CHAPTER 3

MODELING USING NEURAL NETWORKS (NN)

Aspects of NN modeling, architecture and modes of training are described in this section. Multi-layer Perceptrons (MLP) is discussed in great depth while Single-layer Perceptron is presented as an introduction to the MLP.

3.1 NN Architecture

Many types of modeling task that can be handled by NN are related to, among others, different NN architectures developed by researchers in this field. Table 3-1 lists the types of modeling tasks [47] .

Table 3-1 Types of problem modeled and solved using NN

Type of problem	Description	Examples
Pattern classification	Assigning an input pattern or symbol represented by a feature vector to one of many pre-specified classes	Character and speech recognition, blood cell classification
Clustering	Exploring the similarity between patterns and placing or categorizing similar patterns in a cluster (also known as unsupervised pattern classification)	Data mining, data compression
Function approximation	Suppose a set of n labeled training patterns or input-output pairs of $\{(\mathbf{x}_1,y_1), (\mathbf{x}_2,y_2), \ldots, (\mathbf{x}_n,y_n)\}$ have been generated from an unknown function $y(\mathbf{x})$ (subject to noise). Function approximation is finding an estimate of the unknown function, say $\hat{y}$.	Nonlinear function that is too complex to derived analytically,various engineering and scientific modeling problems
Prediction/forecasting	Predicting a quantity in the future event or in the different condition	Stock market prediction, weather forecasting, failure prediction
Optimization	Finding a solution satisfying a set of constraints such that an objective function is maximized or minimized	The TSP, a wide variety of problems in mathematics, statistics, engineering, science, medicine and economics

Table 3-1 Types of problem modeled and solved using NN

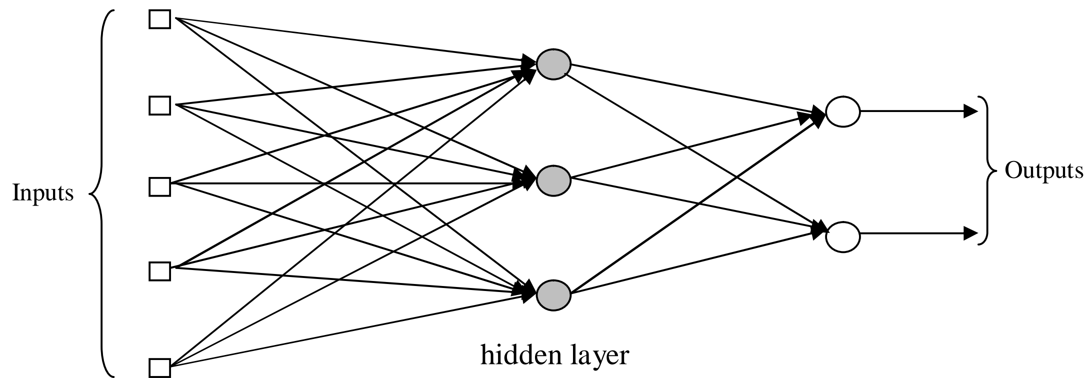
Type of problem	Description	Examples
Content-addressable memory	Accessing an entry in memory not only from its address, but also from its content. The content in the memory can also be recalled by a partial input or distorted content (also known as associative memory)	Used in multimedia information database
Control	Consider a dynamic system defined by control input $u(t)$ and resulting output $y(t)$. The goal is to generate a control input $u(t)$ such that the system follows a desired trajectory determined by a reference model	Engine idle-speed control, kinematic robot control

In earlier works on NN, the pioneers focused on anthropomorphic arguments to introduce and develop the architecture models and related methods for NN learning (cognitive science perspective). Nowadays, the attention is more focused on the algorithms and computations of NN to be applied efficiently to solve many practical problems in the engineering and industrial fields, without too much question on how a brain might work (engineering perspective) [48].

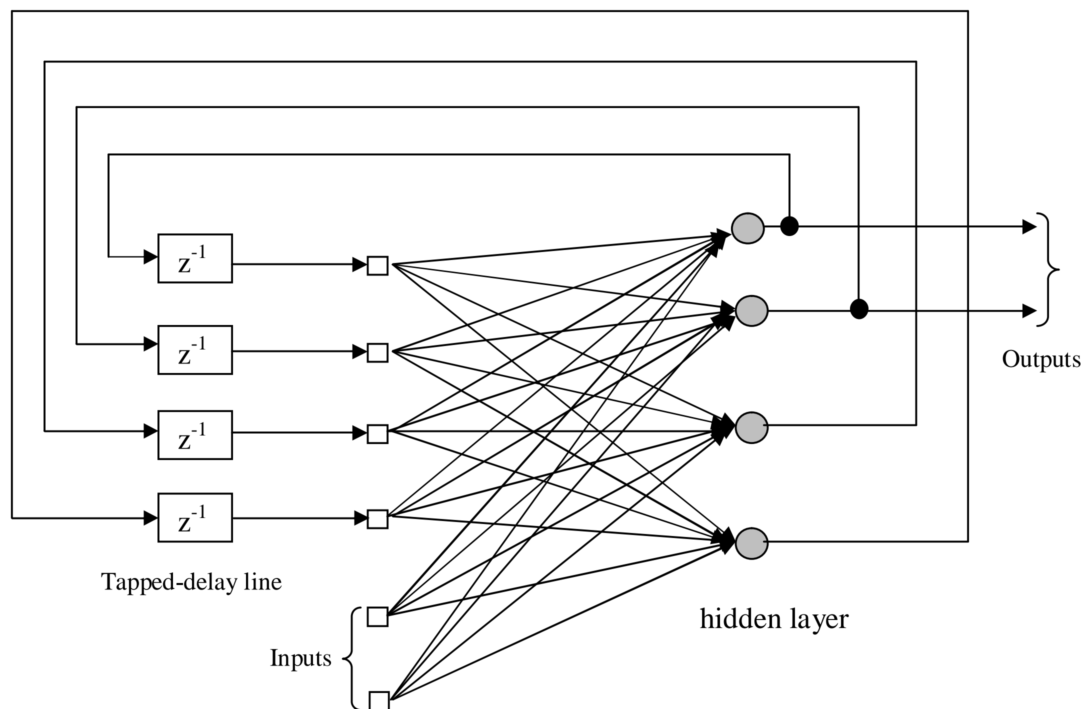
Based on the perspective of connecting patterns, NN can be grouped into two general categories: feed forward and recurrent networks. Feed forward networks have no loops because of the absence of feed back connections between the layers. In this sense, a set of inputs is carried forward and processed through the layers to produce one set of outputs, and hence carry no memories. Therefore, the feed forward networks are static because for a given input vector, the networks always generate the same output vector. In other words, their response to an input vector is independent of the previous state.

On the other hand, recurrent or feed back neural networks are dynamic systems. With the presence of feed back connections or also known as tapped-delay line, loops occur and the networks are able to carry memories and retain information to be used later. When a new input pattern is presented, the neuron outputs are computed. However, because of the feed back connections, the inputs of the neurons are then modified and the networks lead to enter a new state. Therefore, the networks can generate different ouput vectors for a given input vector. In other words, their response to an input vector is dependent on the previous network state and the actual network state [49].

Figure 3-1 shows typical architectures for feed forward and recurrent neural networks.



(a)



(b)

Figure 3-1 NN architectures (a) feed forward with one hidden layer (gray colored circles), and (b) recurrent network with hidden neurons.

Based on the above characteristics, feed forward networks are suitable for processing of patterns with remarkable spatial dependencies such as in the area of pattern recognition and function approximation [50]. On the other hand, feed back neural networks, due to their inherent capability in memorizing the past information, are suitable for processes with spatio-temporal dependencies such as system identification and control of non-linear dynamic systems [51, 52], time series prediction [53] and prediction of other non-linear and non-stationary signals, for examples air pollutants [49], machine fault diagnostic [54]. Further description and discussion on the neural network models along with their variations and considerations can be found in [53, 55, 56].

3.2 NN Modes of Training

There are two modes of NN training: incremental and batch. In incremental mode of training, the weights are updated after each input is presented. This training mode seems to be a natural choice to be used for recurrent or feedback neural networks, because input pattern occurs within a time interval due to the presence of the feed back loops. Nevertheless, incremental mode of training can be applied to both feed forward and feed back neural networks. Another name for this mode of training is online, sequential or example-by-example training.

Another mode of training is batch or concurrent training. In this training mode, weights are only updated after all input patterns are presented into the networks. In other words, weights are only updated once in each epoch or iteration. This mode of training is therefore more suitable for feed forward neural networks that are static networks or when example patterns are not explicit functions of time. Similar to incremental mode of training, batch training can also be applied to both feed forward and feed back neural networks.

It is clear that in incremental mode of training, the sequence of example patterns is important, because weights are updated differently for different example pattern sequence (one epoch for one example pattern). Meanwhile, the sequence of example patterns is not important in batch mode of training, because weights are only updated after all example patterns are complete.

For what follows, training of NN is used in the context of supervised learning, that is example patterns of data or input-output pairs are provided. Generally, a set of the examples is called a training or learning set, through which the NN learning process occurs. Neural networks then extract the information contained in the example patterns and map the input-output relationships. For the current study, only batch mode of training will be used.

3.3 Problem Formulation of NN Learning

Problem formulation of NN employed throughout this study is now concisely presented. The statement starts from basic principle and then some limiting factors related to NN learning are described.

Let (P,T) be a pair of random variables with values in $P = \Re^m$ and $T = \Re$, respectively. The regression of T on P is a function of P, $f: P \rightarrow T$, giving the mean value of T conditioned on P, $E(T|P)$.

Let random samples $O_1^Q = \{(P_1, T_1), \ldots, (P_Q, T_Q)\}$ of size Q can be drawn from the distribution of (P,T) as an observation set. For $Q \geq 1$, $\hat{f}_Q$ will denote an estimator of f based on the random samples, that is a map $\hat{f}_Q: O_1^Q \rightarrow \hat{f}_Q(O_1^Q, .)$, where for fixed O_1^Q, $p \rightarrow \hat{f}_Q(O_1^Q, p)$ is an estimate of the regression function $f(p)$.

Furthermore, for convenience P and T will be referred to as the sets of input and variable output, respectively. Given the observation set O, learning in NN for realization of the estimate $\hat{f}$ means adjusting to vector of parameters weight $\mathbf{w}$ and biases $\mathbf{b}$ using a set of learning rule or learning algorithm in such a way that $\hat{f}$ minimizes the objective function or empirical error defined as:

$$E(\mathbf{w}) = \sum_{q=1}^{Q} \left[t_q - \hat{f}(\mathbf{p}_q; \mathbf{w}) \right]^2 \qquad (3\text{-}1)$$

and generalizes well or outputs properly when a novel input $\mathbf{p}_{\text{test}}$ never seen before is feeded into the network.

In other words, through the learning process the weights (including biases) will be adjusted adaptively through proper training algorithm with respect to the given examples and enforced to specific values so that the perceptron network performs the modeling task correctly or closer to the desired target.

By keeping in mind that training in NN is principally updating the network weights based on the given set of examples so that the network will give proper response to new examples, below are two limiting factors of NN learning.

First, only a finite number of observation points (example pairs) are available. This means that the available examples sometimes must be fully utilized for the NN learning purpose to provide proper learning of the underlying process. Hence, the practicability and feasibility of using limited examples for NN learning to yield accurate prediction output are assured.

Secondly, the realization of target at the points of observation p_q, $q = 1, \ldots, Q$, is observed with an additive noise e_q:

$$e_q = T_q - f(P_q) \tag{3-2}$$

The observations are then noisy and the target noises e_q introduce a random component in the estimation error.

3.3.1 Generalization

Related to the limiting factors, the most important task in NN learning is for the NN model to capture regularity in the observations rather than learn the noise so that the NN model gives proper response to the novel example. If it is the case, NN is said as generalizing well. Otherwise, NN is said to learn very well by giving the correct response to the data in learning set, but predicting very poorly with respect to the new

data in testing set. This condition is well-known as overfitting that result in serious implication in critical application or safety-related areas [57].

Regularization is one way to achieve good generalization. Through regularization, over parameterization of the estimating model with respect to the number of training data is avoided with which overfitting is less likely to occur. Regularization will be further discussed when MLP is elaborated in section 3.5.

3.4 Introduction to Multi-layer Perceptrons (MLP): Single-layer Perceptron

The simplest neural network model is the perceptron. The perceptron model is based on the McCulloch-Pitts neuron model with the activation function of unit step function or hardlimiter. When more neurons with hardlimiter activation function are utilized, a single-layer perceptron will be found.

Recall the McCulloch-Pitts neuron model in Figure 1-2. Now, the neuron model is redrawn as in Figure 3-2. The new symbols introduced in Figure 3-2 will be used throughout the remaining of this thesis.

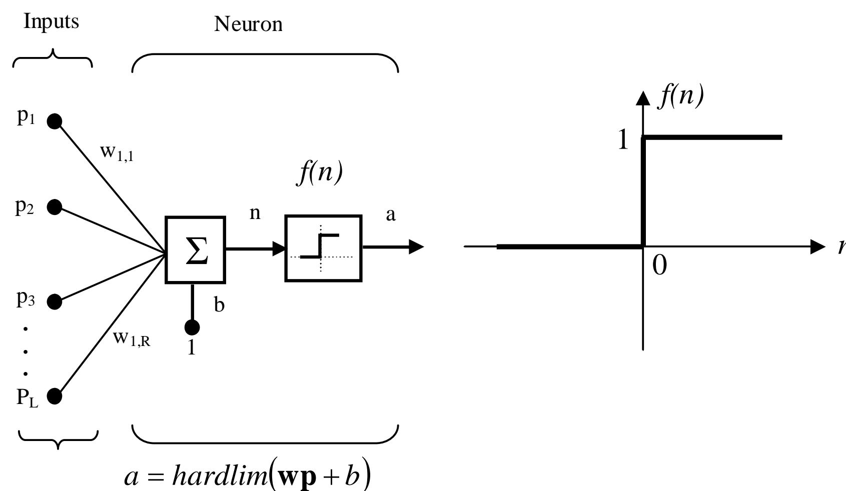


Figure 3-2 Perceptron neuron with an L-element input vector and hardlimiter function.

The complete symbols are as follows:

p_1, p_2, ... , $p_L \equiv$ the individual element inputs.

$L \equiv$ total number of elements in the input vector.

$w_{1,1}$, $w_{1,2}$, … , $w_{1,L}$ ≡ weights

b ≡ bias

w ≡ the (single row) matrix of weights

p ≡ the vector of inputs

hardlim = the hardlimiter activation function

n ≡ the sum of the weighted inputs and bias, $n = \mathbf{wp} + b$

a ≡ the output of the activation function f, that is $a = f(\mathbf{wp} + b)$

$\boldsymbol{\Sigma}$ ≡ the summing junction

Using the hardlimiter activation function, the neuron produces a 1 if the net input n into the transfer function is equal to or greater than 0, otherwise it produces a 0. (There are many other kinds of the activation function f suitable for NN, such as pure linear, logistic function or tangent hyperbolic. See the discussion on MLP).

$$f(n) = \begin{cases} 1 & if \quad n \geq 0 \\ 0 & if \quad n < 0 \end{cases} \tag{3-3}$$

In the perceptron neuron, a set of input vector **p** that is multiplied by weights **w**, and bias b are summed up together at the summing junction $\boldsymbol{\Sigma}$. The summation n is passed through the activation function of hard-limiter to produce the output a. Therefore, it is clear that in a neuron, two processes are represented, namely summing up the weighted inputs **wp** and bias b, and passing the summation n through the activation function f.

For simplicity, the perceptron neuron model can also be represented as in Figure 3-3.

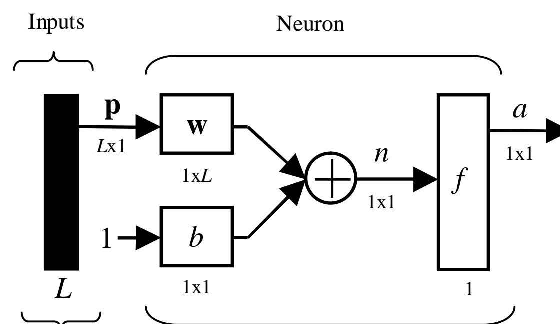


Figure 3-3 The simplified schematic of a neuron model.

Through the simplified schematic, the input vector **p** is represented by a solid dark vertical bar at the left. Also, the dimensions of the input vector, weight and bias matrix, the summation n, the activation function f and the output a are all shown clearly.

The one-neuron perceptron can simply be extended to a single-layer perceptron (SLP) by adding more neurons in the neuron layer. Note that with s number of neurons in the perceptron layer, the dimension of biases **b** and weights **w** now become sx1 and sxR, respectively. The output of the perceptron becomes a vector **a** with dimension sx1. The same with the summation vector **n**. Figure 3-4 shows a schematic and its simplified schematic of a single-layer perceptron.

Because the weight connections involved now become more complicated, it is necessary to indicate the strength of the connection from the jth input to the ith neuron as $w_{i,j}$, where $j = 1, 2, \ldots, L$ and $i = 1, 2, \ldots, s$.

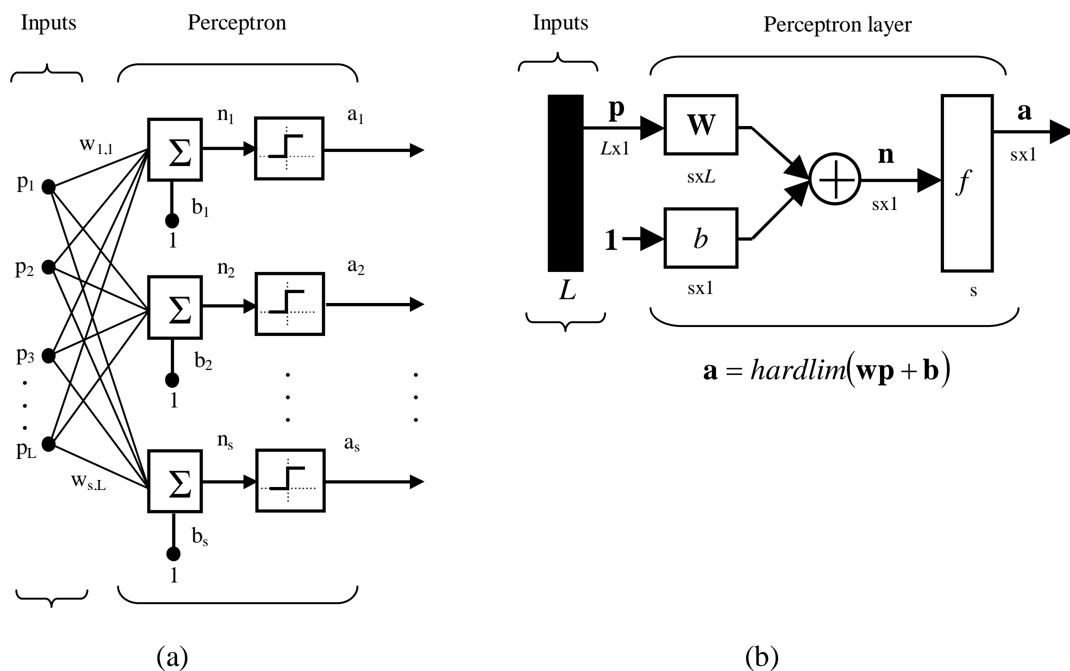


Figure 3-4 A single-layer perceptron (a) its schematic (b) its simplified schematic.

Referring to Figure 3-4 and the notation, the summation vector **n** can now be expressed as follows:

$$\mathbf{n}_{[s\mathbf{x}1]} = \begin{bmatrix} n_1 \\ n_2 \\ \cdot \\ \cdot \\ n_s \end{bmatrix} = \begin{bmatrix} w_{1,1}p_1 + w_{1,2}p_2 \ldots w_{1,R}p_R + b_1 \\ w_{2,1}p_1 + w_{2,2}p_2 \ldots w_{2,R}p_R + b_2 \\ \cdot \\ \cdot \\ w_{s,1}p_1 + w_{s,2}p_2 \ldots w_{s,R}p_R + b_s \end{bmatrix} \quad (3\text{-}4)$$

By feeding the summation vector **n** into the hardlimiter activation function, the output vector **a** results. Depending on the net input **n**, the hardlimiter will give an output either of 1 or 0.

3.4.1 Capability of Single-layer Perceptron

Having detailed discussion on the structure of the one-neuron perceptron and the single-layer perceptron, it seems naturally to think about their capabilities and what can be hoped by adding more neurons in a single-layer perceptron. In other words, what can a single-layer perceptron solve?

The answer of the question is the perceptron can categorize *linearly separable* pattern classification problem. The perceptron can be used to classify input vectors that can be separated by a linear boundary (hyperplane). If a straight line or a plane can be drawn to separate the input vectors into their correct categories, the input vectors are said to be *linearly separable*. When a line or a plane can separate the input vectors correctly, two classes of input vector are created. If two lines are needed in order to categorize the input vectors correctly, four classes of input vector are created, and so forth. It can be summarized that a single-layer perceptron with S neurons can classify or categorize 2^S classes. For example, if a single-layer perceptron is needed to categorize 8 classes of input vector, at least the single-layer perceptron must have 3 neurons. This is further illustrated in Figure 3-5.

A simple justification for the above statement is that for the S neurons there will be S number of components in the output vector **a** making a number of 2^S possibilities as the target vectors. For example, the associated target vectors when using 2 neurons are $[0\ 0]^T$, $[0\ 1]^T$, $[1\ 0]^T$ and $[1\ 1]^T$, thus 4 classes of input patterns can be classified.

Moreover, it is important to note that the boundary lines (decision boundaries) are mathematically represented as **wp** + **b** = 0. The biases **b** indicate how far the boundary lines are shifted from the origin. This is easy to see because the lines separate between the region of the net input **n** ≥ 0 and the region of the net input **n** < 0. Here, **p** represents all points on the boundary and depending on the number of inputs in the network, the equation will represent a line or a plane. For all the boundary lines passing through the point of origin such as in Figure 3-5, it can be easily determined that the bias values are 0. Note also that the lines in Figure 3-5 are only a few examples from many other possible separating lines that can be assigned to solving the problems.

For a three-input vector **p**, the boundaries become planes instead of lines as shown in Figure 3-6. For more input vectors, however, the graphical display is not as convenient.

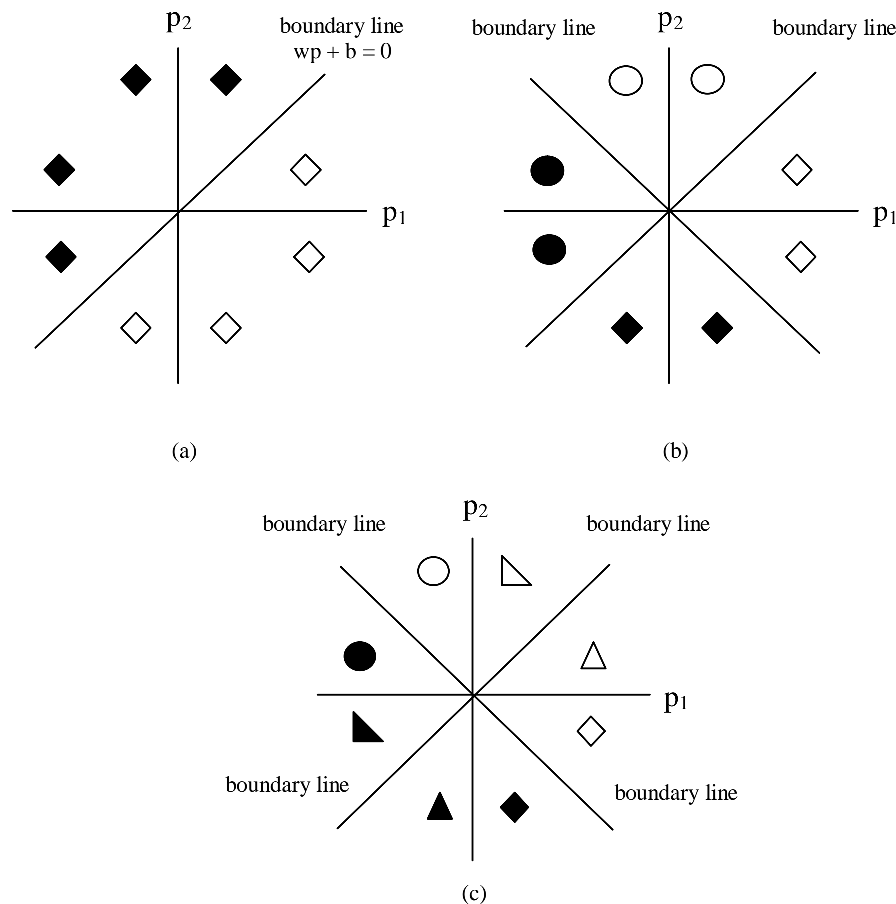


Figure 3-5 Input vectors of (a) two classes (b) four classes, and (c) eight classes.

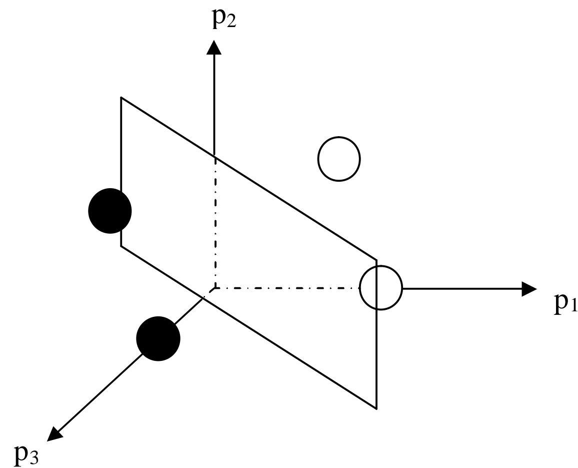


Figure 3-6 A representation of the boundary plane in a three-input space.

3.4.2 Perceptron Learning Rule

The single-layer perceptron applies the method of supervised learning, that is a number of input-target pairs that represents the behaviour of the modeling task under investigation must be provided and supplied into the network. The training set of input-target pairs are usually symbolized as follows:

$$\{\mathbf{p}_1, \mathbf{t}_1\}, \{\mathbf{p}_2, \mathbf{t}_2\}, \ldots, \{\mathbf{p}_Q, \mathbf{t}_Q\}$$

where $\mathbf{p}_q$ is an input vector to the network, $\mathbf{t}_q$ is the corresponding (associated) target output and Q is the total number of training patterns (samples).

Within the framework of supervised learning, one important thing for updating the weights in the single-layer perceptron is the error signal **e**. It is the difference between the desired target **t** and the output of the perceptron **a**, as stated in Equation (3-5).

$$\mathbf{e} = \mathbf{t} - \mathbf{a} \tag{3-5}$$

The error signal will also be very useful when discussing backpropagation training algorithm for MLP. Using the error signal multiplied by the input vector **p**, the initial weights of single-layer perceptron are moved on the suitable values so that the network

classifies input patterns correctly as long as the solution exists. The rule for updating the weights (the perceptron learning rule) then can be written as:

$$\mathbf{w}^{\text{new}} = \mathbf{w}^{\text{old}} + \mathbf{e}\,\mathbf{p}^T \tag{3-6}$$

Similarly for biases, by considering the constant inputs of one for the biases.

$$\mathbf{b}^{\text{new}} = \mathbf{b}^{\text{old}} + \mathbf{e} \tag{3-7}$$

Compiling the weights and biases as **w**, and the input vector and the inputs of one as **p**, the learning rule can be written as a single equation as in Equation (3-8):

$$\mathbf{w}^{\text{new}} = \mathbf{w}^{\text{old}} + \mathbf{e}\,\mathbf{p}^T \tag{3-8}$$

Effectively, the increment (step) of weights in single-layer perceptron training can be written as:

$$\Delta\mathbf{w} = \mathbf{e}\,\mathbf{p}^T \tag{3-9}$$

Remembering that the associated target $\mathbf{t}_q$ is either 0 or 1, the error values e can be either 0, 1 or -1. Furthermore, with respect to the error values, the perceptron learning rule can be summarized as follows:

The first, if $\mathbf{e} = 0$, then $\Delta\mathbf{w} = 0$ (there is no weight update).
The second, if $\mathbf{e} = 1$, then $\Delta\mathbf{w} = \mathbf{p}^T$.
The third, if $\mathbf{e} = -1$, then $\Delta\mathbf{w} = -\mathbf{p}^T$.

It can be summarized that learning in the single-layer perceptron for each training iteration is simply adding the previous weight values by either the incoming input vector **p** or its negative, until the classification task could be completed.

The single-layer perceptron works with incremental mode of training, because the weights are updated after each input is presented where the vectors are presented sequentially into the network. It can be proven that as long as the solution exists, training for single-layer perceptron will complete in finite number of iterations [7].

3.5 Multi-layer Perceptrons (MLP)

With the discussion of single-layer perceptron, the extension to MLP is relatively straightforward. Nevertheless, the most interesting feature of MLP is their increased capability compared to single-layer perceptron. The increased capability makes MLP suitable for many complex and non-linear modeling problems. Characteristics of MLP structure, activation function and learning algorithms play important roles to the suitability of MLP.

3.5.1 MLP Structure

The subsequent discussion will be focused on MLP with one layer of hidden neurons as shown in Figure 3-7. For MLP with only one output neuron, its structure can be translated into the one as shown in Figure 3-8. In the figures, superscripts 1 and 2 refer to hidden and output layer, respectively. Without loss of generality, subsequent discussion will be on MLP with one hidden layer and one output neuron.

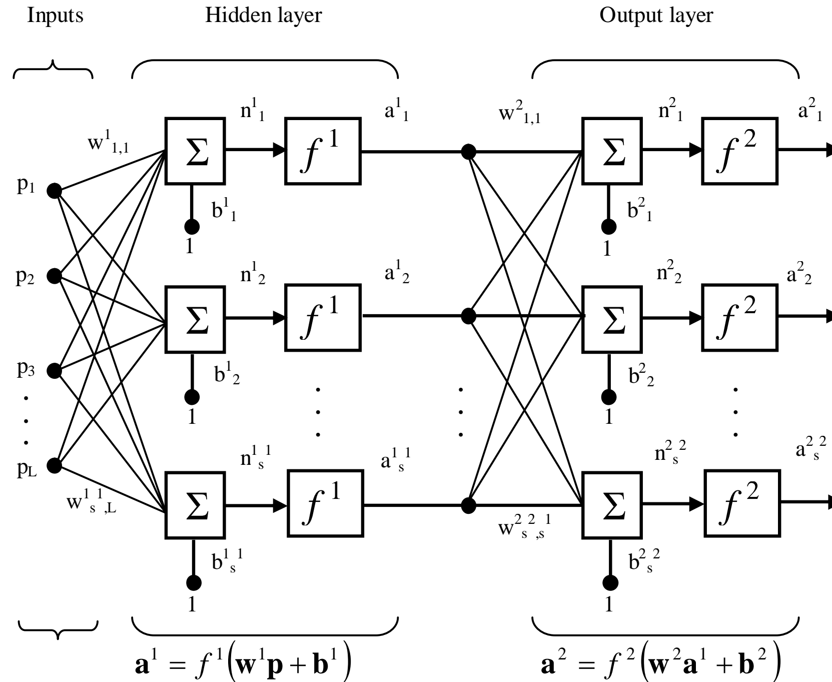


Figure 3-7 MLP with one hidden layer and multiple output.

As pure linear function in the output layer returns an output which is the same as the input, the estimate $\hat{f}$ realized by the MLP given the training set O can be written as:

$$\hat{f}(\mathbf{p};\mathbf{w}) = \sum_{i=1}^{s} \mathbf{w}^{2}_{1,i}\ \tau\left(\mathbf{w}^{1}_{i,j}\,\mathbf{p} + \mathbf{b}_{i}\right) + \mathbf{b}_{o} \tag{3-10}$$

where $\tau(.)$ is a sigmoidal function, which will be described in the following section.

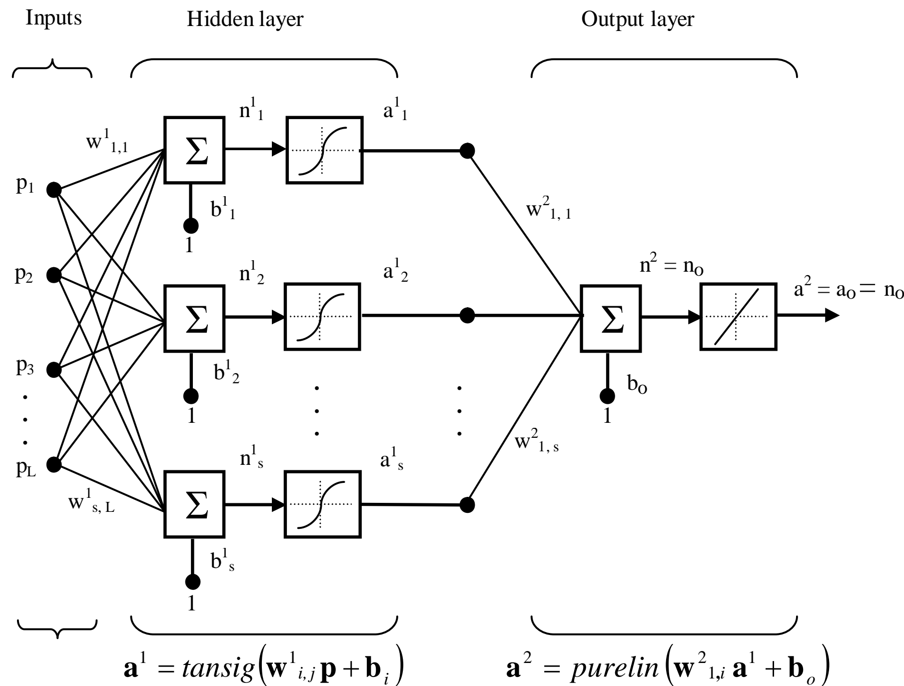


Figure 3-8 MLP with one hidden layer and single output.

3.5.2 Activation Function

Besides hardlimiter and linear functions, other types of activation function include sigmoidal-based activation functions, namely hyperbolic tangent, binary sigmoid (logistic) and bipolar sigmoid. The sigmoidal functions are commonly used for the hidden nodes.

The sigmoidal functions are chosen because of their characteristics. First, the functions are continuous, differentiable and monotonically non-decreasing. Second, the

corresponding derivatives are easy to compute and the value of the derivative at a particular value of the independent variable can be expressed in terms of the value of the function itself. Third, by using the sigmoidal functions in hidden units, a small change in the weights will usually produce a change in the outputs, thus indicates whether the weight change is good or bad.

Various types of sigmoidal function together with the corresponding graphical representation are shown in Figure 3-9, 3-10 and 3-11. For hyperbolic tangent function with output range within the interval [-1, 1], the activation function and the corresponding $f'(n)$ are given in Equation (3-11) and (3-12), respectively.

$$f(n)=\left(\frac{e^{n}-e^{-n}}{e^{n}+e^{-n}}\right) \quad (3\text{-}11)$$

$$f'(n)=(1+f(n))(1-f(n)) \quad (3\text{-}12)$$

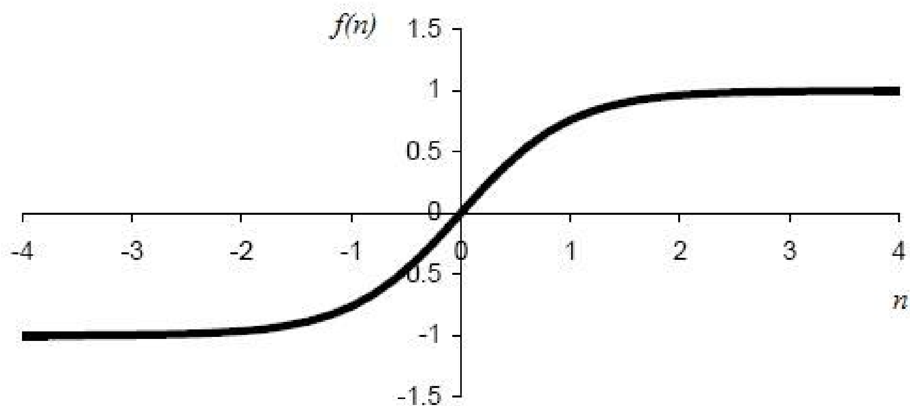


Figure 3-9 Hyperbolic tangent activation function.

For binary sigmoid function (logistic function), Equation (3-13) and (3-14) apply:

$$f(n)=\frac{1}{(1+e^{-n})} \quad (3\text{-}13)$$

$$f'(n)=f(n)(1-f(n) \quad (3\text{-}14)$$

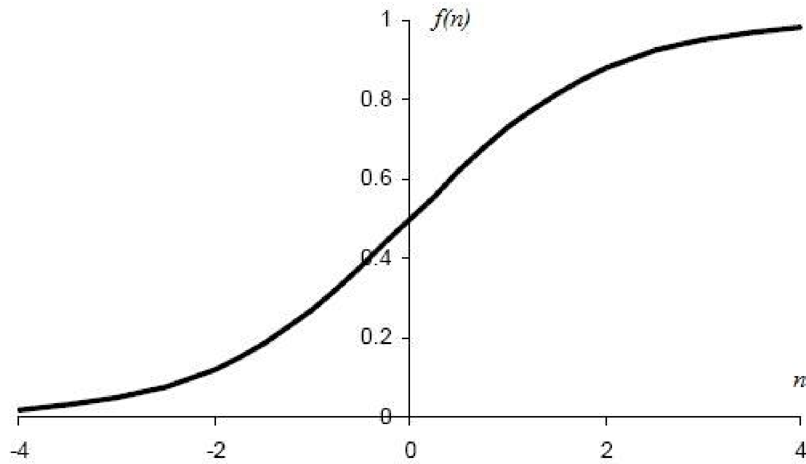


Figure 3-10 Logistic activation function.

For bipolar sigmoid function, Equation (3-15) and (3-16) apply:

$$f(n)=\left(\frac{1-e^{-n}}{1+e^{-n}}\right) \tag{3-15}$$

$$f^{'}(n)=\frac{1}{2}(1+f(n))(1-f(n)) \tag{3-16}$$

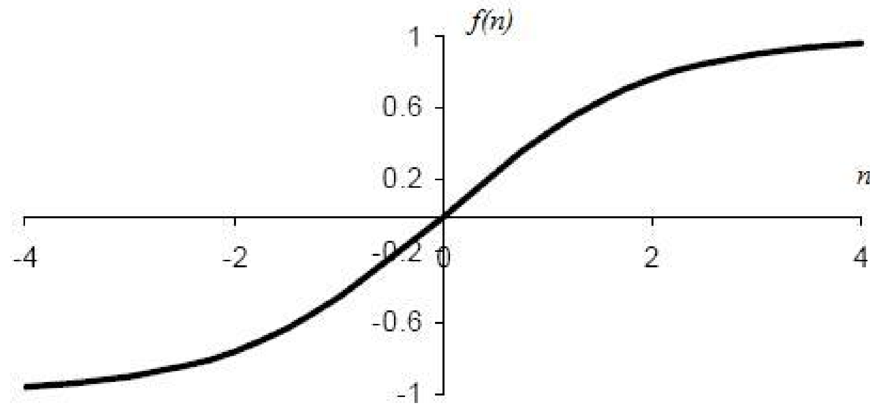


Figure 3-11 Bipolar sigmoid activation function.

3.5.3 Training Algorithm

The most widely used procedure in NN learning is backpropagation, which is a breakthrough in the resurgence of interest in NN study [58]. The term backpropagation refers to the manner in which error information from output layer is backpropagated through the layers within the network. The process is repetead consecutively immediately after the input samples are propagated forward through the network. For each propagating-backpropagating pass, the weights of the network are updated iteratively. The updating process is repeated until predefined stopping criterion is met. The stopping criteria could be in form of performance goal measured by mean square error (MSE), maximum iteration number, minimum performance gradient and minimum change in performance.

3.5.3.1 Gradient Descent

In principle, NN learning task is a minimization problem to particular objective function (recall Equation (3-1)) and the minimization problem is related to the method for updating the NN weights. A basic method employed for updating NN weights is gradient descent, which is described as follows.

The function to be minimized is of the following special form given by Equation (3-17):

$$f(\mathbf{x}) = \frac{1}{2}\sum_{j=1}^{m} r_j(\mathbf{x})^2 \tag{3-17}$$

where $\mathbf{x} = x_i = (x_1, x_2, ..., x_n)$ is a vector where $1 \le i \le n$ and each of r_j is a function from $\Re^n$ to $\Re$, where $1 \le j \le m$. r_j is referred to as a *residual*.

Residual vector **r** is a vector function of **x**: $\Re^n$ to $\Re^m$, hence f can be rewritten as:

$$f(\mathbf{x}) = \frac{1}{2}\|\mathbf{r}\|^2 \tag{3-18}$$

The Jacobian matrix **J** of r with respect to **x**, can be defined as:

$$\mathbf{J} = \frac{\partial r_j(\mathbf{x})}{\partial \mathbf{x}} \tag{3-19}$$

Gradient **g** and Hessian **H** are, respectively, defined as Equation (3-20) and Equation (3-21):

$$\mathbf{g} = \nabla f(\mathbf{x}) = \sum_{j=1}^{m} r_j(\mathbf{x}) \frac{\partial r_j(\mathbf{x})}{\partial \mathbf{x}} = \mathbf{J}^{\mathrm{T}}\mathbf{r} \tag{3-20}$$

$$\mathbf{H} = \nabla^2 f(\mathbf{x}) = \mathbf{J}^{\mathrm{T}}\mathbf{J} + \sum_{j=1}^{m} r_j(\mathbf{x}) \nabla^2 r_j(\mathbf{x}) \tag{3-21}$$

If it is possible to approximate the *residuals* r_j by linear functions ($\nabla^2 r_j(\mathbf{x})$ are small) or the *residuals* r_j themselves are small, the Hessian **H** can be approximated as:

$$\mathbf{H} = \nabla^2 f(\mathbf{x}) = \mathbf{J}^{\mathrm{T}}\mathbf{J} \tag{3-22}$$

Further, if vector **x** represents weights vector **w** and $f(\mathbf{x})$ represents the performance function $E(\mathbf{w})$, the basic weights update rule, which is a gradient descent rule, in NN training can be written as:

$$\mathbf{w}_{i+1} = \mathbf{w}_i - \eta \nabla E(\mathbf{w})\big|_i \tag{3-23}$$

$$\mathbf{w}_{i+1} = \mathbf{w}_i - \eta \mathbf{g}\big|_i \tag{3-24}$$

where $\mathbf{w}_{i+1}$ is weights vector at next iteration, $\mathbf{w}_i$ is weights vector at current iteration, η is learning rate factor and i is iteration step.

Thus, in NN training, the weights are moved along the negative of the performance function gradient **g** using the gradient descent learning.

3.5.3.2 Optimization Methods

Equation (3-24) can be considered as a first order approximation to gradient descent and therefore the method is often very slow in practice. To overcome the weakness of basic gradient descent, methods based on second order approximation to gradient descent have been developed, such as Newton and Levenberg-Marquardt (LM) method.

If $E(\mathbf{w})$ is expanded using a Taylor series in minimizing the performance function, then the Newton's method for updating weights will be found as shown in Equation (3-25):

$$\mathbf{w}_{i+1} = \mathbf{w}_i - (\nabla^2 E(\mathbf{w}))^{-1} \nabla E(\mathbf{w})\Big|_i \tag{3-25}$$

Equation (3-25) can also be written in the form of Equation (3-26):

$$\mathbf{w}_{i+1} = \mathbf{w}_i - (\mathbf{H})^{-1} \mathbf{g}\Big|_i \tag{3-26}$$

LM method for updating weights is by adding an adjustable constant parameter λ to Equation (3-26):

$$\mathbf{w}_{i+1} = \mathbf{w}_i - (\mathbf{H} + \lambda \mathbf{I})^{-1} \mathbf{g}\Big|_i \tag{3-27}$$

where λ is the lambda parameter or the parameter of LM.

With the presence of the adjustable parameter λ, the LM algorithm combines both the advantages of the simple gradient descent that simply moves the weights to decrease the error function value and the Newton's method that has faster convergence. In practice, for moderate size problems, the LM algorithm is faster and robust. LM also outperforms the methods, because $E(\mathbf{w})$ is always reduced at each iteration of the algorithm [59]. For further reading on optimization, the readers are directed to [60, 61].

The key feature in the implementation of LM is procedure to update λ during iteration within the backpropagation framework, which will be described in the following section. The adaption and the implementation of LM are described in Materials and Methods section.

3.5.3.3 Backpropagation

The implementation of the gradient-based methods for updating NN weights within the framework of backpropagation is described in this section. Whenever appropriate, the gradient-based methods can be implemented in the backpropagation programming loop.

The backpropagation steps can be summarized as follows.

First, the weights and other useful parameters are initialized.
Second, the feed forward network (propagating input-output pairs) is executed.
Third, the error information is backpropagated and the weights are adjusted.
Last, the steps are repeated until the difference between the target and the network output values is within an acceptable range of performance goal and other stopping criteria.

Recal Figure 3-8. The summation and the output of the hidden nodes can be, respectively, written as:

$$\mathbf{n}^1 = \mathbf{w}^1{}_{i,j}\,\mathbf{p} + \mathbf{b}_i \tag{3-28}$$

$$\mathbf{a}^1 = tansig\left(\mathbf{w}^1{}_{i,j}\,\mathbf{p} + \mathbf{b}_i\right) \tag{3-29}$$

The MLP output is:

$$\mathbf{a}^o = \mathbf{a}^2 = \mathbf{n}^2 = \mathbf{w}^2{}_{1,i}\,\mathbf{a}^1 + \mathbf{b}_o \tag{3-30}$$

Recal Equation (3-1) where the realization of $\hat{f}\left(\mathbf{p}_q;\mathbf{w}\right)$ is replaced by $\mathbf{a}_o$. The error to be minimized then can be written as:

$$E(\mathbf{w}) = \sum_{q=1}^{Q}\left[t_q - a_{o\,q}\right]^{\mathbf{2}} \tag{3-31}$$

In vector notation:

$$E(\mathbf{w}) = \left[\mathbf{t} - \mathbf{a}_o\right]^{\mathrm{T}}\left[\mathbf{t} - \mathbf{a}_o\right] \tag{3-32}$$

The individual residual or error e is defined as:

$$e = t_q - a_{o\,q} \tag{3-33}$$

Now it is necessary to derive partial derivative of $E(\mathbf{w})$ with respect to the weights consisting of the output layer weights $\mathbf{w}^2{}_{1,i}$ and the hidden layer weights $\mathbf{w}^1{}_{i,j}$ to implement the gradient-based methods.

Partial derivative of $E(\mathbf{w})$ with respect to weights $\mathbf{w}^2{}_{1,i}$ can be derived as:

$$\frac{\partial E(\mathbf{w})}{\partial \mathbf{w}^2{}_{1,i}} = \frac{\partial E(\mathbf{w})}{\partial \mathbf{n}^2}\frac{\partial \mathbf{n}^2}{\partial \mathbf{w}^2{}_{1,i}} = \frac{\partial E(\mathbf{w})}{\partial \mathbf{n}^2}\frac{\partial(\mathbf{w}^2{}_{1,i}\,\mathbf{a}^1 + \mathbf{b}_o)}{\partial \mathbf{w}^2{}_{1,i}} = \frac{\partial E(\mathbf{w})}{\partial \mathbf{n}^2}\mathbf{a}^1 \tag{3-34}$$

The term $\frac{\partial E(\mathbf{w})}{\partial \mathbf{n}^2}$ is now derived as:

$$\frac{\partial E(\mathbf{w})}{\partial \mathbf{n}^2} = \frac{\partial\left(\left[\mathbf{t}-\mathbf{a}_o\right]^{\mathrm{T}}\left[\mathbf{t}-\mathbf{a}_o\right]\right)}{\partial \mathbf{n}^2} = \frac{\partial\left(\sum_{q=1}^{Q}\left[t_q - a_{o\,q}\right]^{\mathbf{2}}\right)}{\partial \mathbf{n}^2} = -2(\mathbf{t}-\mathbf{a}_o)\frac{\partial \mathbf{a}_o}{\partial \mathbf{n}^2} = -2(\mathbf{t}-\mathbf{a}_o) \tag{3-35}$$

Therefore, partial derivative of $E(\mathbf{w})$ with respect to the weights $\mathbf{w}^2{}_{1,i}$ is :

$$\frac{\partial E(\mathbf{w})}{\partial \mathbf{w}^2{}_{1,i}} = -2(\mathbf{t}-\mathbf{a}_o)\mathbf{a}^1 \tag{3-36}$$

In similar way, partial derivative of E with respect to the weights $\boldsymbol{w}^1{}_{i,j}$ can be derived as:

$$\frac{\partial E(\mathbf{w})}{\partial \mathbf{w}^1{}_{i,j}} = \frac{\partial E(\mathbf{w})}{\partial \mathbf{n}^1}\frac{\partial \mathbf{n}^1}{\partial \mathbf{w}^1{}_{i,j}} = \frac{\partial E(\mathbf{w})}{\partial \mathbf{n}^1}\mathbf{p} \tag{3-37}$$

The sensitivity term $\frac{\partial E(\mathbf{w})}{\partial \mathbf{n}^1}$ is derived as:

$$\frac{\partial E(\mathbf{w})}{\partial \mathbf{n}^1} = \frac{\partial E(\mathbf{w})}{\partial \mathbf{n}^2}\frac{\partial \mathbf{n}^2}{\partial \mathbf{n}^1} = \frac{\partial E(\mathbf{w})}{\partial \mathbf{n}^2}\frac{\partial(\mathbf{w}^2{}_{1,i}\,\mathbf{a}^1 + \mathbf{b}_o)}{\partial \mathbf{n}^1} = \frac{\partial E(\mathbf{w})}{\partial \mathbf{n}^2}\frac{\mathbf{w}^2{}_{1,i}\,\partial(\mathbf{a}^1)}{\partial \mathbf{n}^1} = \dot{\boldsymbol{\tau}}(\mathbf{n}^1)\mathbf{w}^2{}_{1,i}{}^{\mathrm{T}}\frac{\partial E(\mathbf{w})}{\partial \mathbf{n}^2} \tag{3-38}$$

Substituting term $\frac{\partial E(\mathbf{w})}{\partial \mathbf{n}^2}$ from Equation (3-35), the sensitivity term $\frac{\partial E(\mathbf{w})}{\partial \mathbf{n}^1}$ and partial derivative of $E(\mathbf{w})$ with respect to $\mathbf{w}^1{}_{i,j}$ are, respectively, defined as:

$$\frac{\partial E(\mathbf{w})}{\partial \mathbf{n}^1} = -2\dot{\boldsymbol{\tau}}(\mathbf{n}^1)\mathbf{w}^2{}_{1,i}{}^{\mathrm{T}}(\mathbf{t}-\mathbf{a}_o) \tag{3-39}$$

$$\frac{\partial E(\mathbf{w})}{\partial \mathbf{w}^1{}_{i,j}} = -2\dot{\boldsymbol{\tau}}(\mathbf{n}^1)\mathbf{w}^2{}_{1,i}{}^{\mathrm{T}}(\mathbf{t}-\mathbf{a}_o)\mathbf{p} \tag{3-40}$$

From Equation (3-36) and (3-40), it can be seen how the error information e is backpropagated from the output layer to the hidden layer after the feedforward step of inputs.

Having the expression of partial derivatives of $E(\mathbf{w})$ with respect to weights, the above backpropagation then can be casted into general form of pseudo-codes as follows:

Step 1. The weights **w** and other useful parameters are initialized.

Step 2. While stopping criteria are false, steps 2 – 10 are repeated.

-------------------------------------- Feedforward phase --

Step 3. For each training example, steps 4 – 9 are carried out.

Step 4. The summation at the hidden nodes are calculated using Equation (3-28): $\mathbf{n}^1 = \mathbf{w}^1{}_{i,j}\mathbf{p} + \mathbf{b}_i$ and the output of hidden nodes are computed using Equation (3-29): $\mathbf{a}^1 = tansig\left(\mathbf{w}^1{}_{i,j}\mathbf{p} + \mathbf{b}_i\right)$

Step 5. The summation and the output of the output node are calculated using Equation (3-30): $\mathbf{a}^o = \mathbf{a}^2 = \mathbf{n}^2 = \mathbf{w}^2{}_{1,i}\mathbf{a}^1 + \mathbf{b}_o$

------------------------------ Backpropagation of error phase ----------------------------------

Step 6. The sensitivity term at the output node is computed using Equation (3-35): $\frac{\partial E(\mathbf{w})}{\partial \mathbf{n}^2} = -2(\mathbf{t} - \mathbf{a}_o)$, and the increment of the output layer weights is updated using Equation (3-36): $\Delta \mathbf{w}^2{}_{1,i} = \frac{\partial E(\mathbf{w})}{\partial \mathbf{w}^2{}_{1,i}} = -2(\mathbf{t} - \mathbf{a}_o)\mathbf{a}^1$

Step 7. The sensitivity term at the hidden nodes is computed using Equation (3-39): $\frac{\partial E(\mathbf{w})}{\partial \mathbf{n}^1} = -2\dot{\boldsymbol{\tau}}(\mathbf{n}^1)\mathbf{w}^2{}_{1,i}{}^{\mathrm{T}}(\mathbf{t} - \mathbf{a}_o)$, and the increment of the hidden layer weights is updated using Equation (3-40): $\Delta \mathbf{w}^1{}_{i,j} = \frac{\partial E(\mathbf{w})}{\partial \mathbf{w}^1{}_{i,j}} = -2\dot{\boldsymbol{\tau}}(\mathbf{n}^1)\mathbf{w}^2{}_{1,i}{}^{\mathrm{T}}(\mathbf{t} - \mathbf{a}_o)\mathbf{p}$

---------------------------------- Updating weights phase ---------------------------------------

Step 8. The weights and biases of the output and hidden layers are updated using Equation (3-24).

Step 9. Stopping criterion is checked.

3.5.4 Bayesian Regularization

Discovering the function f or an estimate of it $\hat{f}$ from the observation data O given is basically an ill-posed problem, as the estimate can have infinite solutions, particularly if the nature of the function f is hard to be assessed.

To help in choosing one particular preferred solution from the infinitely many solutions, a priori information or knowledge of the function f is needed. For examples, the function f is assumed to be smooth, in the sense that two similar inputs will produce two similar outputs, or by assuming the additive noise of Equation (3-2) is drawn from Gaussian distribution.

In finding the estimate $\hat{f}$ using NN approximation, the problem of learning can be reformulated as: how to find the estimate $\hat{f}$ from a family of estimates $\hat{F}$ which minimize both squared error $E(\mathbf{w})$ of training data set and of testing data set, which is another statement of generalization (see section 3.3.1). Regularization is then a way to prevent overfitting, a condition where good generalization is hard to occur.

Bayesian regularization was utilized in this study to improve the quality of NN prediction. It minimizes a linear combination of squared errors and weights. The idea is to find a balance between the number of parameters and goodness of fit by penalizing large models [62].

The cost function of Equation (3-1) is then modified as follows:

$$E(\mathbf{w}) = \beta \sum_{q=1}^{Q} \left[t_q - \hat{f}(\mathbf{p}_q; \mathbf{w}) \right]^2 + \alpha \sum_{i=1}^{W} w_i^2 \tag{3-41}$$

where: α is a weight decay parameter, β is an inverse noise variance parameter and W is the total number of weights. The noise is assumed to apply to the target data t_q and its distribution is assumed to be a zero-mean Gaussian distribution.

Note that the new cost function consists of the sum of squared errors (E_D) and the sum of squared weights (E_w) terms. In addition, parameters α and β are introduced for penalizing large models. Using the modified cost function, it is clear that there is a need to reestimate the parameters accordingly. One estimation method is the Gauss-Newton approximation implemented to the Bayesian learning within the framework of the Levenberg-Marquardt algorithm. The complete derivation and formulation of the algorithm can be found in [63] and the clear explanation of its implementation are further described in Materials and Methods section.

By utilizing the form of regularization, an NN model with fewer number of parameters (weights) is preferred than the one with large number of weights, especially for small data sets or training examples to yield a smoother and stable NN response. The consistency of the NN prediction with respect to small or limited training examples of fatigue data can be seen later in the Results and Discussion section.

3.5.5 The Levenberg – Marquardt Training Algorithm

After discussing backpropagation in general, this section focuses on the training algorithm of Levenberg-Marquardt (LM) which is used as an optimization technique in adjusting the NN weights. It has rapid convergence and robust performance by combining both the advantages of the simple gradient descent and the Newton's method algorithms by the presence of the adjustable parameter λ, as described in 3.5.3 section.

In adjusting the NN weights, the algorithm updates the weights for the next iteration as described by Equation (3-27) which is rewritten here as Equation (3-42):

$$\mathbf{w}_{i+1} = \mathbf{w}_i - (\mathbf{H} + \lambda\mathbf{I})^{-1}\mathbf{g}\big|_i \tag{3-42}$$

The formula can be rewritten to give the weights increment or the step $\Delta \boldsymbol{w}$ as Equation (3-43):

$$\Delta\mathbf{w} = -(\mathbf{H} + \lambda\mathbf{I})^{-1}\mathbf{g} \tag{3-43}$$

where: **H** is the Hessian matrix, λ is the LM parameter, **I** is identity matrix, **J** is the Jacobian matrix, **r** is the residual vector and **g** is the gradient matrix. i and i+1 are the previous iteration and the current iteration, respectively.

In addition, the gradient **g** is given by Equation (3-44):

$$\mathbf{g} = \mathbf{J}^{\mathrm{T}}\mathbf{r} \tag{3-44}$$

The parameter λ is controlled by the so-called gain ratio ς or the ratio between the actual and predicted decrease in the performance function value $E(\mathbf{w})$.

$$\varsigma = \frac{E(\mathbf{w}) - E(\mathbf{w} + \Delta\mathbf{w})}{\hat{E}(\mathbf{w}) - \hat{E}(\mathbf{w} + \Delta\mathbf{w})} \tag{3-45}$$

Using truncated Taylor-series [61], the predicted decrease term is defined as:

$$\hat{E}(\mathbf{w}) - \hat{E}(\mathbf{w} + \Delta\mathbf{w}) = \Delta\mathbf{w}^{\mathrm{T}}(\lambda\Delta\mathbf{w} - \mathbf{g}) \tag{3-46}$$

3.5.6 Implementation

The Levenberg-Marquardt formulation including the controlling parameter λ was implemented in Matlab environment for the NN modeling. The implementation can be described in the following pseudo-codes:

Step 1. The weights $\mathbf{w}$ and parameter λ were initialized.

Step 2. The sum of squared errors over all inputs, $E(\mathbf{w})$ was computed.

Step 3. The Jacobian matrix $\mathbf{J}$ was computed.

Step 4. The equation $\Delta\mathbf{w} = -(\mathbf{H} + \lambda\mathbf{I})^{-1}\mathbf{g}$ was solved to obtain the step or increment of weights $\Delta\mathbf{w}$ ($\mathbf{I}$ is the identity matrix).

Step 5. The sum of squared errors $E(\mathbf{w})$ using $\mathbf{w}+\Delta\mathbf{w}$ was recomputed and σ was computed using Equation (4-5). If $\sigma > 0.75$, $\lambda_{new} = \lambda/2$. If predicted decrease was close to actual decrease, the search direction approached the Gauss-Newton search direction by increasing the step size. If $\sigma < 0.25$, $\lambda_{new} = 2\lambda$. If predicted decrease was far from the actual decrease, the search direction approached the gradient direction by decreasing the step size.

Step 6. If $E(\mathbf{w} + \Delta\mathbf{w}) < E(\mathbf{w})$, then $\mathbf{w}_{new} = \mathbf{w}+\Delta\mathbf{w}$ was accepted as a new iteration and λ from step 5 was used. The steps 2-6 were repeated until the stopping criterion is satisfied.

3.5.6.1 Adaptation of Bayesian Framework within The Levenberg-Marquardt Algorithm

In principle, adaptation of Bayesian framework means that the inclusion of updating parameters α and β within the above Levenberg-Marquardt pseudo-codes.

Recal Equation (3-41) which is rewritten here as:

$$E(\mathbf{w}) = \beta E_D + \alpha E_w \quad (3\text{-}47)$$

where:

$$E_D = \sum_{q=1}^{Q} \left[t_q - \hat{f}\left(\mathbf{p}_q ; \mathbf{w} \right) \right]^2 \tag{3-48}$$

$$E_w = \sum_{i=1}^{W} w_i^{\ 2} \tag{3-49}$$

Using the modified cost function, the gradient **g** and Hessian **H**, respectively, become:

$$\mathbf{g} = 2\beta \mathbf{J}^{\mathbf{T}} \mathbf{r} + 2\alpha \mathbf{w} \tag{3-50}$$

$$\mathbf{H} = 2\beta \mathbf{J}^{\mathbf{T}} \mathbf{J} + 2\alpha \mathbf{I} \tag{3-51}$$

Thus, the increment of weights $\Delta\mathbf{w}$ becomes:

$$\Delta \mathbf{w} = -\left[\beta \mathbf{J}^{\mathbf{T}} \mathbf{J} + \left(\lambda + \alpha \right) \mathbf{I} \right]^{-1} \mathbf{g} \tag{3-52}$$

For the purpose of updating λ, the predicted decrease term of Equation (3-46) was used, where **g** and $\Delta\mathbf{w}$ were from Equation (3-50) and (3-52), respectively.

Further, for the purpose of updating α and β, the Hessian formulation was utilized through the following equations:

$$\gamma = I - 2\alpha\, trace\left(\mathbf{H}^{-1} \right) \tag{3-53}$$

$$\alpha = \frac{\gamma}{2E_w} \tag{3-54}$$

$$\beta = \frac{Q - \gamma}{2E_D} \tag{3-55}$$

where: γ is the effective number of parameters, that is a measure of how many parameters or weights are effectively used (preferred) in the NN learning with respect to the cost function reduction, I is the total number of initial weights during initialization and Q is the number of training examples.

The adaptive formulation is now complete. It is clear that the parameters α and β play important roles to achieve the goal of balancing between the number of parameters and goodness of fit by penalizing large models, as stated in the previous section 3.5.4. The parameter β would drive the errors smaller. On the other hand, the parameter α would reduce the parameters (weights) size. Both would increase the generalization performance of the NN.

Based on the above formulas and referring to [63], the previous pseudo-codes of Levenberg-Marquardt algorithm can be restated as follows:

Step 1. The weights **w** and parameters λ, α and β were initialized. For example: λ = 0.005, $\alpha = 0$ and $\beta = 1$. The algorithm is not too sensitive to the initial choice of the parameters. In addition, the choice of $\alpha = 0$ and $\beta = 1$ means that the NN is starting from the original cost function. Recal Equation (3-47).

Step 2. One step of the Levenberg-Marquardt algorithm to minimize the objective function was taken as per Equation (3-52).

Step 3. The predicted decrease term was computed using Equation (4-6) based on the result of Step 2.

Step 4. The parameter λ was updated based on the predicted decrease term from Step 3.

Step 5. If $E(\mathbf{w} + \Delta\mathbf{w}) < E(\mathbf{w})$, then $\mathbf{w}_{new} = \mathbf{w} + \Delta\mathbf{w}$ was accepted as a new iteration and λ from step 4 was used.

Step 6. The effective number of parameter γ was computed using Equation (3-53) and the Hessian formulation of Equation (3-51) was utilized.

Step 7. The parameters α and β were updated using Equation (3-54) for α and Equation (3-55) for β.

Step 8. Steps 2-7 were repeated until the stopping criterion was satisfied or convergence was achieved.

Note that the Levenberg-Marquardt algorithm with Bayesian regularization parameters has now been described.

3.5.6.2 Realization

For the purpose of this modeling study, it is not adequate to use Graphical User Interface-Neural Networks Toolbox. On the other hand, Matlab provides suitable environment and much more flexibility for users to program their particular modeling tasks. Therefore, programming lines have been written in Matlab to implement the pseudo-codes for both updating the controlling parameter λ and the parameters α and β.

The Matlab programming lines were run on Toshiba Satellite with OS Windows Vista Basic, processor of Intel Pentium Dual-Core and RAM of 1 GB (Appendix A lists the M-file codes to train NN incorporating Bayesian Framework within The Levenberg-Marquardt Algorithm).