CHAPTER II

ARTIFICIAL NEURAL NETWORK

2.1 Artificial Neural Network versus Human Brain

Artificial neural network (ANN) is a mathematical algorithm for information processing and knowledge acquisition through the learning processes. In the design of network calculations, there is a massive interconnection of simple computing cell called neuron which will be used to imitate the human brain's capability. General descriptions of artificial neural networks can be found in various existing literatures (Hagan *et al.*, 1996; Haykin, 1994). This section briefly describes ANNs in comparison with the human brain as follows.

Historically, an artificial neural network (ANN) is a mathematical model or computational model based on biological neural networks. ANN can be considered as parallel distributed processors made up of simple processing units. Its functionality is to store experimental knowledge and to make it available for later use according to Haykin (1994). ANN resembles the human brain in two aspects:

1. Knowledge is required by the network from its environmental through learning processes.

2. Inter-neuron connection strength defined by weight value is used to store the acquired knowledge.

Human brain is basically composed of a specific type of cells, known as biological neurons. These cells provide us the abilities to remember, think, and apply our past experience to make decisions or solve problems. The brain is activated as a result of these biological neurons and the connections between them. Figure 2.1 depicts the two interconnected biological neurons. Each biological neuron comprises of four basic components namely dendrite, soma (cell body), axon, and synapses. Input and output

signals to the soma of a biological neuron are transmitted along the axon and dendrite, while the synaptic resistance controls the strength of the signal. These neurons naturally learn to produce a particular signal by adjusting the synaptic resistance. Neurons that are connected to the others constitute an enormous network called a neural network. The synaptic resistance is formally referred as the weight of neuron.

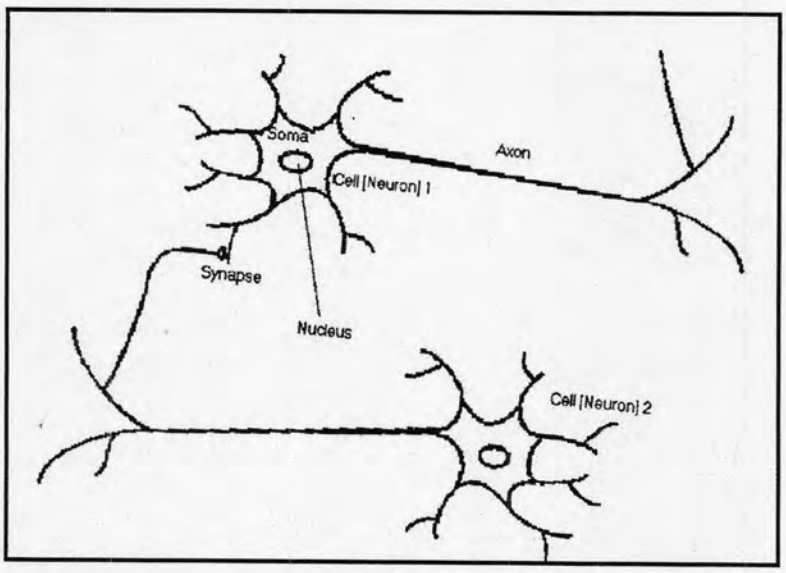

Figure 2.1: Two interconnected biological neurons.

ANN resembles the biological neuron in a way that each component forming an artificial neuron has a consistent functionality in comparison with biological neuron as follows: soma is consistent with a neuron node, axon is consistent with output, dendrite is consistent with an input, and synapse is consistent with a weight.

2.2 Classification of Artificial Neural Networks

For simplicity, we shall hereafter refer to "artificial neural network" as "network" or "ANN" and "artificial neuron" as "neuron". The general procedure of learning process or learning rule in a network is equivalent to adjusting weights of the network. Networks can be classified into three broad categories (Hagan *et al.*, 1996), based on the learning types, as

1. Supervised learning: The learning rule or weights adjustment is referred to the set of input-target pairs called training set applied to the network. The learning objective is to produce the output as close to the target of the same input as possible by using weight adjustment.

2. Unsupervised learning: The learning without a teacher does not require a target associated with each input pattern in the training data set. It explores the underlying structure, or correlations between patterns in the data, and organizes patterns into finite categories from these correlations.

3. Reinforcement learning: It is similar to supervised learning except that, instead of being provided with the target for each network input, the output of the network is assigned to grades or scores. Grade or score is the measure of the correctness of network outputs over a sequence of inputs.

The artificial neural networks with proposed learning rule have been widely used for solving seven classes of the challenging problems in sciences and engineering (Jain *et al.*, 1996) namely: (i) pattern classification such as character and speech recognitions, (ii) clustering or categorization including data mining, data compression and exploratory data analysis, (iii) prediction and forecasting, (iv) function approximations, (v) optimization, (vi) content-addressable memory or associative memory, which is extremely desirable in building multimedia information databases, and (vii) system control, such as engine idle-speed control. For applications on prediction problems, function approximations, pattern classification, data compression and control, the supervised learning neural network gains more popularity among researchers.

In this thesis, the backpropagation algorithm, which is one of the important classes in supervised learning category, is used to model the rainfall-runoff relationship in the designated study area.

2.3 A Neuron Model

A neuron model comprises of five basic elements, as shown in Figure 2.2. These elements are mathematically described as input vector, weight values, summer, activation function and output of neuron.

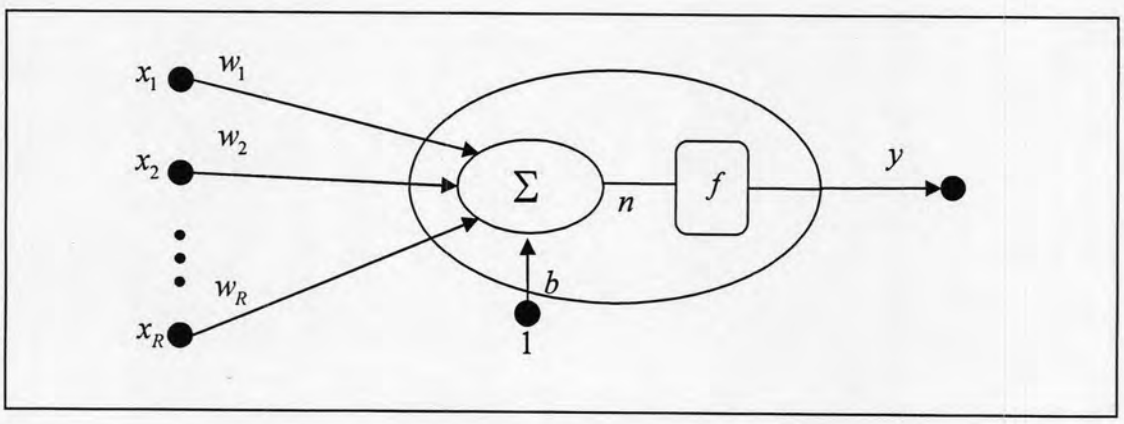

Figure 2.2: A neuron model.

An input vector $\vec{x}$ of a neuron is characterized by $x_1, x_2, ..., x_R$. Generally, this input can be derived from the input data or the output signals produced from other neurons. Weights $w_1, w_2, ..., w_R$ and a bias b are adjustable parameters. Bias b can be considered as weight of which the input is 1. A summer $\sum$ is an operator for generating net input n for an activation function f. The net input n is a weighted sum of all input fed into the neuron. The activation function f could be either linear or nonlinear function. A neuron produces only one output y, which can be written as

$y = f(n)$, where $n = \sum_{i=0}^{R} w_i x_i$ for $i = 0$; $w_0 = b$ and $x_0 = 1$.

Generally, the output of a neuron is defined in terms of the net input n. Three basic types of the activation functions are

1. threshold function or heaviside function defined by

$$f(n) = \begin{cases} 1 & if \quad n \geq 0 \\ 0 & if \quad n < 0 \end{cases}$$, as shown in Figure 2.3

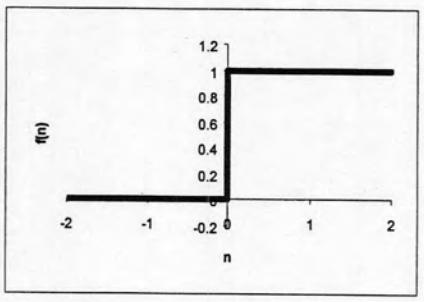

Figure 2.3: Threshold function.

2. piecewise-linear function defined by

$$f(n) = \begin{cases} 1 & , \quad n \geq 1 \\ n & , \quad 1 > n > 0 \\ 0 & , \quad n \leq 0 \end{cases}$$, as shown in Figure 2.4

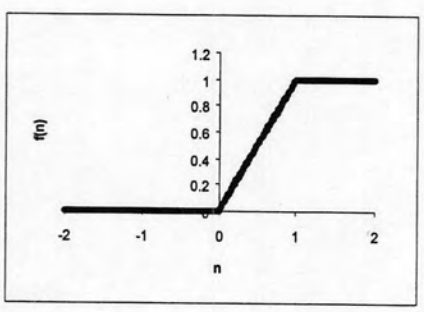

Figure 2.4: Piecewise-linear function.

3. logistic sigmoid function defined by

$$f(n) = \frac{1}{1+e^{-n}}$$, as shown in Figure 2.5

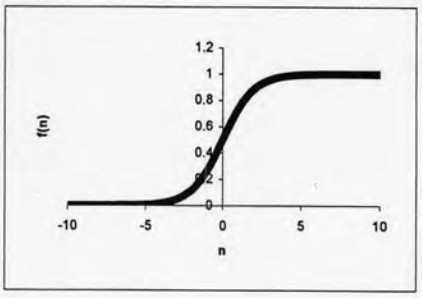

Figure 2.5: Logistic sigmoid function.

2.4 Multilayer Perceptrons

For supervised learning model, to get the better convergence for the network, the learning parameters (weights and biases) need to be adjusted based on the comparison of the output and the target. One of the well-known neural networks for this learning objective is the multilayer feedforward neural networks, which are commonly referred as the multilayer perceptrons. The feedforward network structure is typically consists of three layers: input layer, hidden layer and output layer.

Input layer is the first layer consisting of neurons, called source nodes, derived from elements of the input pattern. The source nodes constitute the input signal for the neurons in the second layer.

Hidden layer is the second layer containing the computational nodes designed for signal from the input layer. In this layer, the weights and biases are adjusted by the calculation of which the output is passed on to the layer below it. The next layer can be another hidden layer or the output layer.

Output layer is the last layer of the network. It consists of computational nodes that are responsible for producing the network output.

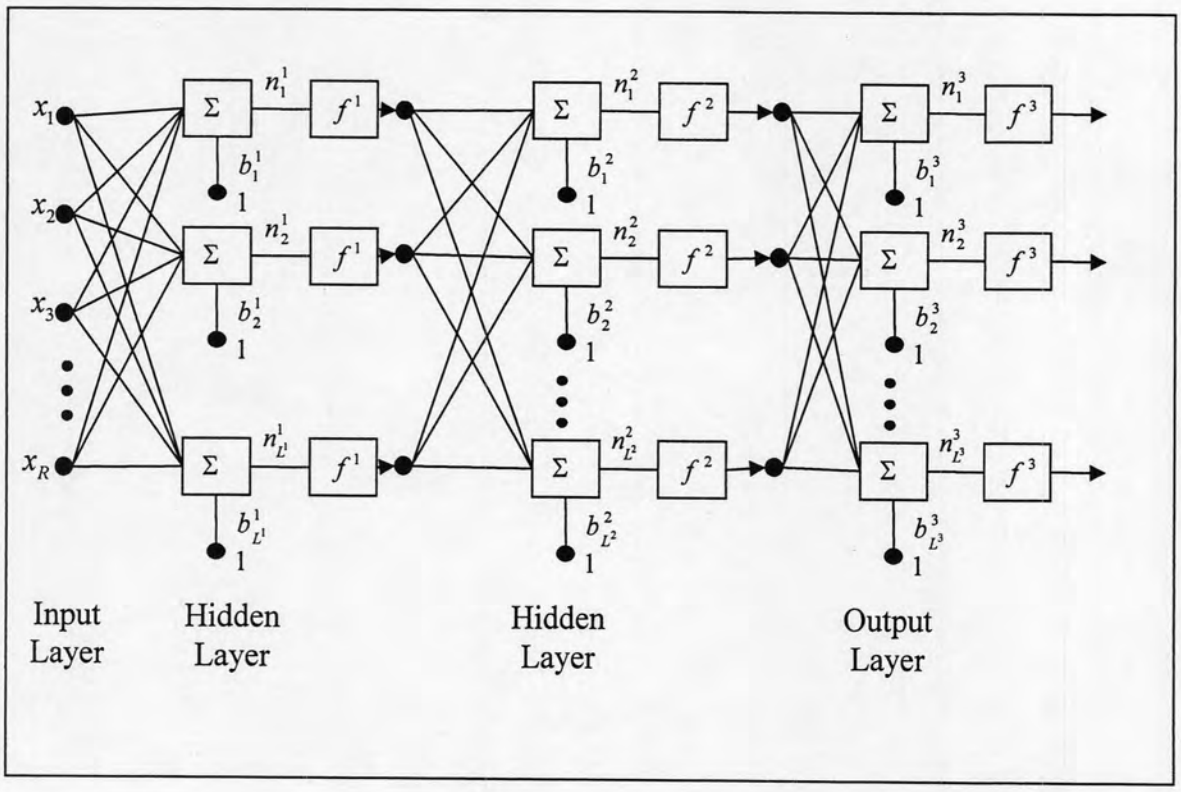

Figure 2.6: Structure of multilayer perceptrons.

When every neuron in each layer is connected to every other neuron in the next layer, the network is called a fully connected network (see Figure 2.6). If, however, some of these connections are missing, the network is said to be partially connected.

2.5 Learning Process

In the learning process of artificial neural network, there are learning rules based on weight and bias adjustment. The adjustment process concerns the minimization of mean square error of the outputs of the network and the corresponding targets.

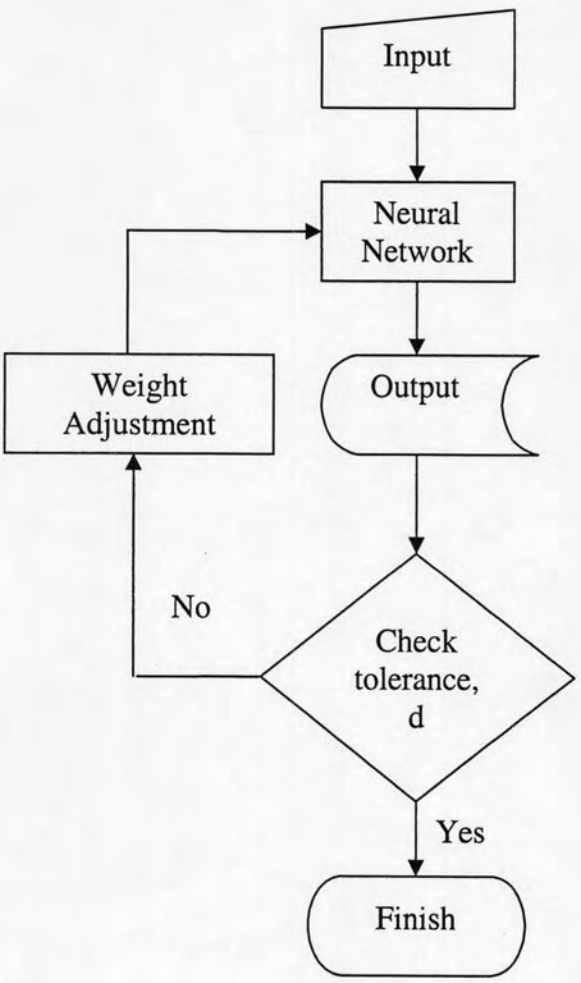

Figure 2.7: Diagram of weight adjustment process.

Figure 2.7 shows the paradigm of weight adjustment process in the back-propagation algorithm. Firstly, the initial weights of the neural network are determined for generating the output after the input is fed into the network. Then, the network

determines the difference between the outputs and the corresponding targets by adopting statistical criterion (mean square error). If the difference or error between them is more than a tolerance *d*, the network will adjust the weights of computational nodes in iterative manner. The name of backpropagation is derived from the direction of weights adjustment, which proceeds backward from the output layer to the first hidden layer in the network. The process continues iteratively until error of the network output and the corresponding target is less than the tolerance *d*. This implies that learning is achieved. Finally, this set of weights from the converged network is tested on another data set for validation.

The weights adjustment process is associated with two kinds of signals: function and error signals. Their characteristics can be described as follows.

1. Function signal is an input signal generated from source nodes or output of neurons in hidden layers. This signal propagates forward through the network and appears finally as outputs of the network.

2. Error signal originates at an output neuron of the network and propagates backward through the network.

To improve the learning process, the backpropagation algorithms are adopted in the weight adjustment process (Furundzic, 1998; Maier and Dandy, 2000; Rajurkar *et al.*, 2004; Sajikumar and Thandaveswara, 1999; Warnerm and Misra, 1996). There is a variation of back propagation algorithms. The standard one is a gradient descent algorithm. Unfortunately, this algorithm is practically slow due to the impact of learning rate (Maier and Dandy, 2000). If the learning rate is too small, so is the training. In addition, the network is likely to be trapped in the neighborhood of local minimum on the error surface. On the other hand, when the learning rate is too large, the network can fall into oscillatory traps. It is difficult to find a learning rate that can balance the high learning speeds and the minimization of the risk of divergence. Therefore, faster algorithms have been proposed and developed to avoid this difficulty for the past few years. Some of these faster algorithms can be classified by the two main approaches: first-order methods and second-order methods. First-order methods use heuristic

techniques, which were developed from the analysis of the performance of the standard steepest descent algorithm such as resilient backpropagation algorithm. Second-order methods use standard numerical optimization techniques based on a quadratic model such as conjugate gradient, Quasi-Newton, and Levenberg-Marquardt algorithms. In both cases, iterative techniques are used to minimize the error function.

In this thesis, the Levenberg-Marquardt algorithm is used in the weight adjustment process for the network training. This algorithm is one of the most efficient learning algorithms for neural network. The main advantage of the Levenberg-Marquardt is its speed of convergence (Hagan and Menhaj, 1994; Maier and Dandy, 2000; Singh *et al.*, 2006).

2.6 Levenberg-Marquardt Algorithm

The Levenberg-Marquardt (LM) algorithm (Hagan and Menhaj, 1994; Hagan *et al.*, 1996; Singh *et al.*, 2006) is originally based on the minimization problem of a performance function $F(\vec{\mathrm{x}})$, which is the sum of square functions. It is derived from Newton's method for the estimation of Hessian matrix, which is the second derivatives of the performance function, in terms of the Jacobian matrix.

The Newton's method for optimizing a performance function $F(\vec{\mathrm{x}})$ is described as follows:

$$\Delta\vec{\mathrm{x}}_k = -\mathrm{H}_k^{-1}\vec{\mathrm{g}}_k, \tag{2.1}$$

where H_k^{-1} is the inverse of Hessian matrix (H_k) at the k^{th} iteration. The Hessian matrix (H_k) characterized by $\mathrm{H}_k \cong \nabla^2 F(\vec{\mathrm{x}}_k)$, and $\vec{\mathrm{g}}_k$ is gradient at the k^{th} iteration given by $\vec{\mathrm{g}}_k \cong \nabla F(\vec{\mathrm{x}}_k)$.

The sum of square functions $F(\vec{\mathrm{x}})$ can be written as

$$F(\vec{\mathrm{x}}) = \frac{1}{2}\sum_{i=1}^{N} v_i^2(\vec{\mathrm{x}}) = \frac{1}{2}\vec{\mathrm{v}}^{\mathrm{T}}\vec{\mathrm{v}}.$$

Here $\vec{x} = [x_1\ x_2 \ldots x_n]^T$ and $\vec{v} = [v_1\ v_2\ \ldots\ v_N]^T$. Here $(\,)^T$ is the transpose of vector or matrix.

The j^{th} element of the gradient can be expressed as

$$\frac{\partial F(\vec{x})}{\partial x_j} = \sum_{i=1}^{N} v_i(\mathrm{w}) \frac{\partial v_i}{\partial x_j}.$$

Therefore, the gradient can be written in matrix form as

$$\vec{g}(\vec{x}) = \nabla F(\vec{x}) = J^T(\vec{x})\vec{v}(\vec{x}), \tag{2.2}$$

where the Jacobian matrix is

$$J(\vec{x}) = \begin{bmatrix} \frac{\partial v_1}{\partial x_1} & \frac{\partial v_1}{\partial x_2} & \cdots & \frac{\partial v_1}{\partial x_n} \\ \frac{\partial v_2}{\partial x_1} & \frac{\partial v_2}{\partial x_2} & \cdots & \frac{\partial v_2}{\partial x_n} \\ \vdots & \vdots & \ddots & \vdots \\ \frac{\partial v_N}{\partial x_1} & \frac{\partial v_N}{\partial x_2} & \cdots & \frac{\partial v_N}{\partial x_n} \end{bmatrix}.$$

The k,j^{th} element of the Hessian matrix is given by

$$H_{kj} = [\nabla^2 F(\vec{x})]_{kj} = \sum_{i=1}^{N} [v_i(\vec{x}) \frac{\partial^2 v_i(\vec{x})}{\partial x_k \partial x_j} + \frac{\partial v_i(\vec{x})}{\partial x_k} \frac{\partial v_i(\vec{x})}{\partial x_j}].$$

Final form of the Hessian matrix can then be rewritten in matrix notation as

$$H(\vec{x}) = \nabla^2 F(\vec{x}) = J(\vec{x})^T J(\vec{x}) + S(\vec{x}), \quad \text{where } S(\vec{x}) = \sum_{i=1}^{N} v_i(\vec{x}) \frac{\partial^2 v_i(\vec{x})}{\partial x_k \partial x_j}.$$

For the Gauss-Newton method, it is assumed that $S(\vec{x}) \approx 0$. The Hessian matrix can then be approximated by

$$H(\vec{x}) = \nabla^2 F(\vec{x}) = J(\vec{x})^T J(\vec{x}). \tag{2.3}$$

From (2.1) - (2.3), we obtain

$$\Delta\vec{x}_k = -[J^T(\vec{x}_k)J(\vec{x}_k)]^{-1} J^T(\vec{x}_k)\vec{v}(\vec{x}_k). \tag{2.4}$$

One serious difficulty of the Gauss-Newton method occurs when the Hessian matrix is not invertible at some iteration. To resolve this, the approximate Hessian matrix has to be slightly modified by

$$G = H + \mu I \quad \text{, where } \mu \text{ is a positive constant.} \tag{2.5}$$

The question here is to show that the matrix G can be made invertible. Let the eigenvalues and eigenvectors of H be $\{\lambda_1, \lambda_2, ..., \lambda_s\}$ and $\{z_1, z_2, ..., z_s\}$, respectively. Then

$$Gz_i = [H + \mu I]z_i = (\lambda_i + \mu)z_i \quad \text{, where } \mu > 0 \text{ and I is the identity matrix.}$$

The eigenvalues of G are $\lambda_i + \mu$. It should be noted that both matrices G and H posses the same eigenvectors. The matrix G can be made positive definite by increasing μ until $\lambda_i + \mu > 0$ for all i and hence the matrix is invertible. Following (2.4), the Levenberg-Marquardt algorithm can be expressed as

$$\Delta\vec{x}_k = -[J^T(\vec{x}_k)J(\vec{x}_k) + \mu_k I]^{-1} J^T(\vec{x}_k)\vec{v}(\vec{x}_k). \tag{2.6}$$

As μ_k increases, the learning rate decreases. This leads to the steepest descent algorithm:

$$\vec{x}_{k+1} = \vec{x}_k - \frac{1}{\mu_k} J^T(\vec{x}_k)\vec{v}(\vec{x}_k) = \vec{x}_k - \frac{1}{2\mu_k}\nabla F(\vec{x}_k) \text{ , for large } \mu_k. \tag{2.7}$$

On the other hand, the algorithm reduces to the Gauss-Newton method as μ_k decreases to zero.

The algorithm begins by setting small value of μ_k (i.e. $\mu_k = 0.01$). For the k^{th} iteration, if $F(\vec{x})$ obtained from computation does not decrease, then μ_k is multiplied by a constant $\delta > 1$. Since a small weight adjustment in (2.7) is taken in the direction of steepest descent, $F(\vec{x})$ should eventually decrease. Once $F(\vec{x})$ decreases, the μ_k is divided by δ in the next iteration. This algorithm is basically the Gauss-Newton, which can provide faster convergence (Singh *et al.*, 2006).