Chapter 5

Flat Neural Networks and Rapid Learning Algorithms

In this chapter, we will introduce flat neural networks architecture. The system equations of flat neural networks can be formulated as a linear system. In this way, the performance index is a quadratic form of the weights, and the weights of the networks can be solved easily using a linear least-square method. Even though they have a linear-system-equations-like equation, the flat neural networks are also perfect for approximating non-linear functions. A fast learning algorithm is given to find an optimal weight of the flat neural networks. This formulation makes it easier to update the weights instantly for both a newly added input and a newly added node. A dynamic stepwise updating algorithm is given to update the weights of the system instantly. Finally, we give several examples of applications of the flat neural networks, such as an infrared laser data set, a chaotic time-series, a monthly flour price data set, and a non-linear system identification problem. The simulation results are compared to existing models in which more complex architectures and more costly training are needed. The results indicate that the flat neural networks are very attractive to real-time processes.

5.1 Introduction

Feedforward artificial neural networks have been a popular research subject recently. The research topics vary from the theoretical view of learning algorithms such as learning and generalization properties of the networks to a variety of applications in control, classification, biomedical, manufacturing, and business forecasting, etc. The backpropagation (BP) supervised learning algorithm is one of the most popular learning algorithms being developed for layered networks [1-2]. Improving the learning speed of BP and increasing the generalization capability of the networks have played a center role in neural network research [3-9]. Apart from multi-layer network architectures and the BP algorithm, various simplified architectures or different non-linear activation functions have been devised. Among those, so-called flat networks including functional-link neural networks and radial basis function networks have been proposed [10-15]. These flat networks remove the drawback
of a long learning process with the advantage of learning only one set of weights. Most importantly, the literature has reported satisfactory generalization capability in function approximation [14-16].

This chapter discusses the flat networks along with an one-step fast learning algorithm and a stepwise update algorithm for training the flat networks. Although only the functional-link network is used as a prototype here, the algorithms can also be applied to the radial basis function network. The algorithms are developed based on the formulation of the functional-link network that has a set of linear system equations. Because the system equations of the radial basis function network have a similar form to the functional-link network and both networks share similar “flat” architecture, the update algorithm can be applied to the radial basis function network as well. The most significant advantage of the stepwise approach is that the weight connections of the network can be updated easily, when a new input is given later after the network has been trained. The weights can be updated easily based on the original weights and the new inputs. The stepwise approach is also able to update weights instantly when a new neuron is added to the existing network if the desired error criterion cannot be met. With this learning algorithm, the flat networks become very attractive in terms of learning speed.

Finally, the flat networks are used for several applications. These include an infrared laser data set, a chaotic time-series, a monthly flour price data set, and a non-linear system identification. The time-series is modeled by the AR(p) (Auto-Regression with p delay) model. During the training stage, a different number of nodes may be added as necessary. The update of weights is carried by the given algorithm. Contrary to the traditional BP learning and multi-layer models, the training of this network is fast because of an one-step learning procedure and the dynamic updating algorithm. We also applied the networks to non-linear system identification problems involving discrete-time single-input, single-output (SISO), and multiple-input, multiple-output (MIMO) plants which can be described by the difference equations [16]. With this learning algorithm, the training is easy and fast. The result is also very promising.

This chapter is organized as follows, wherein Section 2 briefly discusses the concept of the functional-link and its linear formulation. Sections 3 and 4 introduce the dynamic stepwise update algorithm followed by the refinement of the model in section 5. Section 6 discusses the procedures of the training. Finally, several examples and conclusions are given.

5.2 The Linear System Equation of the Functional-link Network

Figure 1 illustrates the characteristic flatness feature of the functional-link network. The network consists of a number of “enhancement” nodes. These enhancement nodes are used as extra inputs to the network. The weights from input nodes to the enhancement nodes are randomly generated and fixed thereafter. To be more
precise, an enhancement node is constructed by first taking a linear combination of the input nodes, and then applying a nonlinear activation function $\xi(.)$ to it. This model has been discussed elsewhere by Pao [10]. A rigorous mathematical proof has also been given by Igelnik and Pao [12]. The literature has also discussed the advantage of the functional-link network in terms of training speed and its generalization property over the general feedforward networks [11]. In general, the functional-link network with $k$ enhancement nodes can be represented as an equation of the form:

$$\mathbf{Y} = \left[\mathbf{x}\right]|\xi\left(\mathbf{x}\mathbf{W}_h + \beta_h\right)|\mathbf{W}$$

(5.1)

where $W_h$ is the enhancement weight matrix, which is randomly generated, $\mathbf{W}$ is the weight matrix that needs to be trained, $\beta_h$ is the bias function, $\mathbf{Y}$ is the output matrix, and $\xi(.)$ is a non-linear activation function. The activation function can be either a sigmoid or a $\tanh$ function. If the $\beta_h$ term is not included, an additional constant bias node with -1 or +1 is needed. This will cover even function terms for function approximation applications, which have been explained using Taylor series expansion in Chen [17].

Denoting by $\mathbf{A}$ the matrix $[\mathbf{x}|\xi(\mathbf{x}\mathbf{W}_h + \beta_h)]$, where $\mathbf{A}$ is the expanded input matrix consisting of all input vectors combined with enhancement components, yields:

$$\mathbf{Y} = \mathbf{A}\mathbf{W}$$

(5.2)

The structure is illustrated in Figure 2.

Figure 1 A flat Functional-link neural network
Pao implemented a conjugate gradient search method that finds the weight matrix, $W$ [11]. This chapter discusses a rapid method of finding the weight matrix. To learn the optimal weight connections for the flat network, it is essential to find the least-square solution of the equation, $AW = Y$. Recall that the least-square solution to the equation, $Y = AW$, is $W = A^+ Y$, where $A^+$ is the pseudoinverse of matrix $A$. To find the best weight matrix $W$, the REIL (Rank Expansion with Instant Learning) algorithm is described in the following [17].

Algorithm Rank-Expansion with Instant Learning (REIL).

Input: The extended input pattern matrix, $A$, and the output matrix, $Y$, where $N$ is the number of the input patterns.
Step 1. Add $k$ hidden nodes and assign random weights, $k \leq N - r$, where $r$ is the rank of $A$.
Step 2. Solve weight, $W$, by minimizing $\|AW - Y\|_2$.
Step 3. If mean-squared error criterion is not met, add additional nodes and go to Step 2; otherwise, stop.

End of Algorithm REIL

The computation complexity of this algorithm comes mostly from the time spent in Step 2. There are several methods for solving least-squares problems [18]. The complexity of FLOP count is the order of $O(Nq^2 + q^3)$, where $N$ is the number of rows in the training matrix, and $q$ is the number of columns. The singular value decomposition is the most common approach. Compared with gradient descent.
search, the least-squares method is time efficient [19].

The above algorithm is a batch algorithm in which we assume that all the input data are available at the time of training. However, in a real-time application, as a new input pattern is given to the network, the \( \mathbf{A} \) matrix must be updated. It is not efficient at all if we continue using the REIL algorithm. We must pursue an alternative approach. Here we take advantage of the flat structure in which extra nodes can be added and the weights can be found very easily if necessary. In addition, weights can be easily updated without running a complete training cycle when either one or more new enhancement nodes are added, or more observations are available. The stepwise updating of the weight matrix can be achieved by taking the pseudoinverse of a partitioned matrix described below [20, 21]. Let us denote prime (\(^{\prime}\)) as the transpose of a matrix. Let \( \mathbf{A}_n \) be the \( n \times m \) pattern matrix defined above, and \( \mathbf{a}' \) be the \( m \times 1 \) new pattern entered to the neural network. Here the subscript denotes the discrete time instance. Denote \( \mathbf{A}_{n+1} \) as the following.

\[
\mathbf{A}_{n+1} = \begin{bmatrix} \mathbf{A}_n & \mathbf{a}' \end{bmatrix},
\]

then the theorem states that the pseudoinverse of the new matrix \( \mathbf{A}_{n+1} \) is

\[
\mathbf{A}_{n+1}^+ = [\mathbf{A}_n^+ - \mathbf{b} \mathbf{d}' | \mathbf{b}],
\]

where

\[
d' = \mathbf{a}' \mathbf{A}_n^+ \text{ and } \mathbf{b} = \begin{cases} (\mathbf{c}')^+, & \text{if } \mathbf{c} \neq 0, \\ (1 + \mathbf{d}' \mathbf{d})^{-1} \mathbf{A}_n^+ \mathbf{d}, & \text{if } \mathbf{c} = 0,
\end{cases}
\]

where

\[
\mathbf{c}' = \mathbf{a}' - \mathbf{d}' \mathbf{A}_n.
\]

In other words, the pseudoinverse of \( \mathbf{A}_{n+1} \) can be obtained through \( \mathbf{A}_n^+ \) and the added row vector \( \mathbf{a}' \). A noteworthy fact is that, if \( n > m \) and \( \mathbf{A}_n \) is of full rank, then \( \mathbf{c} = 0 \). This can be shown as follows. If \( \mathbf{A}_n \) is of full rank and \( n > m \), then

\[
\mathbf{A}_n^+ = (\mathbf{A}_n^+ \mathbf{A}_n)^{-1} \mathbf{A}_n^+
\]

therefore

\[
\mathbf{c}' = \mathbf{a}' - \mathbf{d}' \mathbf{A}_n
\]

\[
= \mathbf{a}' - \mathbf{a}' (\mathbf{A}_n^+ \mathbf{A}_n)^{-1} \mathbf{A}_n \mathbf{A}_n
\]

\[
= \mathbf{a}' (\mathbf{I} - (\mathbf{A}_n^+ \mathbf{A}_n)^{-1} \mathbf{A}_n \mathbf{A}_n)
\]

\[
= 0.
\]

So the pseudoinverse of \( \mathbf{A}_{n+1} \) can be updated based only on \( \mathbf{A}_n^+ \), and the new added row vector \( \mathbf{a}' \) without recomputing the entire new pseudoinverse.

Let the output vector \( \mathbf{Y}_{n+1} \) be partitioned as \( \mathbf{Y}_n \),

\[
\mathbf{Y}_{n+1} = \begin{bmatrix} \mathbf{Y}_n & \mathbf{y}' \end{bmatrix},
\]

© 2001 by CRC Press LLC
where \( y' \) is the new output corresponding to the new input \( a' \) and let

\[
W_{n+1} = A_{n+1}^+ Y_{n+1}, \quad W_n = A_n^+ Y_n.
\]

Then according to the above equations, the new weight, \( W_{n+1} \), can be found as below.

\[
W_{n+1} = W_n - (y' - a' W_n)b. \tag{5.3}
\]

Equation (5.3) has the same form with the recursive least-square solution, if \( c = 0 \). However, Equation (5.3) considers the case if \( A_n \) is not the full-rank (i.e., \( c \neq 0 \)). Compared to the Least Mean Square (LMS) learning rule [22], Equation (5.3) has the optimal learning rate, \( b \), which leads the learning in one-step update, rather than iterative update. The stepwise updating in flat networks is also perfect for adding a new enhancement node to the network. In this case, it is equivalent to add a new column to the input matrix \( A_n \). Denote \( A_{n+1} \stackrel{\Delta}{=} [A_n | a] \). Then the pseudoinverse of the new \( A_{n+1}^+ \) equals

\[
\begin{bmatrix}
A_n - db' \\
b'
\end{bmatrix},
\]

where \( d = A_n^+ a \),

\[
b' = \begin{cases}
(c)^+, & \text{if } c \neq 0, \\
(1 + d'd)^{-1} A_n^+, & \text{if } c = 0,
\end{cases}
\]

and \( c = a - A_n d \). Again the new weights are

\[
W_{n+1} = \begin{bmatrix}
W_n - db' Y_n \\
b' Y_n
\end{bmatrix}, \tag{5.4}
\]

where \( W_{n+1} \) and \( W_n \) are the weights after and before a new neuron is added, respectively. Since a new neuron is added to the existing network, the weights, \( W_{n+1} \), have one more dimension than \( W_n \). Also note again that, if \( A_n \) is of the full rank, then \( c = 0 \) and no computation of pseudoinverse is involved in updating the pseudoinverse \( A_n^+ \) or weight matrix \( W_n \).

The one-step dynamic learning is shown in Figure 3. This raises the question of the rank of input matrix \( A_n \). As can be seen from the above discussion, it is desirable to maintain the full rank condition of \( A_n \) when adding rows and columns. The rows consist of training patterns. In other words, it is practically impossible to observe any rank deficient matrix. Thus, during the training of the network, it is our advantage to make sure that the added nodes will increase the rank of input matrix. Also if the matrix becomes numerically rank deficient based on the adjustable tolerance on the singular values, we should consider removing the redundant input nodes. This is discussed in more detail in Section 5 on principal component analysis (PCA) related topics.

© 2001 by CRC Press LLC
Another approach to achieve stepwise updating is to maintain the Q-R decomposition of the input matrix $A_n$. The updating of the pseudoinverse (and therefore the weight matrix) involves only a multiplication of finitely sparse matrices and backward substitutions. Suppose we have the Q-R decomposition of $A_n$ and denote $A_n \triangleq QR$, where $Q$ is an orthogonal matrix and $R$ is an upper triangular matrix. When a new row or a new column is added, the Q-R decomposition can be updated based on a finite number of Givens rotations [18]. Denote $A_{n+1} \triangleq \hat{Q}\hat{R}$ where, $\hat{Q}$ remains orthogonal, $\hat{R}$ is an upper triangular matrix, and both are obtained through finitely many Givens rotations. The pseudoinverse of $A_{n+1}$ is
\[
A_{n+1}^{\dagger} \triangleq \hat{R}^{-1} \hat{Q}^{-1},
\]
where $\hat{R}^{-1}$ (eventually, $W_{n+1}$) can be computed by backward substitution. This stepwise weight update using Q-R and Givens rotation matrix is summarized in the following algorithm.
Q-R Implementation of Weights Matrix Updating

Input: $A_n \triangleq QR$, vector $a$, and weight matrix $W_n$, where $A_n$ is a $n \times m$ matrix, $Q$ is a $n \times n$ orthogonal matrix, $R$ is a $n \times m$ upper triangular matrix, and $a'$ is a $m \times 1$ row vector.

Output: $A_{n+1} \triangleq \begin{bmatrix} A_n^T \\ a \end{bmatrix}$ $\triangleq \begin{bmatrix} Q \hat{R} \end{bmatrix}$ and weight matrix $W_{n+1}$, where $A_{n+1}$ is a $(n+1) \times m$ matrix, $Q$ is a $(n+1) \times (n+1)$ orthogonal matrix, and $\hat{R}$ is a $(n+1) \times m$ upper triangular matrix.

Step 1. Expand $\hat{Q}$ and $\hat{R}$, i.e.,

$$\hat{Q} \leftarrow \text{diag}(Q, 1) \triangleq \begin{bmatrix} Q & 0 \\ 0 & 1 \end{bmatrix},$$

$$\hat{R} \leftarrow \begin{bmatrix} R \\ a' \end{bmatrix}.$$

Step 2. For $i = 1$ to $m$, do

$$J_i = Rot(r_i, e_i),$$

$$\hat{Q} \leftarrow \hat{Q} * J_i,$$

$$\hat{R} \leftarrow J_i \ast \hat{R}.$$

Step 3. Since $\hat{R}$ is an upper triangular matrix, the new $W_{n+1}$ can be easily obtained by solving $\hat{R} W_{n+1} = \hat{Q} \begin{bmatrix} Y^n \\ y \end{bmatrix}$ using backward substitution.

End of the Q-R Weight-Updating Algorithm

In Step 2, $J_i$ is the Givens rotation matrix, $r_i$ is the $i$-th column of $\hat{R}$ and $e_i$ is a column vector identical to $r_i$ except the $i$-th and $(n+1)$-th components. The $n+1$ component is 0. $Rot$ performs a plane rotation from vector $r_i$ to $e_i$. An example should make this clear. If $r_2 = (1, 3, 0, 0, 4)'$ then $J_2$ rotates $r_2$ to $e_2 = (1, 5, 0, 0, 0)'$. In fact, $J_2$ transforms the plane vector $(3, 4)$ to $(5, 0)$ and keeps other components unchanged. The resulting $\hat{R}$ is an upper triangular matrix while $\hat{Q}$ remains orthogonal.

Similarly, with a few modifications, the above algorithm can be used to update the new weight matrix if a new column (a new node) is added to the network.

© 2001 by CRC Press LLC
5.4 Training with Weighted Least Squares

In training and testing the fitness of a model, error is minimized in the sense of least mean-squares, that is in general,

$$ E = \frac{1}{N}\|Y - AW\|_2, \quad (5.5) $$

where $N$ is the number of patterns. In other words, the average difference between network output and actual output is minimized over the span of the whole training data. If an overall fit is hard to achieve, it might be reasonable to train the network so that it achieves a better fit for most recent data. This leads to the so-called weighted least-squares problem. The stepwise updating of weight matrix based on weighted least-squares is derived as follows.

Let $K_n = \text{diag}(\Theta^{n-1}, \Theta^{n-2}, \ldots, \Theta, 1)$ be the weight factor matrix. Also, let $A_n$ represent input matrix with $n$ patterns and $A_{n+1}$ is $A_n$ with an added new row, that is,

$$ A_{n+1} \triangleq \begin{bmatrix} A_n \\ a \end{bmatrix}. $$

Then the weighted least-squares error for the equation $A_n W_n = Y_n$ is:

$$ E = \frac{1}{N}\|K_n(Y_n - A_n W_n)\|_2. \quad (5.6) $$

With

$$ K_{n+1} \triangleq \text{diag}(\Theta^n, \Theta^{n-1}, \ldots, \Theta, 1) \triangleq \text{diag}(\Theta K_n, 1), \quad (5.7) $$

we have

$$ K_{n+1}^{1/2} A_{n+1} = \begin{bmatrix} \Theta^{1/2} K_n^{1/2} & 0 \\ 0 & 1 \end{bmatrix} \begin{bmatrix} A_n \\ a \end{bmatrix} = \begin{bmatrix} \Theta^{1/2} K_n^{1/2} A_n \\ a \end{bmatrix}. \quad (5.8) $$

The weighted least-squares solution can be represented as

$$ W_n = (K_n^{1/2} A_n)^+ K_n^{1/2} Y_n. \quad (5.9) $$

If $S_n \triangleq (K_n^{1/2} A_n)^+$ is known and a new pattern (i.e., a new row $a'$) is imported to the network, then the weighted pseudoinverse of matrix $S_{n+1} \triangleq (K_{n+1}^{1/2} A_{n+1})^+$ can be updated by

$$ S_{n+1} = [(\Theta^{1/2} K_n^{1/2} A_n)^+ - bd'|b] $$

$$ = [\Theta^{-1/2} S_n - bd'|b], \quad (5.10) $$

where,

$$ d' = a'(\Theta^{1/2} K_n^{1/2} A_n)^+ $$

$$ = \Theta^{-1/2} a'S_n, $$
\[
\mathbf{b} = \begin{cases} 
(c')^+, & \text{if } c \neq 0 \\
(1 + d'd)^{-1}(\Theta^{1/2}\mathbf{K}^{1/2}_{n}A_{n})^+d, & \text{if } c = 0 \\
(1 + \Theta^{-1}a'S_{n}S'_{n}a)^{-1}\Theta^{-1/2}S_{n}\Theta^{-1/2}S'_{n}a \\
(\Theta + a'S_{n}S'_{n}a)^{-1}S_{n}S'_{n}a, \\
\end{cases}
\]

Similar to Equation (5.3), the updating rule for the weight matrix is

\[
W_{n+1} = W_n + (y' - a'W_n)b, \quad (5.11)
\]

the updating rule for the weight matrix, if \(A_n\) is of full rank (i.e., \(c = 0\)), is

\[
W_{n+1} = W_n + (\Theta + a'S_{n}S'_{n}a)^{-1}(y' - a'W)S_{n}S'_{n}a. \quad (5.12)
\]

Equation (5.12) is exactly the same as the weighted recursive least-squares method [23] in which only the full-rank condition is discussed. However, Equation (5.11) is more complete because it covers both \(c = 0\) and \(c \neq 0\) cases. Thus, the weighted weight matrix \(W\) can be easily updated based on the current weights and new observations, without running a complete training cycle, as long as the weighted pseudoinverse is maintained. Similar derivation can be applied to the network with an added neuron.

### 5.5 Refine the Model

Let us take a look again at an input matrix \(A\) of size \(n \times m\), which represents \(n\) observations of \(m\) variables. The singular value decomposition of \(A\) is:

\[
A = U \sum V',
\]

where \(U\) is a \(n \times n\) orthogonal matrix of the eigenvectors of \(AA'\) and \(V\) a \(m \times m\) orthogonal matrix of eigenvectors of \(A'A\). \(\sum\) is a \(n \times m\) ‘diagonal’ matrix whose diagonals are singular values of \(A\). That is,

\[
\sum = \begin{bmatrix}
\sigma_1 & 0 & \cdots & 0 & \cdots & 0 \\
0 & \sigma_2 & \cdots & 0 & \cdots & 0 \\
0 & 0 & \cdots & 0 & \cdots & 0 \\
0 & 0 & \cdots & \sigma_r & \cdots & 0 \\
0 & 0 & \cdots & 0 & \cdots & 0 \\
0 & 0 & \cdots & 0 & \cdots & 0
\end{bmatrix} \triangleq \begin{bmatrix}
\sigma & 0 \\
0 & 0
\end{bmatrix},
\]

where \(r\) is the rank of matrix \(A\). \(AA'\) is the so-called correlation matrix, whose eigenvalues are squares of the singular values. Small singular values might be the result of noise in the data or due to round off errors in computations. This can
lead to very large values of weights because the pseudoinverse of $A$ is given by 

$$A^+ = V \Sigma^+ U',$$

where 

$$\Sigma^+ = \begin{bmatrix} \frac{1}{\sigma} & 0 \\ 0 & 0 \end{bmatrix}.$$

Clearly, small singular values of $A$ will result in a very large value of weights which will, in turn, amplify any noise in the new data. The same question arises as more and more enhancement nodes are added to the model during the training.

A possible solution is to round off small singular values to zeros and therefore avoid large values of weights. If there is a gap among all the singular values, it is easy to cutoff at the gap. Otherwise, one of the following approaches may work:

i) Set an upper bound on the norm of weights. This will provide a criterion to cutoff small singular values. The result is an optimal solution within a bounded region.

ii) Investigate the relation between the cutoff values and the performance of the network in terms of prediction error. If there is a point where the performance is not improved when small singular values are included, it is then reasonable to set a cutoff value corresponding to that point.

The orthogonal least squares learning approach is another way to generate a set of weights that can avoid an ill-conditioning problem [13]. Furthermore, regularization and cross-validation methods are the techniques to avoid both overfitting and generalization problems [24].

### 5.6 Time-series Applications

The literature has discussed time-series forecasting using different neural network models [25, 26]. Here the algorithm proposed above is applied to the forecasting model. Represent the time-series by the AR($p$) (Auto-Regression with $p$ delay) model. Suppose $X$ is a stationary time-series. The AR($p$) model can be represented as the following equation:

$$X_t = (\lambda_1 X_{t-1} + \lambda_2 X_{t-2} + \cdots + \lambda_p X_{t-p}) + \varepsilon_t,$$

where $\lambda_i$’s are autoregression parameters.

In terms of a flat neural network architecture, the AR($p$) model can be described as a functional-link network with $p$ input nodes, $q$ enhancement nodes, and a single output node. This will artificially increase the dimension of the input space, or the rank of the input data matrix. The network includes $p + q$ input nodes and a single output node. During the training stage, a variable number of enhancement nodes may be added as necessary. Contrary to the traditional error backpropagation models, the training of this network is fast because of the one-step learning procedure and dynamic updating algorithm mentioned above. To improve the performance in some special situations, a weighted least-square criterion may be used to optimize the weights instead of the ordinary least-squares error.
Using the stepwise updating learning, this section discusses the procedure of training the neural network for time-series forecasting. First, available data on a single time-series are split into a training set and testing set. Let the time data, \( x(i+k) \), be the \( k \)-th time step after the data \( x(i) \) and assume that there will be \( N \) training data points. The training stage proceeds as follows.

Step 1. Construct Input and Output: Build an input matrix of size \((N - p) \times p\), where \( p \) is the delay-time. The \( i \)-th row consists of \([x(i+0), x(i+1), \ldots, x(i+(p-1))]\). The target output vector \( Y_n \) will be produced using \([x(i+p), \ldots, x_i(N)]'\).

Step 2. Obtain the weight matrix: Find the pseudoinverse of \( A_n \) and the weight matrix \( W_n = A_n^+Y_n \). This will give the linear least-square fit with \( p \) lags, or AR\((p)\). Predictions can be produced either single step ahead or iterated prediction. The network outputs are then compared to the actual continuation of the data using testing data. The error will be large most of the time, especially when we deal with a non-linear time-series.

Step 3. Add a new enhancement node if the error is above the desired level: If the error is above the desired level, a new hidden node will be added. The weights from input nodes to the enhancement node can be randomly generated, but a numerical rank check may be necessary to ensure that the added input node will increase the rank of augmented matrix by one. At this time the pseudoinverse of the new matrix can be updated by using Equation (5.4).

Step 4. Stepwise update the weight matrix: After entering a new input pattern to the input matrix, (i.e., adding \( a' \) to \( A_n \) and forming \( A_{n+1} \)), the new weight matrix \( W_{n+1} \) can be obtained or updated, using either Equation (5.3) or the Q-R decomposition algorithm. Then testing data is applied again to check the error level.

Step 5. Looping for further training: Repeat by going to Step 3 until the desired error level is achieved.

It is worth noting that having more enhancement nodes does not necessarily mean better performance. Particularly, a larger than necessary number of enhancement nodes usually would make the augmented input matrix very ill-conditioned and therefore prone to computational error. Theoretically, the rank of the expanded input matrix will be increased by one, which is not the case as observed in practice. Suppose the expanded input matrix has singular value decomposition \( A = U \sum V' \), where \( U, V \) are orthogonal matrices, and \( \sum \) a diagonal matrix whose diagonal entries give the singular values of \( A \) in ascending order. Let the condition number of \( A \) be the ratio of the largest singular value over the smallest one. If the small singular values are not rounded off to zeros, the conditional number would be huge. In other words, the matrix would be extremely ill-conditioned. The least-square solution resulting from the pseudoinverse would be very sensitive to small perturbations which are not desirable. A possible solution would be to cut off any small singular values (and therefore reduce the rank). If the error is not under the desired level after training, extra input nodes will be produced based on the original input nodes and the enhanced input nodes, where the weights are fixed. This is similar to the idea of ‘cascade-correlation’ network structure [27]. But one step learning is utilized here, which is much more efficient.
5.7 Examples and Discussion

The proposed time-series forecasting model is tested on several time-series data including an infrared laser data set, a chaotic time-series, a monthly flour price data set, and a non-linear system identification. The following examples not only show the effectiveness of the proposed method but also demonstrate a relatively fast way of forecasting time-series. The non-linear system identification of discrete-time single-input, single-output (SISO), multiple-input, multiple-output (MIMO) plants can be described by the difference equations [16]. The most common equation for system identification is

\[ y_p(k + 1) = f[y_p(k), \ldots, y_p(k - n + 1)] + g[u(k), \ldots, u(k - m + 1)], \]

where \([u(k), y_p(k)]\) represents the input-output pair of the plant at time \(k\) and \(f\) and \(g\) are differentiable functions.

The system identification model extends the input dimension, that is the addition of the state variables. The training concept is similar to the one-dimensional (i.e., time) time-series prediction. The proposed algorithm can be also applied to multi-lag, MIMO systems easily as shown in Example 4.

**Example 1** This is one of the data sets used in a competition of time-series prediction held in 1992 [28]. The training data set contains 1000 points of the fluctuations in a far-infrared laser as shown in Figure 4. The goal is to predict the continuation of the time-series beyond the sample data. During the course of the competition, the physical background of the data set was withheld to avoid biasing the final prediction results. Therefore we are not going to use any information other than the time-series itself to build our network model. To determine the size of the network, first we use simple linear net as a preliminary fit, i.e., AR(\(p\)), where \(p\) is the value of so-called lag. After comparing the single step error versus the value of \(p\), it’s noted that optimal choice for the lag value lies between 10 to 15. So we use an AR(15) model and add nonlinear enhancement nodes as needed. Training starts with a simple linear network with 15 inputs and 1 output node. Enhancement nodes are added one at a time and weights are updated using Equation (5.4), as described in Section 3. After about 80 enhancement nodes are added, the network can perform single step predictions exceptionally well. Since the goal is to predict multiple steps beyond the training data, iterated prediction is also produced. Figure 4 shows 60 steps iterated prediction into the future, as is compared to the actual continuation of the time-series. The whole procedure including training and producing predictions took just about less than 20 seconds on a DEC alpha machine, compared to the huge computation with over 1000 parameters to adapt and overnight training time using back-propagation training algorithm. To compare the prediction with previous work [28], the normalized mean squared error (NMSE) is defined as

\[ \frac{1}{\hat{\sigma}^2} \frac{1}{N} \sum_{k \in \tau} (y_k - \hat{y}_k)^2, \]
where \( k = 1, 2, \ldots, N \) denotes the points in the test set \( \tau \), \( \hat{\sigma}_{\tau}^2 \) denotes the sample variance of the observed value in \( \tau \), \( y_k \) and \( \hat{y}_k \) are target and predicted values, respectively. A network with 25 lags and 50 enhancement nodes are used for predicting 50 steps and 100 steps ahead using 1000 data points for training. For 50 steps ahead prediction, the NMSE is about \( 4.15 \times 10^{-4} \), and the NMSE for 100 steps ahead prediction is about \( 8.1 \times 10^{-4} \). The results are better than those previously done, shown in Table 1, in both speed (such as hours, days, or weeks) and accuracy [28] (See Table 2 of reference [28], page 64).

**Table 1**  Previous Results for Example 1

<table>
<thead>
<tr>
<th>methods</th>
<th>type</th>
<th>computer</th>
<th>time</th>
<th>NMSE(100)</th>
<th>-log(lik.)</th>
</tr>
</thead>
<tbody>
<tr>
<td>conn</td>
<td>1-12-12-1; lag25,5,5</td>
<td>SPARC 2</td>
<td>12 hrs</td>
<td>0.028</td>
<td>3.5</td>
</tr>
<tr>
<td>loc lin</td>
<td>low-pass embd, 8dim 4nn</td>
<td>DEC3100</td>
<td>20 min</td>
<td>0.080</td>
<td>4.8</td>
</tr>
<tr>
<td>conn</td>
<td>feedforward, 200-100-1</td>
<td>CRAY Y-MP</td>
<td>3 hrs</td>
<td>0.77</td>
<td>5.5</td>
</tr>
<tr>
<td>conn</td>
<td>feedforward, 50-20-1</td>
<td>SPARC 1</td>
<td>3 weeks</td>
<td>1.0</td>
<td>6.1</td>
</tr>
<tr>
<td>visual</td>
<td>look for similar stretches</td>
<td>SR Iris</td>
<td>10 sec</td>
<td>1.5</td>
<td>6.2</td>
</tr>
<tr>
<td>visual</td>
<td>look for similar stretches</td>
<td>SR Iris</td>
<td>-</td>
<td>0.45</td>
<td>6.2</td>
</tr>
<tr>
<td>conn</td>
<td>feedforward, 50-350-50-50</td>
<td>386 PC</td>
<td>5 days</td>
<td>0.38</td>
<td>6.4</td>
</tr>
<tr>
<td>conn</td>
<td>recurrent, 4-4c-1</td>
<td>VAX 8530</td>
<td>1 hr</td>
<td>1.4</td>
<td>7.2</td>
</tr>
<tr>
<td>tree</td>
<td>k-d tree, AIC</td>
<td>VAX 6420</td>
<td>20 min</td>
<td>0.62</td>
<td>7.3</td>
</tr>
<tr>
<td>loc lin</td>
<td>21dim 30nn</td>
<td>SPARC 2</td>
<td>1 min</td>
<td>0.71</td>
<td>10</td>
</tr>
<tr>
<td>loc lin</td>
<td>3dim time delay</td>
<td>Sun</td>
<td>10 min</td>
<td>1.3</td>
<td>-</td>
</tr>
<tr>
<td>conn</td>
<td>feedforward</td>
<td>SPARC 2</td>
<td>20 hrs</td>
<td>1.5</td>
<td>-</td>
</tr>
<tr>
<td>conn</td>
<td>feedforward, weight-decay</td>
<td>SPARC 1</td>
<td>30 min</td>
<td>1.5</td>
<td>-</td>
</tr>
<tr>
<td>lin</td>
<td>Wiener filter, width 100</td>
<td>MIPS 3230</td>
<td>30 min</td>
<td>1.9</td>
<td>-</td>
</tr>
</tbody>
</table>

**Example 2**  Time series produced by iterating the logist map

\[
f(x) = \alpha x(1 - x), \text{ or } x(n + 1) = \alpha x(n)(1 - x(n))\]

is probably the simplest system capable of displaying deterministic chaos. This first-order difference equation, also known as the Feigenbaum equation, has been extensively studied as a model of biological populations with non-overlapping generations, where \( x(n) \) represent the normalized population of \( n \)-th generation and \( \alpha \) is a parameter that determines the dynamics of the population. The behavior of the time-series depends critically on the value of the bifurcation parameter \( \alpha \). If \( \alpha < 1 \), the map has a single fixed point and the output or population dies away to zero. For \( 1 < \alpha < 3 \), the fixed point at zero becomes unstable and a new stable fixed point appears. So the output converges to a single nonzero value. As the value of \( \alpha \) increases beyond 3, the output begins to oscillate first between two values, then four values, then eight values and so on, until \( \alpha \) reaches a value of about 3.56 when the output becomes chaotic. The \( \alpha \) is set to 4 for producing the tested time-series data from the above map. The logistic map of the time-series equation (the solid curve)
Figure 4(a) Prediction of the time-series 60 steps of the future

Figure 4(b) Network prediction (first 50 points) (left), 4(c) Network prediction (first 100 points)
and the output predicted by the neural network (the ‘x’ curve) is shown in Figure 5(a). A short segment of the time-series is shown in Figure 5(b). The network is trained to predict the \((n+1)\)-th value based only on the value at \(n\). The training set consists of 100 consecutive pairs of \((x_t, x_{t+1})\) time-series values. With just five enhancement nodes, the network can do a single step prediction pretty well after training. To produce multiple steps ahead prediction, ten enhancement nodes can push the iterated prediction up to 20 steps into the future with a reasonable error level (see Figure 5(c)).

**Example 3** As the third example, we tested the model on a trivariate time-series \(X_t=\{x_t, y_t, z_t, t = 1, 2, \cdots, T\}\), where \(T\) ranges up to 100. The data used are logarithms of the indices of monthly flour prices for Buffalo \((x_t)\), Kansas City \((y_t)\) and Minneapolis \((z_t)\) over the period from 8/72 to 11/80 [29]. First we trained the network with 8 enhancement nodes using first 90 data. The next 10 data sets are then tested in one-lag prediction, starting from \(t = 91\). To compare the prediction with previous work, the mean squared error (MSE) is defined as \(\frac{1}{N} \sum_{k \in \tau} (y_k - \hat{y}_k)^2\), where \(k = 1, 2, \cdots, N\) denotes the points in the test set \(\tau\), \(y_k\) and predicted values, respectively. Figure 6(a) shows the flour price indices. Figure 6(b) shows the networking modeling and target output. The prediction and the error are given in Figure 6(c) and Figure 6(d) respectively. The training MSEs for Minneapolis, Kansas City, and Buffalo are 0.0039, 0.0043, and 0.0051, respectively.
prediction MSEs for Minneapolis, Kansas City, and Buffalo are are 0.0053, 0.0055, and 0.0054, respectively. The result is better than previous work using multi-layer network. We also trained the network with six inputs coupled (combined) with 10 enhancement nodes using the first 90 triplets from the data. The network performs well even in multi-lag prediction, or iterated prediction. This is shown in

We also observe that, even though more lags or more enhancement nodes would achieve better fit during the training stage, they do not necessary improve prediction performance, especially in the case of multi-lag.

Example 4  The model is also used for a MIMO nonlinear system. The two-dimensional input-output vectors of the plant were assumed to be $u(k) = [u_1(k), u_2(k)]'$ and $y(k) = [y_{p1}(k), y_{p2}(k)]'$. The difference equation describing the plant was assumed to be of the form [16],

$$
\begin{bmatrix}
    y_{p1}(k+1) \\
    y_{p2}(k+1)
\end{bmatrix} = \begin{bmatrix}
    f_1[y_{p1}(k), y_{p2}(k), u_1(k), u_2(k)] \\
    f_2[y_{p1}(k), y_{p2}(k), u_1(k), u_2(k)]
\end{bmatrix}$$

where the known functions $f_1$ and $f_2$ have the forms:

$$
f_1(y_{p1}, y_{p2}, u_1, u_2) = \frac{0.8y_{p1}^3 + u_1^2u_2}{2 + y_{p2}^2},$$

and

$$
f_2(y_{p1}, y_{p2}, u_1, u_2) = \frac{y_{p1} - y_{p1}y_{p2} + (u_1 - 0.5)(u_2 + 0.8)}{1 + y_{p2}^2}.$$

© 2001 by CRC Press LLC
Figure 6(c) Network prediction (one-lag) (left), 6(d) Network prediction error (one-lag)

Figure 6(e) Iterated prediction (multi-lag) of flour price indices of three cities, where solid lines are the predicted values and the dashed lines are the actual values
The stepwise update algorithm with 5 enhancement nodes is used to train the above system. Using $u_1(k) = \sin(2\pi k/250)$ and $u_2(k) = \cos(2\pi k/250)$, the responses are shown in Figure 7. Figure 7(a) is the plot for $y_{p1}$ and $\hat{y}_{p1}$ and Figure 7(b) is the plot for $y_{p2}$ and $\hat{y}_{p2}$. The dashed-line and solid-line are also overlapped in this case. Figure 7(c) and (d) show the plots of $y_{p1} - \hat{y}_{p1}$ and $y_{p2} - \hat{y}_{p2}$, respectively. The training time is again very fast – about 30 seconds in a DEC workstation.

5.8 Conclusions

In summary, the algorithm described in this chapter is simple and fast and easy to update. Several examples show the promising result. There are two points that we want to emphasize: (1) The learning algorithm for functional-link network is very fast and efficient. The fast learning makes it possible for the trial-error approach to fine-tune some hard-to-determine parameters (e.g., the number of enhancement
(hidden) nodes), and the dimension of the state space, or the AR parameter $p$. The training algorithm allows us to update the weight matrix in real-time if additional enhancement nodes are added to the system. Meanwhile, the weights can also be updated easily if new observations are added to the system. This column-wise (additional neurons) and row-wise (additional observations) update scheme is very attractive to real-time processes. (2) The easy updating of the weights in the proposed approach saves time and resources to re-train the network from scratch. This is especially beneficial when the data set is huge.
References


