3181
	1.6101	1.1596	1.1724	1.1682	1.1662
	1.0431	1.0349	1.0465	1.0420	1.0409
	0.9399	0.9319	0.9421	0.9379	0.9383
	0.8539	0.8478	0.8559	0.8522	0.8542
	0.5655	0.5641	0.5678	0.5666	0.5782
	0.5455	0.5446	0.5470	0.5469	0.5594

Table 4.2. DC motor speed simulations with different normalisation range (Tolerable error = 10^{-3}, mapping – actual input and actual output (X–Y) Activation functions – tan sigmoid at hidden layer and pure linear at output layer)

Normalization	X (−0.1 to +0.9)	X (0.1−2.5)	X (0.1−0.9)	X (0.1−2.5)	X (0.1−0.9)
	Y (0.1−0.9)	Y (0.1−0.9)	Y (0.1−2.5)	Y (0.1−0.9)	Y (0.1−0.9)
Epochs	53	64	71	3,000 (NT)	3,000 (NT)
Test	43.7420	40.2860	41.5429	41.0291	41.9124
Results	58.2756	57.4027	57.2390	57.3006	57.5467
	71.6113	72.3925	71.2981	71.9746	71.7368
	83.1838	84.7799	83.2701	84.3292	83.8102
	92.9339	94.7392	93.2272	94.3757	93.7644
	101.0195	102.6546	101.4171	102.4133	101.8504
	107.6690	108.9259	108.1201	108.8045	108.3780
	113.1148	113.9000	113.5955	113.8828	113.6386
	117.5664	117.8571	118.0663	117.9255	117.8803
	121.2035	121.0171	121.7179	121.1547	121.3059
	124.1752	123.5504	124.7023	123.7417	124.0782
	126.6045	125.5886	127.1431	125.8227	126.3263
	127.2335	127.2335	129.1404	127.5014	128.1526

(1) Actual input vector and actual output vector (**X-Y** mapping)
(2) Actual input vector and change in previous value of output vector (**X-ΔY** mapping)
(3) Change in input vector and actual output vector (**ΔX-Y** mapping)
(4) Change in input vector and change in output vector (**ΔX-ΔY** mapping).

Table 4.3. Electrical load forecasting with different normalisation range (Tolerable error = 1, mapping – actual input and actual output (X–Y) Activation functions – tansig at hidden layer and pure linear at output layer)

Normalization	X (±0.9) Y (0.1–0.9)	X (0.1–2.5) Y (0.1–0.9)	X (0.1–0.9) Y (0.1–0.9)
Epochs	112	404	436
Test	2,257.4	2,254.4	2,285.6
Results	2,279.6	2,251.0	2,285.4
	2,704.8	2,693.3	2,697.4
	3,043.0	3,037.8	3,028.3
	3,302.5	3,286.0	3,296.0
	3,292.2	3,285.8	3,292.4
	3,191.1	3,198.2	3,197.0
	3,161.1	3,156.7	3,164.4
	2,911.6	2,929.1	2,920.3
	2,667.3	2,680.5	2,682.7
	2,751.9	2,741.2	2,755.2
	2,921.3	2,911.1	2,915.9
	3,012.9	3,015.4	3,009.9
	2,898.1	2,918.4	2,902.6
	3,040.4	3,039.2	3,040.0
	2,904.6	2,918.0	2,906.8
	3,106.4	3,098.9	3,105.6
	2,960.8	2,971.5	2,961.4
	2,911.1	2,927.8	2,918.8

Table 4.4. Comparison of ANN training with different normalization ranges (activation function "tansig – purelin", mapping x–y)

Range	ω– characteristics of DC motor	ω – t characteristics of ind. motor	P-δ characteristics of alternator	Ia-t characteristics DC motor	STLF
X (−0.1 to 0.9) Y (0.1 to 0.9)	53	1,311	85	–	–
X (−0.1 to 2.5) Y (0.1 to 0.9)	64	726	61	–	–
X (0.1 to 0.9) Y (0.1 to 2.5)	71	1,100	151	93	–
X (0.1 to 2.5) Y (0.1 to 0.9)	3000	736	62	81	404
X (0.1 to 0.9) Y (0.1 to 0.9)	3000	1,950	162	104	436
X (−0.9 to 0.9) Y (0.1 to 0.9)	[a]	558	69	86	112
X (−0.9 to 0.9) Y (−0.9 to 0.9)	[a]	1,666	54	–	–

[a]ANN not trained

Table 4.5. ANN testing with different normalization ranges for STLF (Tansig-Purelin, X–Y mapping)

Range of Normalization	Max error	Min error	SS error
X (±0.9) – Y (0.1–0.9)	5.6491	−6.0533	11.0324
X (0.1–2.5) – Y (0.1–0.9)	5.4202	−5.8098	10.6585
X (0.1–0.9) – Y (0.1–0.9)	5.1044	−5.8446	9.2511

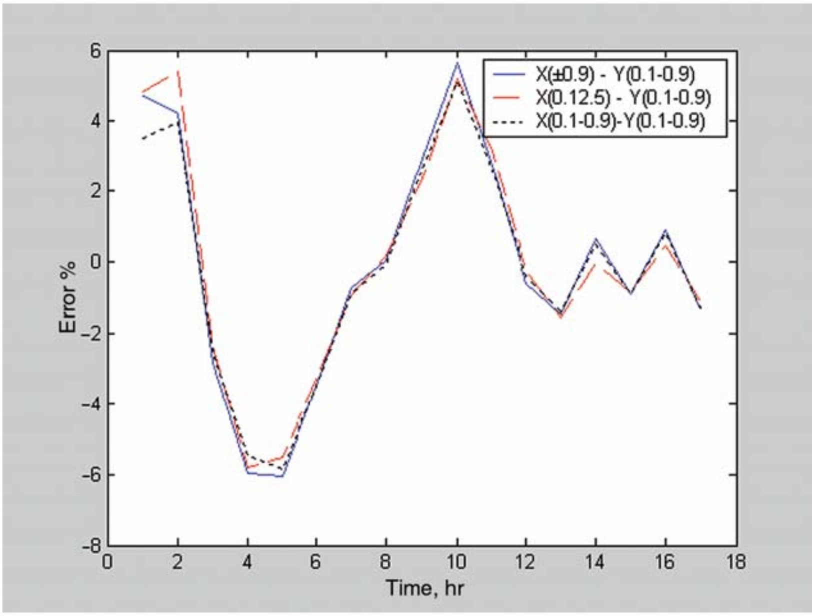


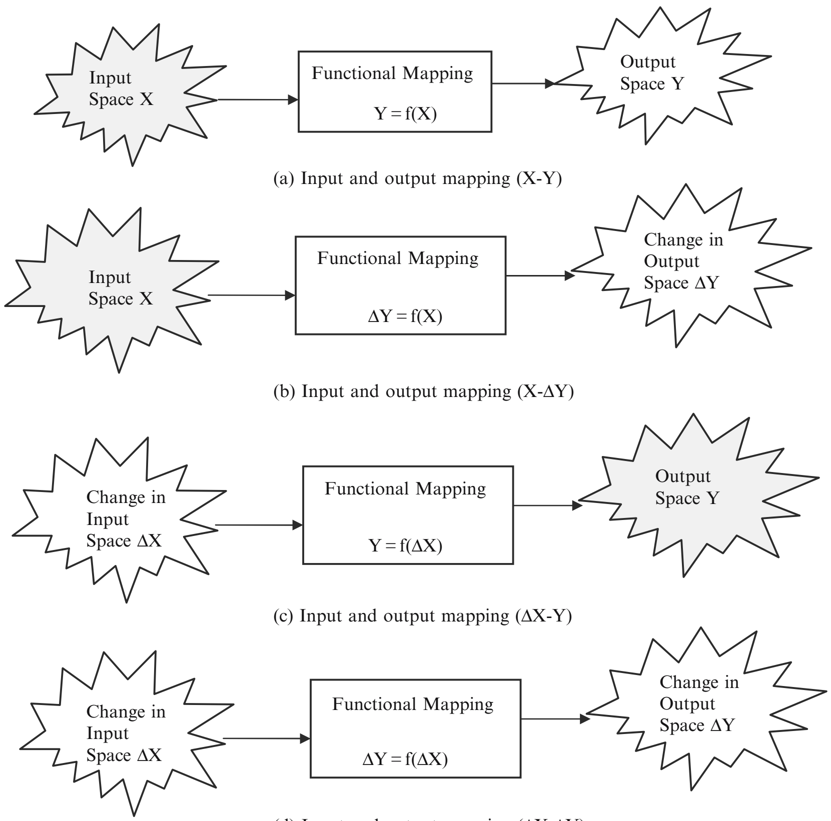
Fig. 4.2. Effect of normalization on short term load forecasting problem

There is no way of knowing a priori which of these myriad approaches is the best one. In this section the effects of all these mappings on the training and testing of following cases have been studied while

(a) Mapping of dc motor current and speed, and
(b) Predicting the electrical load demand.

The training file for dc motor consists of two inputs at adjacent time instances (say I(t-to) and I(t-2*to), where to is the sampling time) and one output O(t). Testing file contains 80% of the training file data and 20% additional data, which can test the model's performance on data from outside the training set. Similarly, for load forecasting problem we have taken data of four Mondays and predict the data of fifth Monday.

CASE – I
The dc motor data are used to train back propagation feedforward neural network for X-Y, X-ΔY, ΔX-Y, ΔX-ΔY mappings. The training algorithms



(a) Input and output mapping (X-Y)

(b) Input and output mapping (X-ΔY)

(c) Input and output mapping (ΔX-Y)

(d) Input and output mapping (ΔX-ΔY)

Fig. 4.3. Different functional mappings between input and output space

used are steepest descent based and its modifications. The modified algorithm is commonly known as Levenberg–Marquardt. These algorithms are available in **MATLAB** Tool Box on neural networks. It is found that X-Y mapping requires least number of epochs (i.e. 98) for training and X-Y mapping requires maximum number of epochs (i.e. 105). Tables 4.6 and 4.7 represent comparative analysis of the results of all these mappings and their training epochs and predicted results for dc motor current and speed prediction under starting conditions respectively. In these simulations: Tolerable error $= 10^{-3}$, Normalisation – input and output both in the range 0.1–0.9, and activation functions – Tan sigmoid at hidden layer and pure linear at output layer.

CASE – II
For electrical load forecasting problem, the comparison between all these mappings is given in Tables 4.8–4.11. Figure 4.4 shows the percentage error during forecasting of the electrical demand of the totally unforeseen data of the fifth Monday.

Table 4.6. DC motor current simulations with different mappings

Mappings	X–ΔY	ΔX–Y	X–Y	ΔX–ΔY	Actual values
Epochs	98	103	104	105	
Test	2.1921	2.1942	2.1951	2.1921	2.1905
Results	1.9394	1.9198	1.9518	1.9388	1.9463
	1.7004	1.6781	1.7136	1.6994	1.6960
	1.4909	1.4728	1.5014	1.4894	1.4860
	1.3129	1.3002	1.3201	1.3113	1.3082
	1.1641	1.1575	1.1682	1.1625	1.2305
	1.0406	1.0405	1.0420	1.0388	1.0955
	0.9384	0.9424	0.9379	0.9366	0.9839
	0.8540	0.8627	0.8522	0.8521	0.8920

Table 4.7. DC motor speed simulations with different mappings

Mappings	X–ΔY	ΔX–ΔY	ΔX–Y	Actual values
Epochs	56	140	248	
Test	48.3522	48.0219	44.6986	49.3902
Results	62.8526	62.7167	58.7081	64.2083
	75.3795	75.3599	71.9871	76.7749
	85.9263	85.9497	83.6112	87.2634
	94.7077	94.7455	93.4166	95.9506
	101.9823	102.0209	101.5404	103.1190
	107.9939	108.0276	108.2138	109.0232
	112.9552	112.9814	113.6756	113.8815
	117.0463	117.0645	118.1362	117.8775
	120.4181	120.4284	121.7800	121.1635
	123.1959	123.1988	124.7576	123.8653
	125.4836	125.4791	127.1963	126.086

4.2.3 Sequence of Presentation of Training Data

In the natural learning process of the human being, generally the simple and easy things we learn quickly. So we start our learning with simple things, which motivate and encourage us to learn more. Once we have learned simple things then more time can be spent on difficult things to learn. Hence, it is very important that how we started our learning or what is the sequence of presentation of data for learning. ANN training performance is also very much dependent on in what manner the data is to be presented to ANN. If we cluster the data and then present it to ANN, then it will learn more efficiently and quickly.

4.2.4 Repetition of Data in the Training Set

Some difficult patterns which are not remembered by ANN we have to repeat them. Now how many times that pattern is to be repeated? This is a very

Table 4.8. Short term electrical load forecasting with different mappings

Mappings	X–ΔY	ΔX–ΔY	ΔX–Y	X–Y	Actual demand
Epochs	35	87	800 (NT)*	52	
Test	2,573.4	2,187.6	2,538.0	2,456.8	2,369
results	2,568.6	2,449.7	2,744.9	2,429.7	2,380
	2,803.4	2,767.7	3,011.2	2,545.6	2,631
	2,995.7	3,034.4	3,157.9	2,738.0	2,871
	3,167.0	3,134.4	3,163.8	2,958.3	3,114
	3,161.7	3,067.1	2,950.3	3,097.5	3,182
	3,102.4	3,021.3	2,785.1	3,148.5	3,168
	3,072.9	2,816.4	2,514.0	3,162.3	3,162
	2,923.6	2,588.5	2,321.1	3,087.9	3,000
	2,784.9	2,588.5	2,367.8	2,960.0	2,827
	2,817.8	2,695.6	2,488.6	2,893.8	2,830
	2,912.4	2,796.2	2,644.3	2,901.3	2,904
	2,978.1	2,763.4	2,683.9	2,944.7	2,969
	2,919.0	2,846.2	2,735.7	2,945.9	2,917
	2,997.5	2,778.8	2,648.2	2,990.8	3,013
	2,914.2	2,878.8	2,685.6	2,974.1	2,931
	3,037.7	2,835.0	2,676.8	3,025.7	3,065

Table 4.9. ANN training performance with different functional mappings

Mapping	DC motor current	DC motor speed	Short term load forecasting
X–Y	104	107	52
ΔX–Y	103	248	800
X–ΔY	98	56	35
ΔX–ΔY	105	140	87

Table 4.10. ANN testing performance with different mappings for dc motor current

Mapping	Max error	Min error	SS error
X–Y	0.0623	−0.0176	0.0112
ΔX–Y	0.0730	−0.0037	0.0122
X–ΔY	0.0664	−0.0049	0.0111
ΔX–ΔY	0.0680	−0.0034	0.0118

important question. For example while teaching English alphabets to the students in the elementary classes; most often the students commit the mistake while writing "b" and "d". Then the teacher gives them as home assignment to repeat these alphabets 10 times, 20 times or even more depending on the students' capability. Same thing is true for ANN learning.

Table 4.11. ANN testing performance with different mappings for STLF

Mapping	Max error	Min error	MSS error
X−Y	22.6300	−5.3319	160.3354
ΔX−Y	13.7167	−5.6914	39.9838
X−ΔY	2.8178	−8.6281	13.2372
ΔX−ΔY	5.0000	−4.7046	7.3138

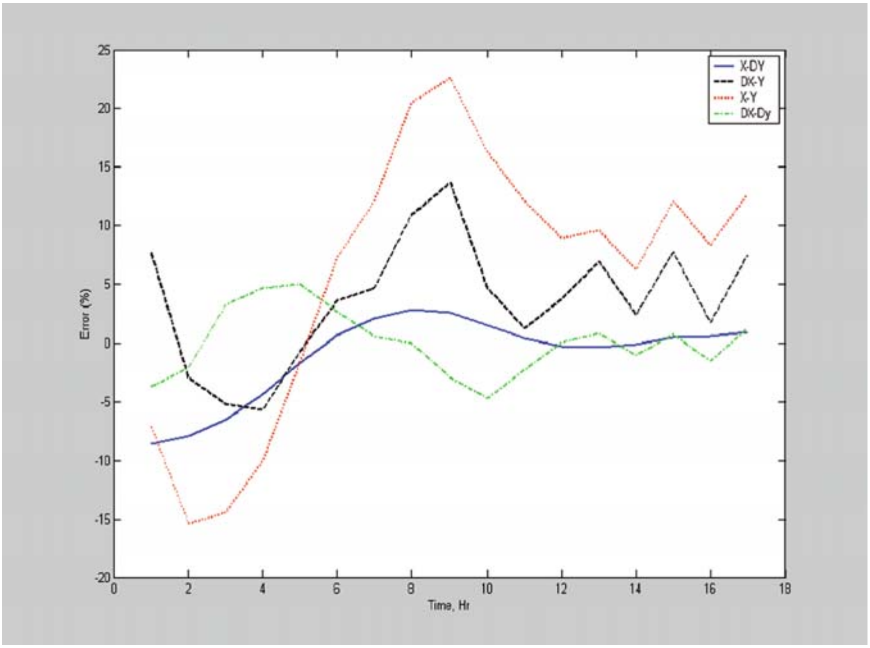


Fig. 4.4. Effect of mapping on short term load forecasting problem

4.2.5 Permissible Noise in Data

The generalization characteristics of ANN models depends on the noise included in the training data, but at the same time the accuracy reduces. Hence, we have to trade off between the generalization capability of neural networks and accuracy required in the results.

Usually when the measurements are taken by different measuring devices, are not accurate due to various reasons. Hence, the noise will be there in the measured quantities. According to the noise either in input or/and output of the training file different pattern mappings are possible. In this chapter the neural network is trained for the following mappings.

(1) Noisy input and accurate output patterns (Xnoise – Y mapping).
(2) Noisy input and noisy output patterns (Xnoise – Ynoise mapping).

Table 4.12. DC motor current simulations with noisy data (Normalisation – input 0.1 to 2.5 and output in the range 0.1–0.9)

Mappings	Xnoise–Y	X–Ynoise	Xnoise–Ynoise
Epochs	400	400	400
Error after training	0.00442398	0.00265015	0.00395185
Error during testing	1.6033 to −2.2048	−0.9237 to −3.800	2.8098 to 4.8763

Table 4.13. Training performance with noisy data for dc motor current (Ia) characteristic

Mappings	Xnoise–Y	X–Ynoise	Xnoise–Ynoise
Training error (After 400Epochs)	0.00442398	0.00265015	0.00395185
Testing error	1.6033 to −2.2048	0.9237 to −3.8	−2.8908 to 4.8763

(3) Accurate input and noisy output (X – Ynoise mapping).
(4) Accurate input and accurate output (X–Y mapping).

The training and testing statistics of the neural network model for the above combination is summarised in Tables 4.12 and 4.13 for dc motor simulation. It has been found that the X–noise mapping required least number of training epochs and also giving good results during predictions. Here the random noise of 5% is added in the training data either/both in input and output data.

4.3 Learning Complexity

Performance of supervised learning depends upon:

a. Training algorithms
b. Initialization of weights
c. Selection of error Function
d. Mode of error calculation
e. Initialization of training parameters

4.3.1 Training Algorithms of ANN

Multi-layered networks have been applied successfully to solve some difficult and diverse problems by training them in a supervised manner with a highly popular algorithm known as the *error back-propagation algorithm*. This algorithm is based on the *error-correction learning rule*.

Basically, the error back-propagation process consists of two passes through the different layers of the network; a forward pass and a backward pass. In the *forward pass*, an activity pattern (input vector) is applied to the sensory nodes of the network and its effect propagates through the network, layer-by-layer. Finally a set of outputs is produced as the actual response of the network. During the forward pass, the synaptic weights of the network are fixed. During the *backward pass*, on the other hand, the synaptic weights are adjusted in accordance with the error-correction rule. Specifically, the actual response of the network is subtracted from a desired (target) response to produce an *error signal.* This error signal is then propagated backward through the network against the direction of synaptic connections – hence the name "error back-propagation". The synaptic weights are adjusted so as to make the actual response of the network move closer to the desired response.

A multi-layered perceptron network has three distinctive characteristics:

1. The model of each neuron in the network includes a differentiable non-linearity, as opposed to the hard limiting used in McCullock and Pitt's perceptron model. A commonly used form of non-linearity that satisfies this requirement is the sigmoid non-linearity.

   $$f(net) = \frac{1}{1 + \exp(-\lambda\ net)} \tag{4.1}$$

   The presence of non-linearity is important to prevent reduction of the model to that of single-layered perceptron. The use of logistic function is encouraging as it is a biologically motivated function.
2. The network contains one or more hidden layers that enable the network to learn complex tasks by extracting multi-dimensional features from the input pattern vectors.
3. The network exhibits a high degree of connectivity determined by the synapses of the network. A change in the connectivity requires a change in the population of synaptic connections/weights.

All these characteristics together with the ability to learn through training that is the multi-layered perceptron derives its computing power. These same characteristics, however, are also responsible for the deficiencies in knowing the network behaviour. First, the presence of a distributed form of non-linearity and the high connectivity of the network make the theoretical analysis of a multi-layered perceptron difficult to undertake. Second, the use of hidden layers makes the learning process opaque to external environment. In an implicit sense, the learning process is rigorous enough to decide which features of the input pattern should be represented by the hidden layers and the search has to be conducted on a larger space of possible functions.

The development of the back-propagation algorithm represents a "landmark" in the field of neural networks in that it provides a *computationally efficient* method for the training of multi-layered perceptrons.

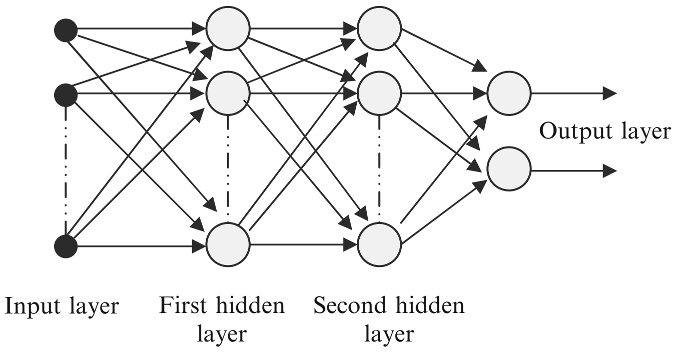


Fig. 4.5. NN architecture with two hidden layers

4.3.1.1 Preliminary Fundamentals

The network shown in Fig. 4.5 is fully connected and the signal flows through the network in a forward direction, from left to right and on a layer-by-layer basis. The error signal flow propagates in a backward direction from right to left, again on a layer-by-layer basis.

The signals should be appropriately called function signals as they are calculated as a function of inputs and associated weights.

The error signal is so called because its computation by every neuron of the network involves an error-dependent function in one or another form.

The hidden layer(s) are not part of the input or output layers and hence designated as "hidden". Their behaviour within the architecture is totally "hidden" from analysis.

Each hidden or output neuron of a multi-layered perceptron is designated to perform two computations.

1. The computation of the function signal appearing at the output of a neuron, which is expressed as a continuous non-linear function of the input signals and synaptic weights.
2. The computation of an instantaneous estimate of the gradient, i.e. the gradient of the error surface with respect to the weights connected to the inputs of a neuron, which is needed for the backward pass through the network (Fig. 4.6).

4.3.1.2 The Back-Propagation Algorithm

Before getting into the derivation of the algorithm, we will see the notations used in the derivation.

$E(n)$ = Instantaneous sum of error squares at iteration n. The average of $E(n)$ over all values of n (i.e. the entire training set) yields the average squared error E_{av}.

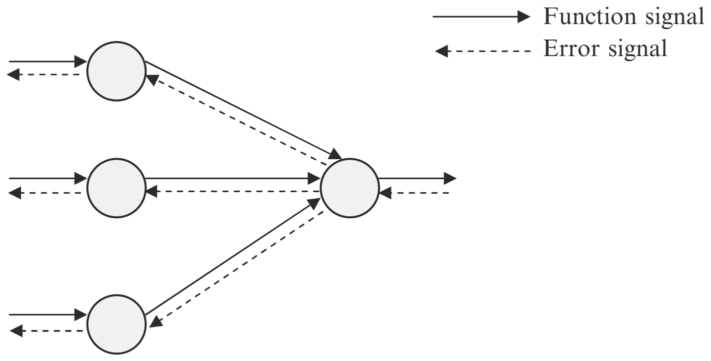


Fig. 4.6. Signal flow illustration

$e_j(n)$ = Error signal at the output of neuron j for iteration n.
$d_j(n)$ = Desired response for neuron j used to compute $e_j(n)$.
$y_j(n)$ = Function signal appearing at the output of neuron j for iteration n.
$w_{ij}(n)$ = Synaptic weight connecting neuron i to neuron j at iteration n.
$\Delta w_{ij}(n)$ = The correction applied to the synaptic weight at iteration n.
$v_j(n)$ = The net internal activity level of neuron j at iteration n.
$\varphi_j(.)$ = The activation function associated with neuron j.
θ_j = The threshold applied to neuron j which is equivalent to an extra synapse.
$x_i(n)$ = The ith element of the input vector (pattern).
$o_k(n)$ = The kth element of the overall output vector (pattern).
η = The learning-rate parameter.

The error signal at the output of neuron j at iteration n (i.e. presentation of the nth training pattern) is defined by

$$e_j(n) = d_j(n) - y_j(n), \tag{4.2}$$

neuron j is an output node.

The *instantaneous sum of squared errors* of the network at the output of neuron j can be written as

$$E(n) = \frac{1}{2}\sum_{j\in c} e_j^2(n), \tag{4.3}$$

where c is the set of all neurons in the output layer of the network.

If N is the total number of patterns in the training set, the *average squared error* over all the patterns is given by

$$E_{av} = \frac{1}{N}\sum_{n=1}^{N} E(n). \tag{4.4}$$

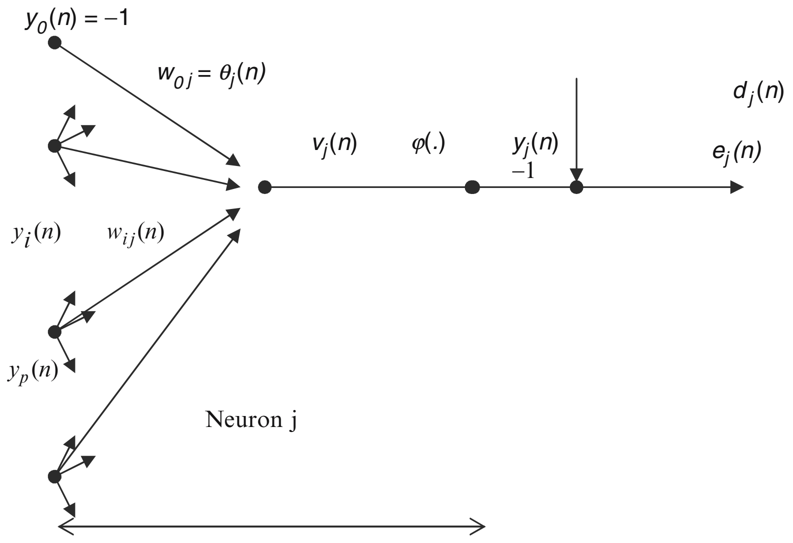


Fig. 4.7. Signal flow of output neuron j

The instantaneous sum of error squares $E(n)$, and therefore, the average squared error E_{av} is a function of the synaptic weights and thresholds. Thus E_{av} represents the *cost function* of the learning process, which adjusts the free parameters of synaptic weights and thresholds so as to minimise the *cost function*. The training is done on a *pattern-by-pattern* basis and the errors *computed* for each pattern presented to the network.

The neuron j as shown in Fig. 4.7, is fed from the layer to its left.

$$v_j(n) = \sum_{i=0}^{p} w_{ij}(n)y_i(n) \tag{4.5}$$

where p is the total number of inputs excluding the threshold applied to neuron j.

$$y_j(n) = \varphi_j(v_j(n)) \tag{4.6}$$

The back-propagation algorithm applies a correction $\Delta w_{ij}(n)$ to the synaptic weight $w_{ij}(n)$, which is proportional to the instantaneous gradient $\partial E(n)/\partial w_{ij}(n)$. According to the chain rule of partial derivatives, we may express the gradient as follows:

$$\frac{\partial E(n)}{\partial w_{ij}(n)} = \frac{\partial E(n)}{\partial e_j(n)} \times \frac{\partial e_j(n)}{\partial y_j(n)} \times \frac{\partial y_j(n)}{\partial v_j(n)} \times \frac{\partial v_j(n)}{\partial w_{ij}(n)}. \tag{4.7}$$

Now, differentiating (4.3) with respect to $e_j(n)$, we get

$$\frac{\partial E(n)}{\partial e_j(n)} = e_j(n) \tag{4.8}$$

Differentiating (4.2) with respect to $v_j(n)$, we get

$$\frac{\partial e_j(n)}{\partial y_j(n)} = -1. \tag{4.9}$$

Differentiating (4.6) with respect to $v_j(n)$ yields

$$\frac{\partial y_j(n)}{\partial v_j(n)} = \varphi_j'(v_j(n)). \tag{4.10}$$

Finally, differentiating (4.4) with respect to $w_{ij}(n)$ yields

$$\frac{\partial v_j(n)}{\partial w_{ij}(n)} = y_i(n). \tag{4.11}$$

Thus (4.7) becomes $\dfrac{\partial E(n)}{\partial w_{ij}(n)} = -e_j(n)\varphi_j'(v_j(n))y_j(n)$ (4.12)

We know by delta learning rule, the correction to weight is

$$\Delta w_{ij}(n) = -\eta \frac{\partial E(n)}{\partial w_{ij}(n)}, \tag{4.13}$$

where η is a positive constant called the *learning rate.*

From equations (4.12) & (4.13), we have

$$\Delta w_{ij}(n) = \eta \delta_j(n) y_i(n), \tag{4.14}$$

where $\delta_j(n) = e_j(n)\varphi'(v_j(n))$ is called the *local gradient* at neuron j. The *local gradient* $\delta_j(n)$ for output neuron j is equal to the product of the corresponding error signal $e_j(n)$ and the derivative $\varphi'(v_j(n))$ of the associated activation function.

We note that a key factor involved in the calculation of the weight adjustment $\Delta w_{ij}(n)$ is the error signal $e_j(n)$. There are two distinct cases of adjustment, depending on where in the network neuron j is located.

Case 1: Neuron j is an output node

When neuron j is located in the output layer of the network, the case is pretty straight forward as the neuron will be supplied with a desired response. We can use (4.2) to compute the error signal $e_j(n)$ associated with this neuron and then use (4.14) to compute the *local gradient.*

Case 2: Neuron j is a hidden node

When neuron j is located in a hidden layer of the network, there is no specific desired response for that neuron. Accordingly, the error signal for a hidden neuron would have to be determined recursively in terms of the error signals of all the neurons to which the neuron is directly connected.

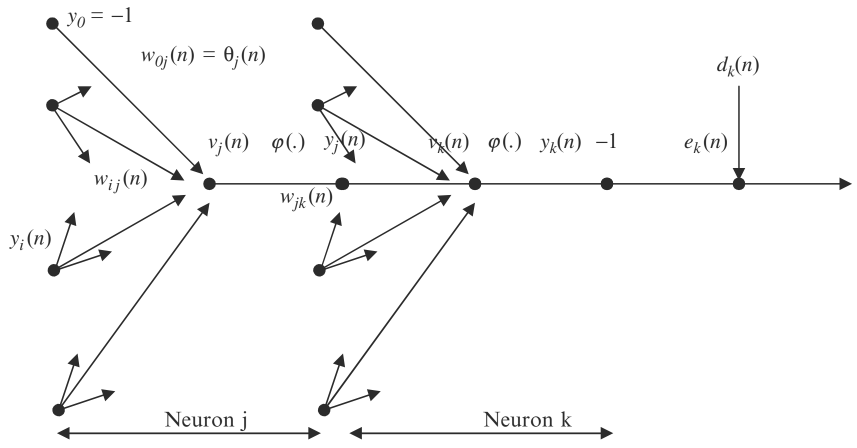


Fig. 4.8. Signal flow of hidden neuron j

Consider the case of the hidden neuron j as shown in Fig. 4.8 below.

We can redefine the local gradient

$$\delta_j(n) = e_j(n)\varphi'_j(v(n)) \tag{4.15}$$

as

$$\delta_j(n) = -\frac{\partial E(n)}{\partial y_j(n)} \times \frac{\partial y_j(n)}{\partial v_j(n)} \tag{4.16}$$

$$= -\frac{\partial E(n)}{\partial y_j(n)}\varphi'_j(v_n(n)) \tag{4.17}$$

neuron j is a hidden node.

To calculate the partial derivative $\partial E(n)/\partial y_j(n)$, we may proceed as follows (see Fig. 4.4)

$$E(n) = \frac{1}{2}\sum_{k\in c} e_k^2(n) \tag{4.18}$$

neuron k is an output node

$$\frac{\partial E(n)}{\partial y_j(n)} = \sum_k e_k \frac{\partial e_k(n)}{\partial y_j(n)} \tag{4.19}$$

Using the chain rule of partial derivatives, we can write (4.19) as

$$\frac{\partial E(n)}{\partial y_j(n)} = \sum_k e_k(n)\frac{\partial e_k(n)}{\partial v_k(n)} \times \frac{\partial v_k(n)}{\partial y_j(n)} \tag{4.20}$$

However,

$$\begin{aligned} e_k(n) &= d_k(n) - y_k(n) \\ &= d_k(n) - \varphi_k(v_k(n)) \end{aligned} \tag{4.21}$$

Hence

$$\frac{\partial e_k(n)}{\partial v_k(n)} = -\varphi'_k(v_k(n)). \tag{4.22}$$

Also, the net internal activity for neuron k is

$$v_k(n) = \sum_{j=0}^{q} w_{jk}(n) y_j(n), \tag{4.23}$$

where q is the total number of inputs (excluding the threshold) applied to neuron k.

Differentiating (4.23) with respect to $y_j(n)$ yields

$$\frac{\partial v_k(n)}{\partial y_j(n)} = w_{jk}(n) \tag{4.24}$$

Thus using (4.22) and (4.24), we get

$$\begin{aligned}\frac{\partial E(n)}{\partial y_j(n)} &= -\sum_k e_k(n)\varphi'_k(v_k(n))w_{jk}(n) \\ &= -\sum_k \delta_k(n) w_{jk}(n),\end{aligned} \tag{4.25}$$

where we have used the definition of the *local gradient* $\delta_k(n)$ given by (4.14) with the index k substituted for j. Finally using (4.25) in (4.17), we get the local gradient $\delta_j(n)$ for the hidden neuron j as

$$\delta_j(n) = \varphi'_j(v_j(n)) \sum_k \delta_k(n) w_{jk}(n) \tag{4.26}$$

The factor $\varphi'_j(v_j(n))$ involved in the computation of the local gradient $\delta_j(n)$ depends solely on the activation function associated with the hidden neuron j. The remaining factor, namely the summation over k, depends on two sets of terms. The first set of terms, the $\delta_k(n)$, requires the knowledge of the error signals $e_k(n)$, for all those neurons that lie in the layer to the immediate right of the hidden neuron j, and that are directly connected to neuron j; the second set of terms, the $w_{jk}(n)$, consists of the synaptic weights associated with these connections.

We may summarise the relations as follows:

$$\begin{aligned}\begin{pmatrix} \textit{Weight correction} \\ \Delta w_{ij}(n) \end{pmatrix} &= \begin{pmatrix} \textit{learning rate parameter} \\ \eta \end{pmatrix} \begin{pmatrix} \textit{local gradient} \\ \delta_j(n) \end{pmatrix} \\ &\quad \times \begin{pmatrix} \textit{input signal of neuron } j \\ y_i(n) \end{pmatrix}.\end{aligned}$$

The local gradient $\delta_j(n)$ depends on whether neuron j is an output node or a hidden node:

1. If neuron j is an output node, $\delta_j(n)$ equals the product of the derivative $\varphi_j'(v_j(n))$ and the error signal $e_j(n)$, both of which are associated with neuron j as given by (4.14).
2. If neuron j is a hidden node, $\delta_j(n)$ equals the product of the associated derivative $\varphi_j'(v_j(n))$ and the weighted sum of the δ's computed for the neurons in the next hidden or output layers that are connected to neuron j as given by (4.26).

4.3.1.3 The Two Passes of Computation

The application of back-propagation algorithm is in two steps or two distinct passes of computation. The first pass is referred as the *forward pass* and the second pass is the *backward pass.*

In the forward pass, the synaptic weights remain unaltered throughout the network, and function signals of the network are computed on a neuron-by-neuron basis.

The function signal appearing at the output of neuron j is computed as

$$y_j = \varphi(v_j(n)), \tag{4.27}$$

where

$$v_j(n) = \sum_{i=0}^{p} w_{ij}(n) y_i(n) \tag{4.28}$$

p is the total number of inputs (excluding the threshold) applied to neuron j and $w_{ij}(n)$ is the synaptic weight connecting neuron i to j, and $y_i(n)$ is the input signal of neuron j or the function signal appearing at the output of neuron i.

If neuron j is in the first hidden layer of the network, then the index i refers to the i^{th} input terminal of the network, for which we write

$$y_i(n) = x_i(n) \tag{4.29}$$

On the other hand, if neuron j is in the output layer of the network, the index j refers to the jth output terminal of the network, for which we can write

$$y_j(n) = o_j(n) \tag{4.30}$$

This output is compared with the desired response $d_j(n)$, obtaining the error signal $e_j(n)$ for the jth output neuron. Thus the forward phase of computation begins at the first hidden layer by presenting it with the input vector, and terminates at the output layer by computing the error signal.

In the backward pass, the error signals computed are passed leftward through the network, layer-by-layer and recursively computing the local gradient δ for each neuron. The synaptic weights are varied according to the back-propagation rule. The local gradient is computed by (4.15) or (4.26), depending on whether the neuron is in the output layer or hidden layer(s). The recursive computation is continued layer-by-layer, by propagating the changes to all synaptic weights from output layer to input layer. The computation of δ for each neuron of the multi-layered architecture requires the derivative of the activation function $\varphi(.)$ associated with that neuron. For this derivative to exist, we require the function $\varphi(.)$ to be continuous. In basic terms, *differentiability* is the only criterion that an activation function would have to satisfy. It has been observed that a non-linear activation function with maximum variation in the mid-values gives stability to the learning process. Such an activation commonly used is the sigmoid activation, whose derivative attains maximum at mid-value.

4.3.1.4 Rate of Learning and Momentum

The back-propagation algorithm provides an "approximation" to the trajectory in the error-weight space computed by the method of *steepest descent.*

According to the *method of steepest descent*, the weights are adjusted in an iterative fashion along the error surface with an aim of moving them progressively toward the optimum solution. The successive adjustments to the weights are in the direction of the *steepest descent* of the error surface.

The *rate of learning* η decides the scaling of the gradient of the error surface to be used for weight adjustment. The smaller we make the learning rate parameter, the smaller will be the changes to the synaptic weights in the network from one iteration to another and the smoother will be the trajectory in the error-weight space, this improvement being achieved at the cost of a slower learning. If we make the *rate of learning* η too large, so as to speed up the rate of learning, the resulting large changes in the synaptic weights may make the trajectory in the error-weight space oscillatory and unstable. It is better to make the learning rate adaptive, i.e. start with a larger η and progressively reduce as we move closer to the minimum. This is the implementation of back-propagation with *adaptive learning rate.*

Another simple method of increasing the rate of learning, and yet avoiding the danger of instability, is to include a *momentum* term as shown below.

$$\Delta w_{ij}(n) = \alpha \Delta w_{ij}(n-1) + \eta \delta_j(n) y_i(n), \tag{4.31}$$

where α is usually a positive number called the *momentum constant.* The delta rule as given by (4.14) is a special case with $\alpha = 0$.

In order to see the effect of using the momentum constant α, write (4.31) as a time series with index t. The index goes from $t = 0$ to current iteration $t = n$.

$$\Delta w_{ij}(n) = \eta \sum_{t=0}^{n} \alpha^{n-t} \delta_j(t) y_i(t), \tag{4.32}$$

$$\Delta w_{ij}(n) = -\eta \sum_{t=o}^{n} \alpha^{n-t} \frac{\partial E(t)}{\partial w_{ij}(t)}. \tag{4.33}$$

The above equations represent a time series of length $n+1$.

Following observations can be made:

1. The current adjustment $\Delta w_{ij}(n)$ represents the sum of an exponentially weighted time series. For the time series to be *convergent*, the momentum constant must be $0 \leq |\alpha| < 1$. The momentum constant can be positive or negative but it unlikely to use a negative α, in practice.
2. When the partial derivative $\partial E(t)/\partial w_{ij}(t)$ has the same algebraic sign on consecutive iterations, the exponentially weighted sum $\Delta w_{ij}(n)$ grows in magnitude and so the $w_{ij}(n)$ is adjusted by a large amount. Hence the inclusion of momentum in the back-propagation algorithm tends to accelerate the *descent* in steady downhill direction.
3. When the partial derivative $\partial E(t)/\partial w_{ij}(t)$ has opposite signs on consecutive iterations, the exponentially weighted sum $\Delta w_{ij}(n)$ shrinks in magnitude and so the $w_{ij}(n)$ is adjusted by a small amount. Hence the inclusion of momentum has a *stabilising effect* in the directions that oscillate in sign.

Thus the incorporation of momentum in the back-propagation algorithm represents a minor modification to the weight update and yet it can have highly beneficial effects on learning behaviour of the algorithm. The momentum term also helps in preventing the learning process from trapping in local minima. The momentum term can also be made adaptive just like the learning rate and the back-propagation implementation with adaptive η and/or α has been found to be much more efficient that the standard implementation.

4.3.1.5 The Stopping Criteria

There are several stopping criteria, each with its own practical merit, which may be used to terminate the weight adjustments. The logical thing to do is to think in terms of the unique properties of a local or global minimum of the error surface. Let the weight vector $\mathbf{w}^*$ denote a minimum, be it local or global. Various convergent criteria can be stated as follows:

- The back-propagation algorithm is considered to have converged when the Euclidean norm of the gradient vector reaches a sufficiently small gradient threshold. This means $\mathbf{g}(\mathbf{w}) \to \mathbf{0}$ at $\mathbf{w} = \mathbf{w}^*$. The drawback of this convergence criterion is that, for successful trials, learning time may be long. Also it requires the computation of the gradient vector $\mathbf{g(w)}$ of the error surface to the weight vector $\mathbf{w}$.
- Another unique property of a minimum that can be used is the fact that the *cost function* or error measure $E_{av}(\mathbf{w})$ is stationary at the point $\mathbf{w} = \mathbf{w}^*$. The back-propagation algorithm is considered to have converged when the absolute rate of change in the average error per epoch is sufficiently small. Typically considered ranges are from 0.01 to 1% per epoch.
- Kramer and Sangiovanni-Vincentelli (1989) suggested a hybrid criterion of convergence consisting of the former and the latter, as stated below: The back-propagation algorithm is terminated at the weight vector $\mathbf{w_{final}}$ when $||\mathbf{g}(\mathbf{w_{final}})|| \leq \varepsilon$, where ε is sufficiently small, or $E_{av}(\mathbf{w_{final}}) \leq \tau$, where τ is also sufficiently small.

- Another useful criterion for convergence is as follows:
  After each learning iteration the network is tested for its generalisation performance. The learning is stopped when the generalisation performance is adequate, or when it is apparent that the generalisation performance has peaked.

4.3.1.6 Initialization of the Network

The first step in back-propagation is, of course, to initialise the network. A good choice for the initial values of the free parameters (i.e. adjustable synaptic weights and threshold levels) of the network can be of tremendous help in a successful network development. In cases where the prior information is available, it may be better to use the information to guess the initial values of the free parameters. But how do we initialise the network if no prior information is available? It is also important to note that if all the weights start out with equal values and the solution requires that unequal weights be developed, the system can never learn. This is because the error is propagated back through the weights in proportion to the values of the weights. This means that all hidden units connected directly to the output units will get identical error signals, and since the weight changes depend on the error signals, the weights from those units to the output units must always be the same. This problem is known as the *symmetry-breaking* problem. Internal symmetries of this kind also give the *cost function* landscape periodicities, multiple minima, (almost) flat valleys and (almost) *flat plateaus* or *temporary minima.* The last are most troublesome, because the system can get struck on such a plateau during training and take immense time to find its way down the cost function surface. Without modifications to the training set or learning algorithm, the network may escape this type of "minimum" but performance improvement in these temporary minima drops to a very low, but non-zero level because of the very low gradient of the *cost function.* In the MSE vs. training time curve, a temporary minimum can be recognised as a phase in which the MSE is virtually constant for a long time after initial learning. After a generally long training time, the approximately flat part in the energy landscape is abandoned, resulting in a significant and sudden drop in the MSE curve. The problem of unequal weights can be counteracted by starting the system with random weights. However, as learning continues, internal symmetries may develop and the network may encounter again temporary minima.

The customary practice is to set all the free parameters of the network to random numbers that are *uniformly distributed* inside a small range of values. This is because if the weights are too large, the sigmoids will saturate from the very beginning of training and the system will become struck in a kind of *saddle point* near the starting point (Haykin, 1994). This phenomenon is called *premature saturation* (Lee et al. 1991). Premature saturation is avoided by choosing the initial weights and threshold levels of the network to be uniformly distributed inside a small range of values. This is so because when the weights are small, the units operate in their *linear regions* and consequently it is impossible for the activation function to saturate. It is also maintained that premature saturation is less likely to occur when the *number of hidden neurons* is maintained *low*, and in consistent with the network requirement but the viability of this belief is under question many a times.

Gradient descent can also become struck in *local minima* of the cost function. These are isolated valleys of the cost function surface in which the system may get "stuck" before it reaches the global minimum. This is so because in these valleys, every change in the weight values causes the cost function to increase and hence the network is unable to escape. Local minima are fundamentally different from temporary minima as they cause the performance improvement of the classification to drop to zero and hence the learning process terminates even though the minimum may be located far above the global minimum. Local minima may be abandoned by including a *momentum* term in the weight updates or by adding "noise" using the on-line mode training, which is a *stochastic* learning algorithm in nature. The momentum term can also significantly accelerate the training time that is spent in a temporary minimum as it causes the weights to change at a faster rate. Other approaches include the modification of the cost function or the employment of techniques such as simulated annealing.

There are two ways of initializing the weights, from which ANN starts learning. If the initial weights are good, ANN needs less time to learn otherwise it requires more time and/or stuck in local minima

1. Random selection
2. Using evolutionary algorithm

4.3.1.7 Faster Training in Back-Propagation Learning

Plain back-propagation is terribly slow and it is desired to have faster training. There are a series of things that can be done to speed up training.

- Fudge the derivative term.
- Scale the data.
- Direct input–output connections.
- Vary the sharpness (gain) of the activation.
- Use a different activation.
- Use better algorithms.

1. *Fudge the derivative term*
   The first major improvement to back-propagation is extremely simple: fudge the derivative term in the output layer. If we are using the sigmoid function given by
   $$f(net) = \frac{1}{1 + \exp(-\lambda\, net)} \tag{4.34}$$

and the derivative is

$$f'(net) = f(net)\,(1 - f(net))\,. \tag{4.35}$$

The derivative is largest at $net = 0.5$ and it is here that we will get the largest weight changes. Unfortunately, at values near 0 or 1, the derivative term gets close to 0 and the weight change becomes very small. In fact, if the network's response is 1 and the target is 0, the network is off by quite a lot with very small weight changes. It can take a very long time for the training process to correct this. Falhlman's solution was to add 0.1 to the derivative term making it:

$$f'_{new}(net) = 0.1 + f'(net). \tag{4.36}$$

The solution of Chen and Mass was to drop the derivative term altogether, in effect, the derivative was 1. This method passes back much larger error quotas to the lower layer, so large that a smaller η must be used there. In their experiments on 10-5-10 codec problem, they found that the best results came when η was 0.1 times the upper level η; hence they called this method the "differential step size" method. One must experiment with both upper and lower level η values to get the best results depending on the problem. Besides that, the η used for the upper layer must be much smaller that the η used without this method.

2. *Direct input–output connections*:
   Adding direct connections from the input layer to the output layer can often speed up training. It is supposed to work best when the function to be approximated is almost linear and it only needs a small amount of adjustment from non-linear hidden layer units. This method can also cut down the number of hidden layer units needed. It is not recommended when there are a large number of output units because, then there are more free parameters to the net and possibly hurt the generalisation.
3. *Adjusting the sharpness (gain) of the activation*:
   Izni and Pentland showed that training time can be decreased by increasing the sharpness or gain λ in the standard sigmoid as given in (4.34). In fact, they showed that the training time goes as $1/\lambda$ for training without momentum and $1/\sqrt{\lambda}$ for networks with momentum. This is not a perfect speed-up scheme since when λ is too large, we run the risk of becoming trapped in a local minimum. Sometimes the best value for λ is less that 1.

4.3.1.8 Better Algorithms

Everyone wants faster training and there are many variations on backpropagation that will speed up the training time enormously, but the credibility of these variations are at question, at times. Very slow online update methods will sometimes give the best results when compared to these acceleration algorithms. People have observed this with sonar data: the best results come from one pattern at a time updates. Having said this, in most cases, the acceleration algorithms work much faster than either online or batch training that they should be used first and then if better results are wanted, one can try slower online methods.

In Sect. 5.4, we have already discussed the effect of having adaptive learning rate and momentum. As the training proceeds, increase η and α adaptively if we keep going downhill, in terms of error. When the weight change gets too large, we end up on the other side of the valley and for this, we must decrease the learning rate and momentum in some way. These are basically first-order algorithms, where we use only the first order information of the error gradient in weight updating. Then there is a set of algorithms known as conjugate gradient methods, which use second order information also for faster training. We will discuss a few such algorithms.

- The Resilient propagation Algorithm.
- The Delta-Bar-Delta Algorithm.
- The Quick-propagation Algorithm.
- The Conjugate Gradient methods.

(a) *The Resilient propagation algorithm*:
The Resilient propagation is a first-order algorithm performing supervised batch learning in multi-layered perceptrons. The basic principle of Rprop is to eliminate the harmful influence of the size of the partial error derivative on the weight step. As a consequence, only the sign of the derivative is considered to indicate the *direction* of weight update. The *size* of the weight change is exclusively determined by a weight-specific, so called "update-value" ${\Delta_{ij}}^{(t)}$:

$$\Delta w_{ij}^{(t)} = \begin{cases} -\Delta_{ij}^{(t)};\ if \dfrac{\partial E}{\partial w_{ij}}^{(t)} > 0 \\ +\Delta_{ij}^{(t)};\ if \dfrac{\partial E}{\partial w_{ij}}^{(t)} < 0 \\ 0;\ otherwise \end{cases} \tag{4.36}$$

where $\frac{\partial E}{\partial w_{ij}}^{(t)}$ denotes the summed gradient information over the patterns of the pattern set ("batch learning").

It should be note that, by replacing the ${\Delta_{ij}}^{(t)}$ by a constant update-value Δ, (4.36) yields the so-called "Manhattan" Algorithm.

The second step of Rprop learning is to determine the new update values ${\Delta_{ij}}^{(t)}$. This is based on a sign-dependent adaptation process.

$$\Delta_{ij}^{(t)} = \begin{cases} \eta^{+} \times \Delta_{ij}^{(t)};\ if \dfrac{\partial E}{\partial w_{ij}}^{(t-1)} \times \dfrac{\partial E}{\partial w_{ij}}^{(t)} > 0 \\ \eta^{-} \times \Delta_{ij}^{(t)};\ if \dfrac{\partial E}{\partial w_{ij}}^{(t-1)} \times \dfrac{\partial E}{\partial w_{ij}}^{(t)} < 0 \\ \Delta_{ij}^{(t)};\ otherwise \end{cases} \tag{4.37}$$

where $0 < \eta^{-} < 1 < \eta^{+}$.

In other words, the adaptation rule works as follows: Every time, the partial derivative of the corresponding weight w_{ij} changes its sign, which indicates that the last update was too big and the algorithm has jumped over local minimum, the update value ${\Delta_{ij}}^{(t)}$ is decreased by the factor η^-. If the derivative retains the sign, the update value is slightly increased in order to accelerate the convergence in shallow regions. Additionally, in case of a change in sign, there should be no adaptation in the succeeding learning step. In practice, this can be achieved by setting ${\frac{\partial E}{\partial w_{ij}}}^{(t)} = 0$.

In order to reduce the number of freely adjustable parameters, often leading to a tedious search in parameter space, the increase and decrease factor are set to fixed values. The choice of decrease factor η^- was lead by the following considerations. If a jump over a minimum occurred, the previous update value was too large, for, it cannot be derived from gradient information how much the minimum was missed. We have to estimate the correct value. It will be a good guess to halve the update value (maximum likelihood estimator), so we choose $\eta^- = 0.5$. The increase factor η^+, on the other hand, has to be large enough to allow fast growth of the update values in shallow regions of error function, but, on the other hand, the learning process can be considerably disturbed if a too large increase factor leads to persistent changes of the direction of the weight-step. In several experiments, the choice of $\eta^+ = 1.2$ gave very good results, independent of examined problems. Slight variations of this value neither improve nor deteriorate convergence time.

(b) *Delta-Bar-Delta Algorithm*:
The Delta-Bar-Delta is a method that implements four heuristics regarding gradient descent. It was developed by Jacobs (1988). The method consists of a weight update rule and learning update rule. The weight update rule is applied to each weight $w_{ij}(n)$ at iteration n through the relationship given by

$$w_{ij}(n+1) = w_{ij}(n) - \eta_{ij}(n+1)\frac{\partial E(n)}{\partial w_{ij}(n)}, \tag{4.38}$$

where $\eta(n)$ is the learning rate for the weight $w_{ij}(n)$ at update iteration n.

The learning rate update rule for a given weight $w_{ij}(n)$ is defined as

$$\Delta\eta_{ij}(n) = \begin{cases} k & ; if\ \bar{\delta}_{ij}(n-1) \times \delta_{ij}(n) > 0 \\ -\phi\eta_{ij}(n) & ; if\ \bar{\delta}_{ij}(n-1) \times \delta_{ij}(n) < 0 \\ 0 & ; otherwise \end{cases} \tag{4.38}$$

where

$$\delta_{ij}(n) = \frac{\partial E(n)}{\partial w_{ij}(n)} \tag{4.39}$$

the partial derivative of the error with respect to $w_{ij}(n)$ at iteration n, and

$$\bar{\delta}_{ij}(n) = (1-\theta)\delta_{ij}(n) + \theta\bar{\delta}_{ij}(n-1) \tag{4.40}$$

where k and ϕ are constants used increment or decrement the learning rate respectively, and $0 < \theta < 1$ is an exponential “smoothing” base constant for the nth iteration.

The heuristics implemented are as follows:

1. Every parameter (weight) has its own individual learning rate.
2. Every learning rate is allowed to vary over time to adjust to changes in the error surface.
3. When the error derivative for a weight has the same sign for several consecutive update steps, the learning rate for that weight should be increased. This is because the error surface has a small curvature at such points and will continue to slope at the same rate for some distance. Therefore, the step-size should be increased to speed up the downhill movement.
4. When the sign of the derivative of a weight alternates for several consecutive steps, the learning rate for that parameter should be decreased. This is because the error surface has a high curvature at that point and the slope may quickly change sign. Thus, to prevent oscillation, the value of the step-size should be adjusted downward.

There are a few drawbacks of this algorithm. Using momentum along with the algorithm can enhance the performance; however, it can also make the search diverge wildly – especially if k is even moderately large. The reason is that momentum “magnifies” learning rate increments and quickly leads to inordinately large learning steps. One possible solution is to keep the k factor very small, but this can lead to slow increase in η and little speedup.

Another related problem is that, even with a small k, the learning rate can sometimes increase so much that the small exponential decrease is not sufficient to prevent wild jumps. Increasing ϕ exacerbates the problem instead of solving it because it causes drastic reduction of learning rate at inopportune moments, leaving the search stranded at points of high error. Thus the algorithm is very sensitive to small variations in the value of its parameters – especially k.

(c) *Quick-propagation algorithm*:
Standard back-propagation calculates the weight change based upon the first derivative of the error with respect to the weight. If the second derivative information is also available, then better step-size and optimum search direction can be found out. Back-propagation networks are also slow to train. Quick-propagation is a variation of standard back-propagation to speed up training.

The quickprop modification is an attempt to estimate and utilise the second derivative information (Fahlman 1988). This algorithm requires saving the previous gradient vector as well as previous weight change. The calculation of weight change uses only the information associated with the weight being updated.

$$\Delta w_{ij}(n) = \frac{\nabla w_{ij}(n)}{\nabla w_{ij}(n-1) - \nabla w_{ij}(n)} \times \Delta w_{ij}(n-1), \tag{4.41}$$

where $\nabla w_{ij}(n)$ is the gradient vector component associated with weight vector w_{ij} in step n, $\nabla w_{ij}(n-1)$ is the gradient vector component associated with weight w_{ij} in the previous step and $\Delta w_{ij}(n-1)$ is the weight change in step $n-1$.

A maximum growth factor μ is used to limit the rate of increase of step-size like
If $\Delta w_{ij}(n) > \mu \Delta w_{ij}(n-1)$, then $\Delta w_{ij}(n) = \mu \Delta w_{ij}(n-1)$
Fahlman suggested an empirical value 1.75 for μ.

There are some complications in this method. First is the step-size calculation that requires the previous value, which is not available at the time of starting. This is overcome by using the standard back-propagation method for weight adjustment. The gradient descent weight change is given by

$$w_{ij}(n+1) = w_{ij}(n) - \eta \nabla w_{ij} \tag{4.42}$$

Value of η is taken suitably small.

Second problem is that the weight values are unbounded. They become so large that they may cause an overflow. Suitable scaling of slope by a factor less than 1 reduces the rate of increase of the weights.

(d) *The conjugate gradient (CG) methods*:
The conjugate gradient algorithms have become very popular for training back-propagation networks. Just like all the second order methods, the CG algorithm is implemented in batch-mode. The CG algorithm can search the minimum of a multivariate function faster than the conventional gradient descent procedure for BP networks. Each conjugate gradient step is, at least, as good as the steepest descent method from the same point. The formula is simple and the memory usage is in the same order as the number of weights. Most important, the CG technique obviates the tedious tasks of determining optimal learning parameters. Moreover, the CG technique has very reliable convergence behaviour as compared with the first-order gradient methods.

The basic back-propagation algorithm adjusts the weights in the steepest descent direction, i.e. negative of the gradient. This is the direction in which the performance function is decreasing most rapidly. It turns out that although the function decreases most rapidly along the negative of the gradient, this does not necessarily produce faster convergence. In the conjugate gradient algorithms, a search is performed along the conjugate directions, which produces generally faster convergence (conjugate directions means at orthogonal directions), than steepest descent directions.

In most training algorithms that we have discussed up to this point, a *learning rate* is used to determine the length of the weight update (step-size). In CG methods, the step-size is adjusted at every iteration. A search is made along conjugate gradient directions to determine the step-size, which will minimise the performance function along that line.

The basic attempt in all second order enhancement methods is that the current search direction $\mathbf{d}(n)$ to be a compromise between the *exact* gradient $\nabla E(n)$ and the previous search direction $\mathbf{d}(n-1)$, i.e. $\mathbf{d}(n) = -\nabla E(n) + \beta\mathbf{d}(n-1)$ with $\mathbf{d}(0) = -\nabla E(0)$.

The search direction is chosen (by appropriately setting β) so that it distorts as little as possible the minimisation achieved by the previous step. In conjugate gradient methods, the current search is chosen to be conjugate to the previous search direction. Analytically, we require

$$\bar{d}(n-1)^t H(n-1)\bar{d}(n) = 0 \tag{4.43}$$

where the Hessian $H(n-1)$ is assumed to be positive definite (H is the Hessian matrix with components $H_{ij} = \frac{\partial^2 E}{\partial w_i \partial w_j}$).

β plays the role of an adaptive momentum and chosen according to the Polack–Ribiere rule

$$\beta = \beta(n) = \frac{[\nabla E(n) - \nabla E(n-1)]^t \, \nabla E(n)}{\|\nabla E(n-1)\|^2} \tag{4.44}$$

Thus the search direction in the conjugate gradient methods at iteration n is given by

$$\begin{aligned}\bar{d}(n) &= -\nabla E(n) + \beta\bar{d}(n-1) \\ &= -\nabla E(n) + \frac{[\nabla E(n) - \nabla E(n-1)]^t \, \nabla E(n)}{\|\nabla E(n-1)\|^2}\bar{d}(n-1)\end{aligned} \tag{4.45}$$

Now using $\mathbf{d}(n-1) = (1/\rho)\Delta\mathbf{w}(n-1)$ and substituting the preceding expression for $\mathbf{d}(n)$ in $\Delta\mathbf{w}(n)$ leads to the weight update rule:

$$\Delta\bar{w}(n) = -\rho\nabla E(n) + \beta(n)\Delta\bar{w}(n-1) \tag{4.46}$$

When E is quadratic, the conjugate methods theoretically converge in N or fewer iterations. In general, E is not quadratic, and therefore, this method would be slower than what theory predicts. However, it is reasonable to assume that E is approximately quadratic near a local minimum. Therefore, conjugate gradient descent is expected to accelerate the convergence of back-propagation once the search enters a small neighbourhood of a local minimum.

4.3.2 Selection of Error Functions

Normally in the supervised learning of multi-layer neural networks, sum squared error is used. There are many other error functions which may used for ANN training as given in Table 4.14.

These error functions, their derivates and delta functions have been plotted against error as shown in Fig. 4.9. The three-dimensional surfaces for some error function are shown in Fig. 4.10.

Table 4.14. Different error functions for ANN learning

1.	Sum square error	$½\sum e_i^2$
2.	Logarithmic error	$\sum[(1+y_pk)\mathrm{In}\{(1+y_pk)/(1+O_pk)\}]+$ $\sum[(1-y_pk)\mathrm{In}\{(1-y_pk)/(1-O_pk)\}]$
3.	Mean fourth power error	$\sum e_i^4/p$
4.	Hyperbolic square error	$\sum \mathrm{In}\{(1-e_i^2)/(1+e_i^2)\}$
5.	Hubber's error	$\sum e_i^2/2$ if $\lvert e_i\rvert < c$ $\sum c(e_i - c/2)$ if $\lvert e_i\rvert >= c$
6.	Cauchy's error	$\sum c^2[\mathrm{In}\{1+(e_i/c)^2\}]/2$
7.	Geman–McClure error	$\sum e_i^2/\{2(1+e_i^2)\}$
8.	Welsch error	$\sum c^2[1-e_i-(e_i/c)^2]/2$
9.	fair's error	$\sum c^2[e_i/c - \mathrm{In}(1-e_i/c)]$
10.	Mean median error	$\sum 2[(1+e_i/2)1/2-1]$
11.	Log-cos-hyperbolic error (Tasos Falas 1999)	$\sum \mathrm{In}[\cos\ h(e_i^2)]$
12.	Andrew error	$\sum \cos(\pi^* e_i)/\pi 2$ If $e_i <= 1$ $\sum e_i$ If $e_i > 1$
13.	Entropy error	$-\mathrm{In}(1-e_i)$
14.	Hamlet error	$e_i - \mathrm{In}(1-e_i)$
15.	Fahlman error	$\sum[(1+e_i)\mathrm{In}(1+e_i)+(1-e_i)\mathrm{In}(1-e_i)]$

4.3.3 Mode of Error Calculation

The error can be calculated in the pattern mode or in batch mode. In pattern mode of error calculation the error is calculated after present each pattern, i.e. one set of training inputs, which is used to modify the weights. In batch mode all the patterns are presented and the errors are calculated for each pattern and then sum square error is used to modify the weights.

4.4 Summary

Finally it could be concluded that the back propagation feedforward neural networks training and testing performance is dependent on network complexity, problem complexity and complexity of learning algorithm. In this chapter it has been found that:

1. Tan sigmoid activation function at hidden layer and pure linear at output layer is taking less training epochs and also giving very good results during testing. Pure linear–pure linear combination is also taking same training time but the results predicted with this pair is not very encouraging.
2. Prediction accuracy is comparable for different mappings. However training time is minimum for X-Y type of mapping (it requires only 98, 56, and 35 epochs for dc machine current, speed predictions and electrical load forecasting problem, respectively. The other mappings require significantly large training epochs.

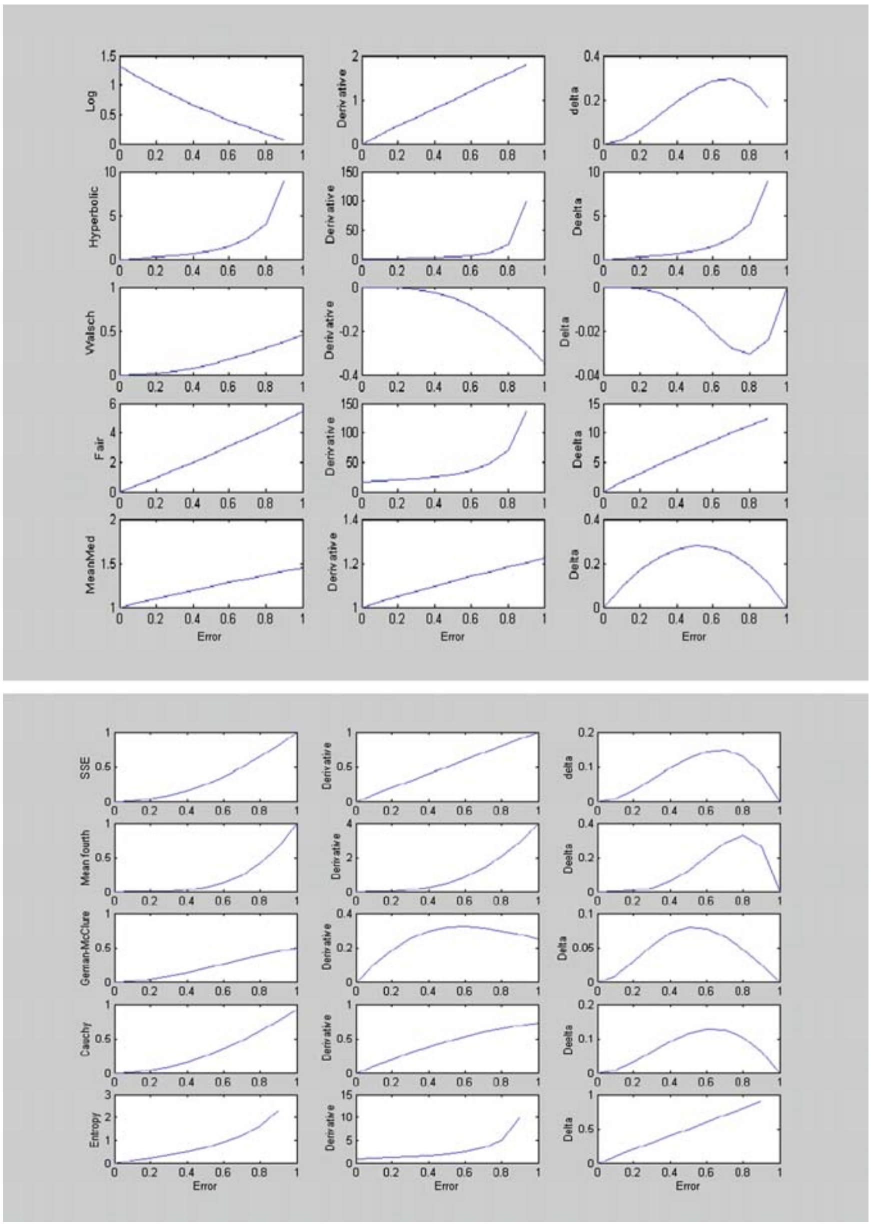


Fig. 4.9. Error functions, their derivates and delta functions for different error value

3. Xnoise–Y mapping is able to train up to error level 0.00265015 in 400 training epochs and the error during testing is also low as compared to the other noisy mappings as shown in Table 2.4.

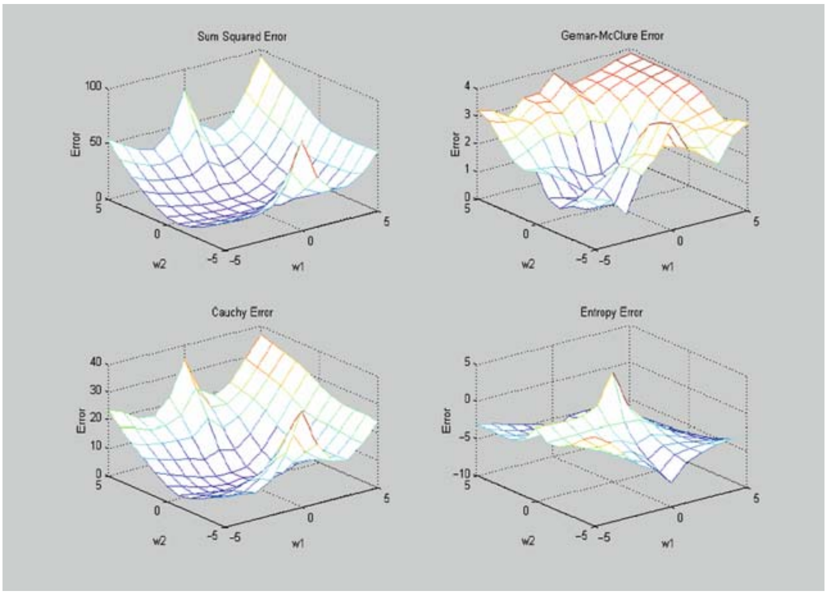


Fig. 4.10. Error surfaces for different error functions

4. Normalization ranges −0.9 to 0.9 for input and 0.1 to 0.9 for output are found very satisfactory for almost all problems.
5. They change the weights each time by some fraction of the change needed to completely correct the error. This fraction, ß, is called learning rate.
6. High learning rates cause the learning algorithm to take large steps on the error surface, with the risk of missing a minimum, or unstably oscillating across the error minimum.
7. Small steps, from a low learning rate, eventually find a minimum, but they take a long time to get there.
8. Some NN simulators can be set to reduce the learning rate as the error decreases.
9. Local minima problem can be avoided by introducing the momentum term.
10. To increase the speed of back propagation learning algorithms, adaptive learning rate and momentum factor is considered.
11. The error tolerance also affects the training time and generalization capabilities of ANN.

4.5 Bibliography and Historical Notes

In multilayer ANN, it is reported that the training is very time-consuming phase. Among the different approaches suggested to ease the back-propagation training process, input data pre-treatment has been pointed out, although no specific procedure has been proposed. We have found that input data normalization with certain criteria, prior to a training process, is crucial to obtain good results as well as to fasten significantly the calculations. There are some researchers (Sevilla Sola 1997; Kartam 1997) reported that how data normalization affects the training performance of ANN.

4.6 Exercises

1. Consider a neuron whose activation function is sigmoid $f(x) = \frac{1}{1+e^{-\lambda x}}$
   a. Prove that the derivative of f(x) with respect to x is given as $f'(x) = \lambda.f(x)(1 - f(x))$.
   b. Write a MATLAB program for plotting the activation function and its derivative for different values of x for $\lambda = 0.1, 0.5, 1.0$.
2. Repeat (a) and (b) parts of questions 1 with hyperbolic activation function.
3. Implement the following logic gates using feedforward bckpropagation ANN with one, two and three hidden layers:
   a. NAND gate
   b. NOR gate
   c. EX-OR gate.
4. Repeat question 3 for product aggregation neuron and compare the training and testing performance of above ANN and Product ANN.
5. Write a step by step procedure for backpropagation algorithms.
6. Study the effect of different intial weights on the training performance on backpropagation learning algorithms.
7. Write a MATLAB program to generate at least 100 training and 25 testing data for the following function

$$f(x) = x^{*}e^{x}.$$

   a. Develop ANN model to map this function and compare the results for fixed parameter backpropagation and adaptive backpropagation learning.
   b. Also train the ANN model using 5% noise in the training data and test with actual data.