A CONTEXT SENSITIVE NEURAL NETWORK BY OVERLAPPED SYSTEMS

by

Atef S Mohamed

A thesis submitted to the faculty of graduate studies
Lakehead University
in partial fulfillment of the requirements for the degree of
Masters of Science in Mathematical Science

Department of Computer Science

Lakehead University

Mar 2006

Your file Votre référence
ISBN: 978-0-494-24061-8
Our file Notre référence
ISBN: 978-0-494-24061-8

Contents

List of Tables v

List of Fingures vii

Acknowledgments viii

Abstract ix

1 Introduction 1

1.1 Artificial neural network basics . . . 2

1.1.1 Basic neural network components . . . 3

1.1.2 Neural network topology . . . 4

1.1.3 Multi-layer perceptron (MLP) model . . . 5

1.1.4 Error back-propagation training of MLP . . . 6

1.2 Generalization ability of a neural network . . . 9

1.2.1 Weight decay approach . . . 13

1.2.2 Complexities in regularization and VC dimension . . . 13

1.3 Highlighting some neural network types . . . 16

1.3.1 Radial basis functions . . . 16

1.3.2 The basic SOM . . . 19

2 Context dependent neural networks **22**

2.1 Introduction to contextual features . . . 22

2.1.1 Definition of context . . . 23

2.1.2 Illustration example of the contextual features definitions . . . 25

2.1.3 Identification of context sensitive features . . . 27

2.2 Managing context-sensitive features . . . 28

2.2.1 Strategies for managing context . . . 28

2.2.2 Hybrid strategies . . . 32

2.2.3 Context dependency in neural networks . . . 33

2.3 Overcoming the slow convergence problems . . . 33

2.3.1 Context sensitive neural network for problem segmentation . . 35

2.4 Context dependent neuron model . . . 37

2.4.1 Context dependent neuron model structure . . . 37

2.4.2 Dependence of weights on context variables 39
2.4.3 Mathematical model of a context dependent net 39
2.4.4 Minimizing the error of a context dependent Network 40
2.4.5 Learning algorithm for feed-forward back-propagation context dependent networks 40
2.4.6 Example: XOR problem 42

3 Overlapped neural networks **46**
3.1 Overlapped neural network for function localization 48
3.1.1 Overlapped multi-neural network training 51
3.2 Overlapped self organizing maps 53
3.2.1 HOSOM structure 53
3.2.2 HOSOM training 54
3.3 Shared weight neural networks 57
3.3.1 Shared-weight neural network architecture 57

4 Novel context sensitive model **59**
4.1 Introduction 60
4.2 Model and structure of our network 61
4.2.1 Performance metrics state vector as context dependency 61
4.2.2 Neural network structure 64
4.2.3 SPM and SSE 65
4.2.4 Training algorithm 67
4.2.5 SPM incorporation algorithm 69
4.3 A Simulation Example 69
4.3.1 Neural network design 71
4.3.2 Simulation results 71
4.4 Conclusion 75

5 Conclusion **77**

References **80**

List of Tables

1.1 The most commonly used activation functions. 4

2.1 Duality of relevance and context-sensitivity 25
2.2 Example of the different types of features 26
2.3 Some examples from machine learning literature 29

4.1 Performance metrics after imposing different SPM state vectors 75

List of Figures

1.1 McCulloch and Pitts neuron model. 3
1.2 Illustration of (a) an acyclic graph and (b) a cyclic graph. The cycle in (b) is emphasized with thick lines.. 4
1.3 A three-layer multi-layer perceptron configuration. 5
1.4 MLP example for back-propagation trainingsingle neuron case. 6
1.5 Notations used in a multi-layered MLP neural network model. 8
1.6 Illustration of how the error back-propagation is computed. 9
1.7 Generalization versus memorization. 10
1.8 Factors influencing the generalization for neural networks. 11
1.9 The status of hidden neural connections. 11
1.10 Illustration of the over-fitting dilemma. 15
1.11 Schematic illustration of the VC dimension. 15
1.12 A radial basis function network. 16
1.13 Illustration of the batch process in which the input samples are distributed into sublists under the best-matching models, and then the new models are determined as (generalized) medians of the sublists over the neighborhoods N_i. 21

2.1 The result of combining samples from different contexts. 30
2.2 *Contextual normalization*: The result of combining normalized samples from different contexts. 30
2.3 *Contextual expansion*: The result of combining expanded samples from different contexts. 30
2.4 *Contextual classifier selection*: Different classifiers are used in different contexts. 31
2.5 *Contextual classification adjustment*: The classification is adjusted for different contexts. 31
2.6 *Contextual weighting* : The impact of weighting on classification. . . . 32
2.7 A simple neural network model based on modulation to implement the idea of context sensitivity 34
2.8 Schematic diagram of a context-sensitive network model. 35
2.9 Context dependent neuron model 38

2.10 Five-point XOR problem solved by a two-layer traditional network. (Left) Input points and discriminating lines of both hidden neurons. (Right) Input points transformed by the first layer and the discriminating line of the output neuron. 43
2.11 Five-point XOR problem solved by one context dependent neuron. . . 43
2.12 Five-point XOR problem. 44

3.1 Multiple network, overlapped-multi-network and ordinary feed-forward network. 47
3.2 An OMNN consisting of two parts: main part and partitioning part. 48
3.3 An example of OMNN showing the relationship between partitioning part and main part. 50
3.4 The HOSOM structure. 54
3.5 Standard shared-weight neural network architecture. 57

4.1 In R^N dimensional space, every SPM factor pushes or pulls the output vector towards its satisfactory state as possible. 62
4.2 Model Structure, two overlapped back-propagation neural networks. . 64
4.3 An intersection of two streets with three lanes each, the middle lane is left turn lane. 70
4.4 SSE of MNN sharing all weights with training of both neural networks (Solid), only training RNN (Dashed)[X-axis is the training, Y-axis is the error]. 71
4.5 SSE of MNN sharing no weights (Solid), sharing all weights with RNN (Dashed) [X-axis is the training, Y-axis is the error]. 72
4.6 SSE of RNN sharing no weights (Solid), partially sharing weights with MNN (Dashed)[X-axis is the training, Y-axis is the error]. 72
4.7 Cumulative delayed vehicles over the time resulted by traffic computations over MNN output, [X-axis is the time, Y-axis is the number of cumulative delayed vehicles.]. 73
4.8 Cumulative delayed vehicles over the time resulted by traffic computations over MNN output using partial SPM, [X-axis is the time, Y-axis is the number of cumulative delayed vehicles.]. 75

Acknowledgments

I would like to thanks my professors at lakehead university who provided me with the best guidance to my research

Abstract

The use of context dependency in neural networks is an important issue in many cognitive situations. In this report we introduce a novel context dependent neural network model based on overlapped multi-neural network structures. We present a detailed study about contextual features and some of its applications in neural networks. We also present some different strategies for applying overlapping in neural networks.

The generalization ability of a neural network is mainly influenced by three factors, the number and performance of the learning data samples, the complexity of the learning algorithm employed, and the network size. Neural network overlapping is one of the practical techniques of achieving a better generalization and recognition rate. This is due to its ability of decreasing the number of free weights of a neural network and providing less complexity of the neural network function. For this purpose overlapped neural networks have been used in feed-forward neural networks (MFNN) , self organizing maps (SOM) and in shared weight neural networks (SWNN). Overlapped neural networks also have the ability of performing a function localization over the neural network feature space.

Among the feature space of any problem, three different types of features (from the relevance point of view) can be distinguished: primary, contextual, and irrelevant features. Researches in the contextual features are mainly concerned with two issues. Identifying such contextual features, and managing them. We are presenting the strategy of identifying these context-sensitive features and five basic strategies for managing them. We are also presenting a context sensitive model for overcoming the slow convergence problems, and a context dependent (cd) neuron model that is considered a generalization of the traditional neuron model.

We introduce a novel approach for problems regardless of sufficiency or accuracy of their historical observations or lab simulation data. Our approach is based on imposing a context of problem performance metrics into networks and gaining the enhancement towards its satisfactory state. We use an overlapped system of back propagation neural networks for our purpose. A main neural network is responsible for mapping input and output relation while a regulatory neural network evaluates the performance metrics satisfaction. We provide special training and testing algorithms for the overlapped system that guarantees a synchronized solution for both neural networks. An example of traffic control problem is simulated. The result of simulation shows a great enhancement of the solution using our approach.

Chapter 1

Introduction

Since late 1980s there has been an explosion in research of neural networks. Today, neural network successful applications are reported across a big range of fields.

Neural network is a paradigm of learning tool, which is able to discover underlying dependencies between the given inputs and outputs by using training data sets. After the training process, it represents high-dimensional nonlinear functions.

Many research institutions, industries, and commercial firms have already started to apply neural network successfully to many diverse types of real world problems. The most important applications include the following, (See [19] and its references)

- Classification and pattern recognition for visual, sound, olfactory and tactile patterns.
- Time series forecasting for financial, weather, engineering time series.
- Diagnostics, e.g., in medicine or engineering.
- Robotics, including control, navigation, coordination, object recognition problems.
- Process control, like nonlinear and multivariate control of chemical plants, power stations and vehicles or missiles.
- Optimization, such as combinatorial problems, e.g., resource scheduling and routing.
- Signal processing, speech and word recognition.
- Machine vision, e.g., inspection in manufacturing, check reader, face recognition and target recognition.
- Financial forecasting for interest rates and stock indices, currencies.
- Financial services, like credit worthiness, forecasting and data mining , services for trade like segmentation of customer data.

A Neural network function differs based on its application, e.g. In certain application areas, such as speech and word recognition, neural networks outperform conventional statistical methods. While in other fields, such as specific areas in robotics and financial services, they show promising application in real world situations. One of the first successful applications was the NETtalk project (Sejnowski and Rosenberg 1987), aimed at training a neural network to pronounce English text consisting of seven consecutive characters from written text, presented in a moving window that gradually scanned the text.

The nonlinear nature of neural networks, the ability of neural networks to learn from their environments in supervised and unsupervised ways, as well as the universal approximation property of neural networks make them highly suited for solving difficult signal processing problems. For practical understanding of neural networks, it is imperative to develop a proper understanding of basic neural network structures and how they impact training algorithms and applications.

A challenge in surveying the field of neural network paradigms is to identify those neural network structures that have been successfully applied to solve real world problems from those that are still under development or have difficulty scaling up to solve realistic problems. It is also critical to understand the nature of the problem formulation so that the most appropriate neural network paradigm can be applied. In addition, it is also important to assess the impact of neural networks on the performance, robustness, and cost-effectiveness of the systems.

1.1 Artificial neural network basics

A Structure is the first step of understanding neural networks. In general a neural networks consists of a set of simple analog signal processors called "*processing elements*" or "*neurons*" connected through weighted links called "*connections*". Each processing element works by itself as a processing element on its inputs that comes to it through its input connections and generate one output and then spread it over its output connections to be processed again by other connected processing element. These connections are having some feature of changing the strength of the passing signal according to its connection weight. The output connection from a neuron can be an input to another neuron or a final output of the neural network. The input connection to a neuron can be an output of another neuron or an initial input to the neural network. The processing that is accomplished at any neuron over its inputs is established through applying a specific function called *net function.* The output of this function is the value of the neuron after processing its inputs. This value is called a *net value.* Another function called *activation function* or *output function* is then applied over the net value to produce the *neuron's output value* associated to the current inputs.

In neural network processing begins with the entire network in a quiescent state, an external comprised of a set of signals to be processed by the network is applied to the input layer, then each processing element generates a single output signal with magnitude that is a function of the total simulations received by the unit. Collectively , the output produced by all processing elements on the layer are then passed as input pattern to the subsequent layer , until the final layer produces output for the current input pattern (see [23]).

1.1.1 Basic neural network components

Among numerous neural network models that have been proposed over the years, all share the neuron as a common building block for its networked interconnected structures. The most widely used neuron model is McCulloch and Pitts neuron model illustrated in Fig 1.1.

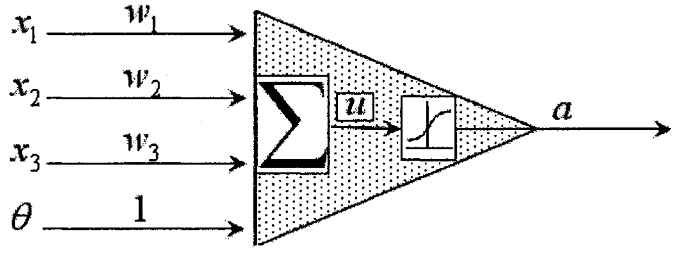


Figure 1.1: McCulloch and Pitts neuron model.

In Fig 1.1, each neuron consists of two parts, the net function and the activation function. The net function determines how the network inputs $\{x_j : 1 \leq j \leq N\}$ are combined inside the neuron. In this figure, a weighted linear combination is adopted:

$$u = \sum_{j=1}^{N} w_j x_j + \theta \tag{1.1}$$

$\{w_j : 1 \leq j \leq N\}$ are parameters expressing the synaptic weights. The quantity θ is called the *bias* and is used to model the threshold.

The output of the neuron, denoted by a in this figure, is related to the network input u via a linear or nonlinear transformation by the activation function:

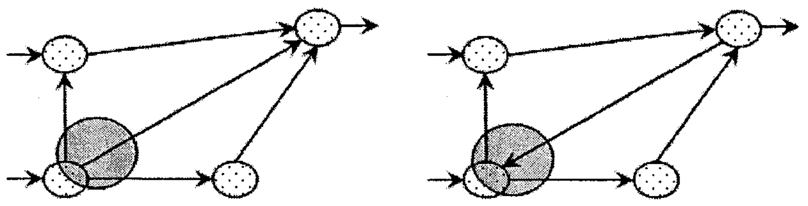
$$a = f(u) \tag{1.2}$$

In various neural network models, different activation functions have been proposed. The most commonly used activation functions are summarized in Table 1.1 , (see [23])

Activation Function	Formula
Sigmoid	$f(u) = \frac{1}{1+e^{-u/T}}$
Hyperbolic tangent	$f(u) = tanh(\frac{u}{T})$
Threshold	$f(u) = \begin{cases} 1 & \text{if } u \geq \text{Threshold} \\ -1 & \text{if } u < \text{Threshold} \end{cases}$
Gaussian radial basis	$f(u) = exp\left[-u\|u-m\|^2/\sigma^2\right]$ where m and σ are parameters to be specified
Linear	$f(u) = au + b$

Table 1.1: The most commonly used activation functions.

1.1.2 Neural network topology



(a) Acyclic topology. (b) Cyclic topology.

Figure 1.2: Illustration of (a) an acyclic graph and (b) a cyclic graph. The cycle in (b) is emphasized with thick lines..

In a neural network, multiple neurons are interconnected to form a network to facilitate distributed computing. Schematically, the configuration of the interconnections can be described efficiently with a directed graph. A directed graph consists of nodes (neurons) and directed arcs (synaptic links). The topology of the graph can be categorized as either acyclic or cyclic. Refer to Fig 1.2(a); a neural network with acyclic topology consists of feed-forward loops. Such an acyclic neural network is often used to approximate a nonlinear mapping between its inputs and outputs. As shown in Fig 1.2(b), a neural network with cyclic topology contains at least one cycle formed by directed arcs. Such a neural network is also known as a recurrent network. Due to the feedback loop, a recurrent network leads to a nonlinear dynamic system model that contains internal memory.

A special and common case, is the multi layered neural networks, in which the processing elements are grouped together into a layer structure where each processing element on each layer performs an analog integration on its inputs to determine its net and activation value.

1.1.3 Multi-layer perceptron (MLP) model

This is the most well known and most popular neural network among all the existing neural network paradigms. It consists of a feed-forward, layered network of neurons. Each neuron in an MLP has a nonlinear activation function that is often continuously differentiable. Some of the most frequently used activation functions for MLP include the sigmoid function and the hyperbolic tangent function. A typical MLP configuration is depicted in Fig 1.3. Each triangle represents an individual neuron. These neurons are organized in layers, labelled as the hidden layer #1, hidden layer #2, and the output layer in this figure. While the inputs at the bottom are also labelled as the input layer, there is usually no neuron model implemented in that layer. The name hidden layer refers to the fact that the output of these neurons will be feeded into upper layer neurons and, therefore, is hidden from the user who only observes the output of neurons at the output layer. Fig 1.3 illustrates a popular configuration of MLP where interconnections are provided only between neurons of successive layers in the network. In practice, any acyclic interconnections between neurons are allowed.

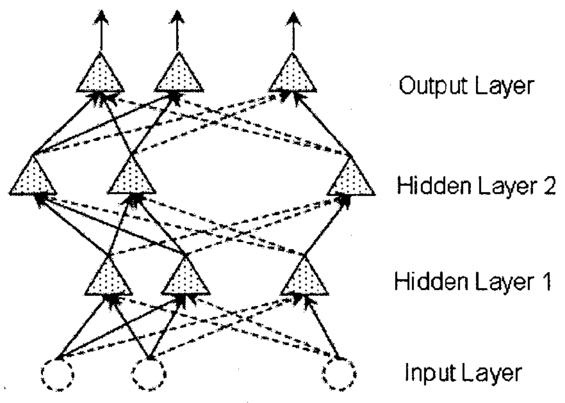


Figure 1.3: A three-layer multi-layer perceptron configuration.

It has been proven that with a sufficient number of hidden neurons, an MLP with as few as two hidden layer neurons is capable of approximating an arbitrarily complex mapping within a finite support (see [23] and its references).

1.1.4 Error back-propagation training of MLP

The key step in applying an MLP model is to find the proper weight matrices. Assuming a layered MLP structure, the weights feeding into each layer of neurons form a weight matrix of that layer. The values of these weights can be found using the error back-propagation training method.

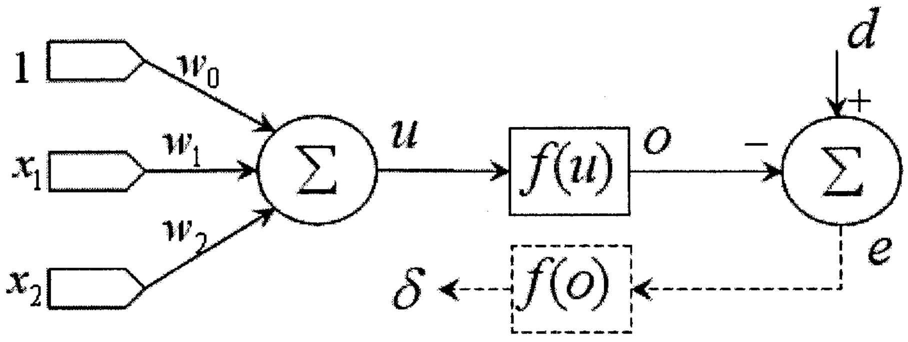


Figure 1.4: MLP example for back-propagation trainingsingle neuron case.

Let us first consider a simple example consisting of a single neuron to illustrate this procedure. Fig 1.4 represents the neuron in two separate parts: a summation unit to compute the net value u, and a nonlinear activation function to computer the neuron's output $o = f(u)$. Then the output o is to be compared with a desired target value d, and their difference will be computed as the error e. There are two inputs $\langle x_1, x_2 \rangle$ with corresponding weights w1 and w2. The input labelled with a constant 1 represents the bias. Here, the bias link weight is labelled w_0. The net value is computed as:

$$u = \sum_{i=0}^{2} w_i x_i = W_X. \tag{1.3}$$

where $x_0 = 1$, $W = [w_0\ w_1\ w_2]$ is the weight matrix, and $x = [1\ x_1\ x_2]^T$ is the input vector, ^{T} is the matrix transpose. Given a set of training samples $\{[x(k), d(k)] : 1 \leq k \leq K\}$, the error back-propagation training begins by feeding all K inputs through the MLP network and computing the corresponding output $\{o(k) : 1 \leq k \leq K\}$. We usually use an initial random setup for the weight matrix W, although some researchers like [6] have provided better guess for the initial setup of the weight matrix. Then a sum of square error will be computed as:

$$E = \sum_{k=1}^{K} [e(k)]^2 = \sum_{k=1}^{K} [d(k) - o(k)]^2 = \sum_{k=1}^{K} [d(k) - f(W_X(k))]^2. \tag{1.4}$$

The objective now is to adjust the weight matrix W to minimize the error E. This leads to a nonlinear least square optimization problem. There are numerous nonlinear optimization algorithms available to solve this problem. Basically, these algorithms adopt a similar iterative formulation:

$$W(t+1) = W(t) + \Delta W(t). \tag{1.5}$$

where $\Delta W(t)$ is the correction made to the current weights $W(t)$. Different algorithms differ in the form of $\Delta W(t)$. The basis of the error back-propagation learning algorithm is called the steepest descend gradient method where

$$\Delta W(t) = -\eta \frac{\partial E}{\partial W} \tag{1.6}$$

Here η is caller the learning factor. Usually, it is a value between 0 and 1, it is specified by the network designer.

The derivative of the scalar quantity E with respect to individual weights can be computed as follows:

$$\frac{\partial E}{\partial w_i} = \sum_{k=1}^{K} \frac{\partial [e(k)]^2}{\partial w_i} = \sum_{k=1}^{K} 2[d(k) - o(k)] \left(-\frac{\partial o(k)}{\partial w_i} \right) \text{ for } i = 0, 1, 2. \tag{1.7}$$

Where

$$\frac{\partial o(k)}{\partial w_i} = \frac{\partial f(u)}{\partial u} \times \frac{\partial u}{\partial w_i} = f'(u) \frac{\partial}{\partial w_i} \left(\sum_{j=0}^{2} w_j x_j \right) = f'(u) x_i. \tag{1.8}$$

Hence

$$\frac{\partial E}{\partial w_i} = -2 \sum_{k=1}^{K} [d(k) - o(k)] f'(u(k)) x_i(k). \tag{1.9}$$

With $\delta(k) = [d(k) - o(k)] f'(u(k))$, the above equation can be expressed as:

$$\frac{\partial E}{\partial w_i} = -2 \sum_{k=1}^{K} \delta(k) x_i(k). \tag{1.10}$$

$\delta(k)$ is the error that represents the amount of correction needed to be applied to the weight w_i for the given input $x_i(k)$. The overall change Δw_i is thus the sum of such contribution over all K training samples. Therefore, the weight update formula has the format of:

$$w_i(t+1) = w_i(t) + \eta \sum_{k=1}^{K} \delta(k) x_i(k). \tag{1.11}$$

If a sigmoid activation function $f(u) = \frac{1}{1+e^{-u/T}}$ is used,then the derivative $f'(u)$ is

$$f'(u) = f(u)[1 - f(u)]. \tag{1.12}$$

Then $\delta(k)$ can be computed as:

$$\delta(k) = \frac{\partial E}{\partial u} = [d(k) - o(k)] \times o(k) \times [1 - o(k)]. \tag{1.13}$$

So far, we discussed how to adjust the weights of an MLP with a single layer of neurons.

Lets now discusses how to perform training for a multiple layer MLP. First, some new notations are adopted to distinguish neurons at different layers. In Fig 1.5, the net-function and output corresponding to the k-th training sample of the j-th neuron of the $(L-1)$-th are denoted by $u_j^{L-1}(k)$ and $o_j^{L-1}(k)$, respectively. The input layer is the 0-th layer. In particular, $o_j^0(k) = x_j(k)$. The output is feeded into the i-th neuron of the L-th layer via a synaptic weight denoted by $w_{ij}^L(t)$ or, for simplicity, w_{ij}^L , since we are concerned with the weight update formulation within a single training epoch.

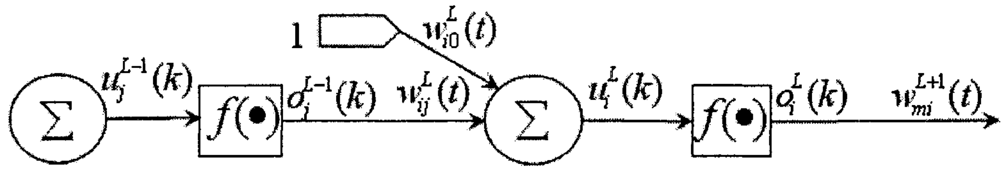


Figure 1.5: Notations used in a multi-layered MLP neural network model.

To derive the weight adaptation equation, $\partial E/\partial w_{ij}^L$ must be computed:

$$\frac{\partial E}{\partial w_{ij}^L} = -2\sum_{k=1}^{K} \frac{\partial E}{\partial u_i^L(k)} \times \frac{\partial u_i^L(k)}{\partial w_{ij}^L} = -2\sum_{k=1}^{K} \left[\delta_i^L(k) \times \frac{\partial}{\partial w_{ij}^L} \sum_{m=1}^{M} w_{im}^L o_m^{L-1}(k)\right]$$

$$= -2\sum_{k=1}^{K} \delta_i^L(k) \times o_j^{L-1}(k). \tag{1.14}$$

Where $1 \le m \le M$, and M is the number of neurons in layer $(L-1)$.

In Equation 1.14, the output $o_j^{L-1}(k)$ can be evaluated by applying the k-th training sample $x(k)$ to the MLP with weights fixed to w_{ij}^L. However, the delta error term $\delta_i^L(k)$ is not readily available and has to be computed. Recall that the delta error is

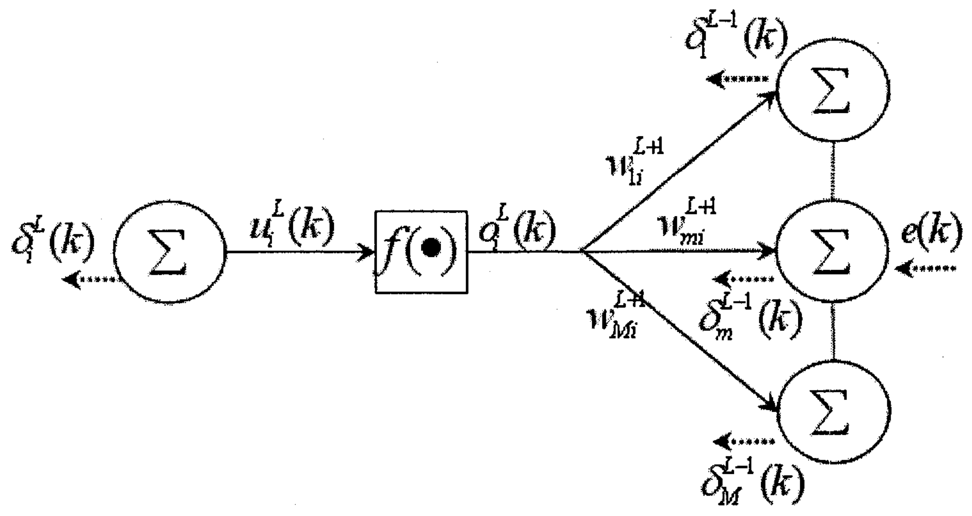


Figure 1.6: Illustration of how the error back-propagation is computed.

defined as $w_{ij}^L(k) = \partial E/\partial u_i^L(k)$. Fig 1.6 is now used to illustrate how to iteratively compute $\delta_i^L(k)$ from $\delta_m^{L+1}(k)$ and weights of the $(L+1)$-th layer.

Note that $o_i^L(k)$ is feeded into all M neurons in the $(L+1)$-th layer. Hence:

$$\delta_i^L(k) = \frac{\partial E}{\partial u_i^L(k)} = \sum_{m=1}^{M} \frac{\partial E}{\partial u_m^{L+1}(k)} \times \frac{\partial u_m^{L+1}(k)}{\partial u_i^L(k)} = \sum_{m=1}^{M} \left[\delta_m^{L+1}(k) \times \frac{\partial}{\partial u_i^L(k)} \sum_{j=1}^{J} w_{mj}^L f\left(u_j^L(k)\right) \right]$$

$$= f'\left(u_i^L(k)\right) \times \sum_{m=1}^{M} \left(\delta_m^{L+1}(k) \times w_{mi}^L\right). \tag{1.15}$$

Equation 1.15 is the error back-propagation formula that computes the delta error from the output layer back toward the input layer, in a layer-by-layer manner.

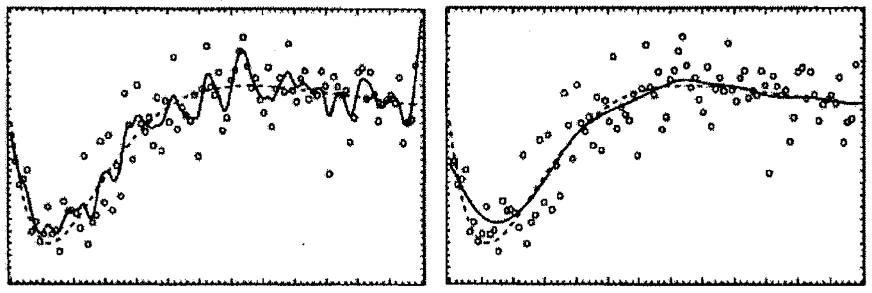
1.2 Generalization ability of a neural network

The basic topics of multi-layered feed-forward neural networks (MFNNs), such as the network structures, mathematical descriptions, and back-propagation learning algorithms were discussed in the previous section. Beyond these aspects, significant progress has been made on many related issues. In fact, numerous extensions to the basic MFNNs with the back-propagation algorithm have emerged. Most of these were developed to overcome some of the inherent limitations of the basic back-propagation learning algorithms. These extensions have involved the alternative error measure criteria for the standard back-propagation learning algorithm, complex regularization techniques for both improving the generalization capability of MFNNs and pruning the networks, sensitivity calculation based network pruning techniques for the purpose of optimizing the network structure and accelerating the learning phase, the procedures of dealing with the second derivatives of MFNNs to improve the convergence speed of the back-propagation algorithm, and many other advanced studies are still in progress for adapting the learning of MFNNs.

In this section we will concentrate on neural network generalization ability in particular in studding some extensions of the back-propagation that provides enhancement to the training process as well as the generalization ability of the neural network. (see [9])

Definition 1.1. ***Generalization ability*** *of a neural network determines how well the mapping surface of the network will renderer the unseen inputs to the output space.*

Fig 1.7 shows two symbolic cases of a neural network convergence in Fig 1.7(a) the wiggly curve shows that the neural network function is being too complex and the network is behaving over-fitting in the training data, in this case the network is memorizing the training data not generalizing them , and hence for any new stimulus the network will be going to categorize the input into one of the memorized classes instead of recognize it, while in Fig 1.7(b) the convergence surface is performing better generalization for the unseen inputs because in this case the solution error for every unseen inputs is going to be less while probably the sum squared error over the whole test set would be bigger. However, in classification problems, the maximum recognition error over all samples is the important factor of generalization measure according to Geman and Bienenstock (1992) bias-variance dilemma (see [9] and its references).



(a) Training data (circles) is being memorized. (b) Training data (circles) have been generated.

Figure 1.7: Generalization versus memorization.

As shown in Fig 1.8, generalization is mainly influenced by three factors:

First, the number and performance of the learning data samples, which represent how well the problem at hand is characterized, generally speaking, a larger number

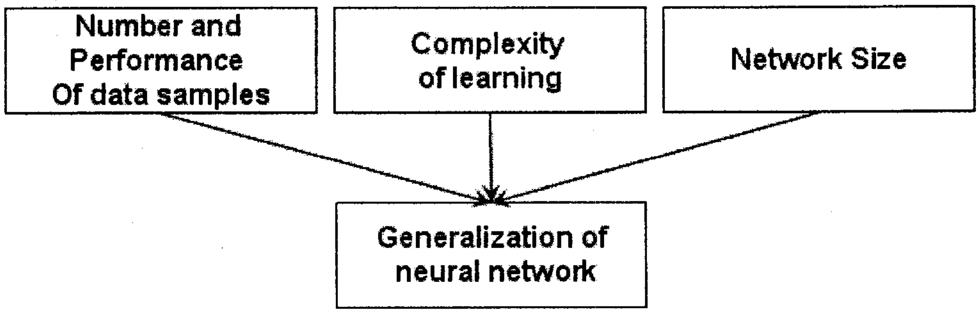
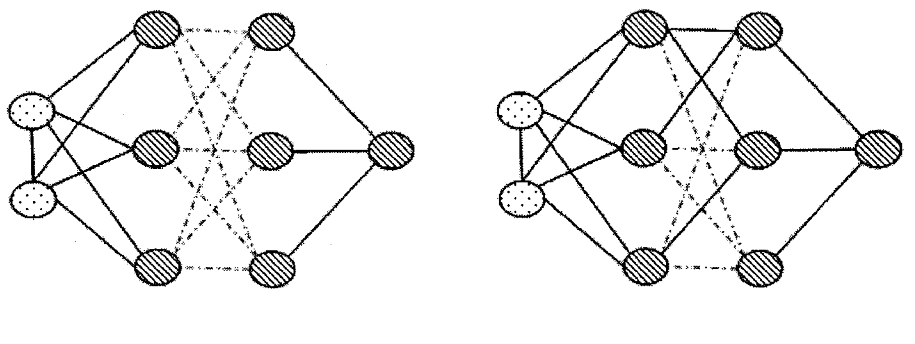

Number and Performance Of data samples
Complexity of learning
Network Size
Generalization of neural network

Figure 1.8: Factors influencing the generalization for neural networks.

of learning data samples can provide a better representation for the underlying problem, and if a suitable learning algorithm and network size are used, a better solution to the problem should be obtained.

Second, the complexity of the learning algorithm employed. It is known that extra training to the neural network result in more function complexity and over-fitting problem which definitely decrease the generalization ability of the neural network.

The third factor of the generalization for the neural networks is the network size. It is generally admitted that generalization of the back-propagation architecture will depend on the relative size of the training data and the trained network size. However, it is observed that the back-propagation networks are sometimes very slow in learning. This is because the synaptic connection weights, especially the hidden connection weights (connections among hidden neurons), are significantly smaller for a large network. This means that the networks cannot utilize hidden connections efficiently. Thus, hidden neurons cannot be appropriately used in speeding up the learning.



(a) All hidden connections are inactive. (b) Some hidden connections are inactive.

Figure 1.9: The status of hidden neural connections.

This situation is illustrated in Fig 1.9. The network in Fig 1.9(a) is a back-propagation network in which the hidden connections among the hidden neurons are inactive, while the input and output connections are active. If the hidden connections are weak; that is, the absolute values of the hidden weights are small, it is certain that the hidden neurons are not appropriately used in speeding up the learning. As shown in Fig 1.9(b), some hidden connections of the network are active.

In this case, the hidden neurons are expected to be used in improving the generalization as well as speeding up the learning. In order to adapt the size of the back-propagation network and activate hidden connections, an approach of complexity regularization may be applied. In this approach, a term is added to the error measure function that discourages the learning algorithm from seeking solutions that are too complex. This term represents, in fact, a measure of the network's complexity; that is, both the quantities and number of weights. The resulting criterion or cost function is of the form

$$Cost = Network\ error\ measure + Model\ complexity\ measure.$$

where the first term on the right-hand side measures the network error between the network outputs and the task or desired outputs, while the second term is determined only by the complexity of the network structure. This type of criterion is sometimes referred to as the minimum description length (MDL) criterion because it has the same form as the information theoretic measure of description length.

Simply speaking, the description length of a set of data is defined as the total number of bits required to represent the data. But for a neural network that is designed to represent a set of data, the total description length should be defined as the sum of the number of bits required to encode the errors. The cost function introduced above may be considered as one such form if the term of the network error measure is related to the number of bits required to encode the errors, and the term of complexity measure corresponds to the number of bits required to describe the network model. The learning process that minimizes this cost function then, to a certain degree, provides a minimal description of the data. In the context of back-propagation learning, or any other supervised learning procedure, such a cost function may be represented in a symbolic way as

$$E_t(w) = E(w) + \lambda E_c(w). \tag{1.16}$$

where the $E(w)$ is the error function used in the standard back-propagation learning, $E_c(w)$ is the complexity measure, and the parameter λ is a small positive constant that is used to control the influence of the term of the complexity measure $E_c(w)$ in relation to the conventional error measure $E(w)$. Consequently, the learning algorithm derived using such a criterion is a simple extension of the back-propagation algorithm. Later, we show the *weight decay* approach as one the approaches may be obtained as a choices of the complexity measure.

1.2.1 Weight decay approach

The weight decay approach is a method of reducing the effective number of weights in the network by encouraging the learning algorithm to seek solutions that use as many zero weights as possible. This is accomplished by adding a term that is the sum of all the squared weights to the criterion function that penalizes the network for using the nonzero weights. Then, the new criterion function is formulated as

$$E_t(w) = E(w) + \frac{\lambda}{2}\|w\| = E(w) + \frac{\lambda}{2}\sum_i w_i^2. \tag{1.17}$$

where the sum in the second term on the right-hand side performed over all the weights represents the complexity measure E_c of the network. It is to be seen that in this modification of the standard back-propagation learning algorithm, an extra term of the form λ_w is added for updating the weight vector. Therefore, one has the following new updating formulation:

$$w(k+1) = w(k) - \eta\left(\frac{\partial E}{\partial w(k)} + \lambda w(k)\right) = (1-\eta\lambda)w(k) - \eta\frac{\partial E}{\partial w(k)}. \tag{1.18}$$

This shows that the effect of λ is to "decay" the weight vector by a factor of $(1-\eta\lambda)$. The weight decay approach does not actually delete weights from the network, nor does it typically produce weights that are exactly zero. Weights that are not essential to the solution decay to zero and can be removed. When some weights are forced to take on values near zero, some other weights remain relatively large. The result is that the average weight size is smaller.

Another simple weight decay method is to define the cost function as

$$E_t(w) = E(w) + \lambda|w|. \tag{1.19}$$

In this case, an additional term $\lambda sgn(w)$ is used in the weight vector updating rule, Equation 1.19. If $W_i > 0$, the weight is decremented by λ; otherwise, if $w_i < 0$, then it is incremented by λ.

We have seen some methods that enhances the network structure and hence the generalization ability. The following is another method that can define a measure of the function complexity of the neural network algorithm.

1.2.2 Complexities in regularization and VC dimension

The VC (Vapnik-Chervonenkis) dimension h is a property of a set of approximating functions of a learning machine that is used in all important results in the statistical learning theory, (see [23]). Despite the fact that the VC dimension is very important, the unfortunate reality is that its analytic estimations can be used only for the simplest sets of functions. Here for simplicity we only present the basic concept of the VC dimension for a two-class pattern recognition case, while it can be generalized for some sets of real approximating functions.

Consider a task of classification, in which we need to find a rule to assign an input to one two different classes. One possible formalization of this task is to estimate a function $f : R^N \rightarrow \{-1, 1\}$ using input-output training data pairs generated identically and independently distributed according to an unknown probability distribution $P(x, y)$

$$(X_1, Y_1), \cdots, (X_n, Y_n) \in R^N \times Y \ : \ Y = \{-1, 1\}$$

such that f will correctly classify unseen examples (X, Y). An example is assigned to class 1 if $f(X) \geq 0$ and to class -1 otherwise. The test examples are assumed to be generated from the same probability distribution $P(X, Y)$ as the training data. The best function f that one can obtain, is the one that minimizes the expected error (Risk):

$$R[f] = \int l(f(X), Y) dP(X, Y). \tag{1.20}$$

where l denotes a suitably chosen loss function. Unfortunately, the risk cannot be minimized directly, since the underlying probability distribution $P(x, y)$ is unknown. Therefore, we must try to estimate a function that is close to the optimal one based on the available information, i.e., the training sample and properties of the function class F the solution f is chosen from. To this end, we need what is called an induction principle. A particular simple induction principle consists of approximating the minimum of the risk in Equation 1.20 by the minimum of the empirical risk

$$R_{emp}[f] = \frac{1}{n} \sum_{i=1}^{n} l(f(X_i), Y_i). \tag{1.21}$$

It is possible to give conditions to the learning machine which ensure that, asymptotically (as $n \rightarrow \infty$), the empirical risk will converge towards the expected risk. However, for small sample sizes, large deviations are possible and over-fitting might occur (see Fig 1.10). Given only a small sample (left), either the solid or the dashed hypothesis might be true, the dashed one being more complex but also having a smaller training error. Only with a large sample are we able to see which decision more accurately reflects the true distribution. If the dashed hypothesis is correct, the solid would under-fit (middle); if the solid were correct, the dashed hypothesis would over-fit (right). Then, a small generalization error can usually not be obtained by simply minimizing the training error (Equation 1.21).

One way to avoid the over-fitting dilemma is to restrict the complexity of the function class F from which one chooses the function f. The intuition, which will be formalized in the following, is that a simple (e.g., linear) function that explains most of the data is preferable to a complex one.

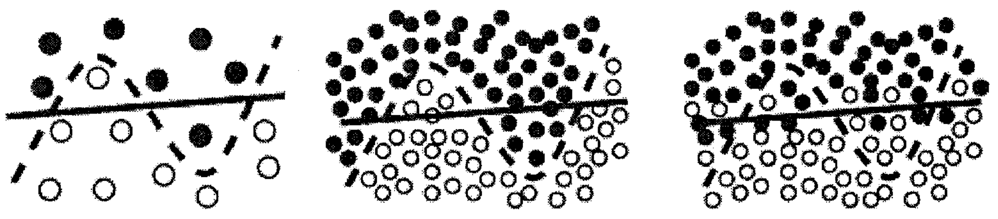


Figure 1.10: Illustration of the over-fitting dilemma.

A specific way of controlling the complexity of a function class is given by Vapnik Chervonenkis (VC) theory and the structural risk minimization (SRM) principle. Here, the concept of complexity is captured by the VC dimension h of the function class F from which the estimate f is chosen. Roughly speaking, the VC dimension measures how many (training) points can be shattered for all possible labelling using functions of the class. Constructing a nested family of function classes $F_1 \subset \cdots \subset F_k$ with non-decreasing VC dimension, the SRM principle proceeds as follows. Let $f_1, \cdots, f_k$ be the solutions of the empirical risk minimization (Equation 1.21) in the function classes F_i . SRM chooses the function class F_i (and the function f_i) such that an upper bound on the generalization error is minimized.

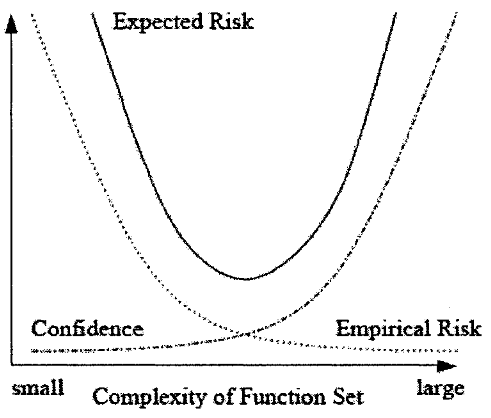


Figure 1.11: Schematic illustration of the VC dimension.

In Fig 1.11, The dotted line represents the training error (empirical risk), and the dash-dotted line represents the upper bound on the complexity term (confidence). With higher complexity, the empirical error decreases but the upper bound on the risk confidence becomes worse. For a certain complexity of the function class, the best expected risk (solid line) is obtained. Thus, in practice, the goal is to find the best trade-off between empirical error and complexity.

Theorem 1.1. *Let h denote the VC dimension of the function class F and let*

R_{emp} be defined by Equation (1.21) using the 0/1-loss. For all $\delta > 0$ and $f \in F$, the inequality bounding the risk

$$R[f] \leq R_{emp}[f] + \sqrt{\frac{h\left(ln\frac{2n}{h}+1\right) - ln(\delta/4)}{n}}. \tag{1.22}$$

holds with probability of at least $1 - \delta$ for $n > h$.

1.3 Highlighting some neural network types

There are around 50 different types of neural networks in use today [23]. According to the propagation direction most of these types can be categorized as either feed-forward or feed-back neural networks. According to the structure growing they may be categorized either as static or dynamic neural networks. In this section we explain two types of them, that are helpful in understanding the following chapters.

1.3.1 Radial basis functions

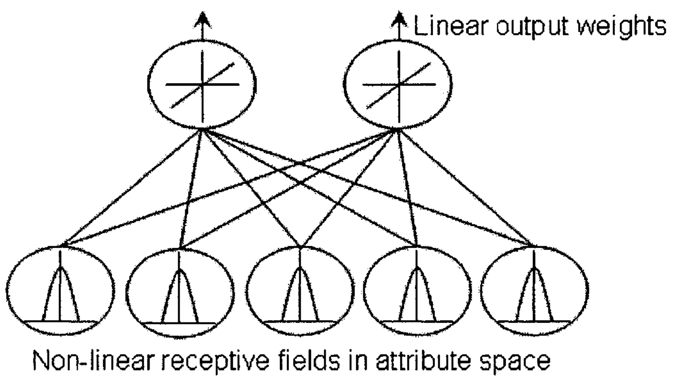


Figure 1.12: A radial basis function network.

The radial basis function network generally consists of two weight layers, a hidden layer of units performing linear or non-linear functions of the attributes, followed by an output layer of weighted connections to nodes whose outputs have the same form as the target vectors (see [10] and its references).

They can be described by the following equation:

$$y = w_0 + \sum_{i=1}^{n_k} w_i f(\|x - c_i\|). \tag{1.23}$$

where f is a radial basis function, w_i is the output layer neuron i weight, w_0 is the output offset, x is the input to the network, c_i is the center associated with the basis function f, n_h is the number of basis functions in the network, and $\| \quad \|$ denotes the Euclidean norm.

Structurally it can be viewed as an MLP with one hidden layer, except that each node of the the hidden layer computes an arbitrary function of the inputs (Gaussian is the most popular), and the transfer function of each output node is the trivial identity function.

Instead of "synaptic strengths" the hidden layer has parameters appropriate for whatever functions are being used; for example, Gaussian widths and positions. This network offers a number of advantages over the MLP under certain conditions, although the two models are computationally equivalent.

These advantages include a linear training rule once the locations in attribute space of the non-linear functions have been determined, and an underlying model involving localized functions in the attribute space, rather than the long-range functions occurring in perceptron-based models. Fig 1.12 shows the structure of a radial basis function. The non-linearities comprise a position in attribute space at which the function is located (often referred to as the functions center), and a non-linear function of the distance of an input point from that center, which can be any function at all. Common choices include a gaussian response function, $exp(-x^2)$ and inverse multi-quadrics ($[z^2 + c^2]^{-\frac{1}{2}}$) as well as non-local functions such as thin plate splines ($z^2 \log z$) and multi-quadrics ($[z^2 + c^2]^{\frac{1}{2}}$). Although it seems counter-intuitive to try and produce an interpolating function using non-localized functions, they are often found to have better interpolating properties *in the region populated by the training data.* The radial basis function network approach involves the expansion or pre-processing of input vectors into a high-dimensional space. This attempts to exploit a theorem of Cover (1965) which implies that,"a classification problem cast in a high-dimensional space is more likely to be linearly separable than would be the case in a low-dimensional space".

Training

In RBF network the training consists of parameterizing the unknown parameters in a particular RBF network. Generally speaking, this means determining (1) the number of basis functions (hidden units), (2) centers and widths of each basis function, and (3) output layer weights. For some algorithms, these steps are carried out separately, while in others, all parameters are found simultaneously. Furthermore, different techniques can be mixed and matched for training the different parameters.

A number of methods can be used for choosing the centers for a radial basis function network. It is important that the distribution of centers in the attribute space should be similar to, or at least cover the same region as the training data. It is assumed that the training data is representative of the problem, otherwise good performance cannot be expected on future unseen patterns.

A first order technique for choosing centers is to take points on a square grid covering the region of attribute space covered by the training data. Alternatively, better performance might be expected if the centers were sampled at random from the training data itself, using some or all samples, since the more densely populated regions of the attribute space would have a higher resolution model than sparser regions. In this case, it is important to ensure that at least one sample from each class is used as a prototype center.

When center positions are chosen for radial basis function networks with localized non-linear functions such as Gaussian receptive fields, it is important to calculate suitable variances, or spreads for the functions. This ensures that large regions of space do not occur between centers, where no centers respond to patterns, and conversely, that no pair of centers respond nearly identically to all patterns. This problem is particularly prevalent in high dimensional attribute spaces because volume depends sensitively on radius. Prager & Fallside (1989) have introduced a quantitative discussion of this point.

The process of optimizing the weights of RBF networks is simply performed by solving a linear system. The same problem arises in ordinary linear regression, the only difference being that the input to the linear system is the output of the hidden layer of the network, not the attribute variables themselves.

Let $y_{ki}^{(H)}$ be the output of the k-th radial basis function on the i-th example. The output of each target node j is computed using the weights w_{jk} as

$$y_{ji} = \sum_k w_{jk} y_{ki}^{(H)}. \tag{1.24}$$

Let the desired output for example i on target node j be Y_j. Then the error is

$$E(w) = \frac{1}{2} \sum_{ji} \left(\sum_k w_{jk} y_{ki}^{(H)} - Y_{ji} \right)^2. \tag{1.25}$$

This follows that

$$\frac{\partial E}{\partial w_{rs}} = \sum_k \sum_i w_{rk} y_{ki}^{(H)} y_{ji}^{(H)} - \sum_i Y_{ri} y_{si}^{(H)}. \tag{1.26}$$

The error is minimum where this derivative vanishes. Let R be the correlation matrix of the radial basis function outputs,

$$R_{jk} = \sum_i y_{ki}^{(H)} y_{ji}^{(H)}. \tag{1.27}$$

The weight matrix W^* which minimizes E lies where the gradient vanishes:

$$W_{jk}^* = \sum_r \sum_i Y_{ji} y_{ri}^{(H)} R_{rk}^{-1}. \tag{1.28}$$

Thus, the problem is solved by inverting the square $(H \times H)$-matrix R, where H is the number of radial basis functions. The matrix inversion can be accomplished by standard methods such as LU decomposition (Renals & Rohwer, 1989) and (Press et. al., 1988) if R is not singular. This is typically the case, but things can go wrong. If two radial basis function centres are very close together a singular matrix will result, and a singular matrix is guaranteed if the number of training samples is not at least as great as H There is no practical way to ensure a non-singular correlation matrix. Consequently the safest course of action is to use a slightly more computationally expensive singular value decomposition method. Such methods provide an approximate inverse by diagonalizing the matrix, inverting only the eigenvalues which exceed zero by a parameter-specified margin, and transforming back to the original coordinates. This provides an optimal minimum-norm approximation to the inverse in the least-mean-squares sense.

Another approach to the entire problem is possible (Broomhead & Lowe, 1988) . Let n be the number of training examples. Instead of solving the $H \times H$ linear system given by the derivatives of E in Equation 1.26, this method focuses on the linear system embedded in the error formula (1.24) itself:

$$\sum_k w_{jk} y_{ki}^{(H)} = Y_{ji}. \tag{1.29}$$

Unless $n = H$, this is a rectangular system. In general an exact solution does not exist, but the optimal solution in the least-squares sense is given by the pseudo-inverse (Kohonen,1989) $y^{(H)^+}$ of $y^{(H)}$, for the matrix with elements $Y_{ji}^{(H)}$:

$$W^* = Y y^{(H)^+}. \tag{1.30}$$

This formula is applied directly. The identity $Y^+ = Y^T(YY^T)^+$, ^{T} denotes the matrix transpose, can be applied to Equation 1.30 to show that the pseudo-inverse method gives the same result as Equation 1.28

$$W^* = Y y^{T^{(H)}} \left(Y^{(H)} Y^{T^{(H)}} \right)^+. \tag{1.31}$$

The requirement to invert or pseudo-invert a matrix dependent on the entire data-set makes this a batch method. However an online variant is possible, known as Kalman Filtering (Scalero & Tepedelenlioglu, 1992). It is based on the somewhat remarkable fact that an exact expression exists for updating the inverse correlation R^{-1} if another example is added to the sum in Equation 1.27, which does not require re-computation of the inverse.

1.3.2 The basic SOM

The Self-organizing Map (SOM) is an effective software tool for the visualization of high-dimensional data. In its basic form it produces a similarity graph of input data. It converts the nonlinear statistical relationships between high-dimensional data into simple geometric relationships of their image points on a low-dimensional display, usually a regular two-dimensional grid of nodes. As the SOM thereby compresses information while preserving the most important topological and/or metric relationships of the primary data elements on the display, it may also be thought to produce some kind of abstractions (see [8]).

The SOM may be described formally as a nonlinear, ordered, smooth mapping of high-dimensional input data manifolds onto the elements of a regular, low-dimensional array. This mapping is implemented in a way that resembles the classical vector quantization as follows:

Assume first for simplicity that the set of input variables $\{\xi_j\}$ is definable as a real vector $x = [\xi_l, \xi_2, \cdots, \xi_n]^T \in R^n$. With each element in the SOM array we associate a parametric real vector $m_i = [\mu_l, \mu_2, \cdots, \mu_n]^T \in R^n$ that is called a model. Assuming a general distance measure between x and m_i denoted $d(x, m_i)$, the image of an input vector x on the SOM array is defined as the array element m_c that matches best with x, i.e., that has the index

$$c = arg\min_i\{d(x, m_i)\}. \tag{1.32}$$

Differing from the traditional vector quantization, the task is to define m_i in such a way that the mapping is ordered and descriptive of the distribution of x.

Consider Fig 1.13 where a two-dimensional ordered array of nodes, each one having a general model m_i associated with it. The initial values of the m_i may be selected as random, preferably from the domain of the input samples in a symmetric way.

Then consider a list of input samples $x(t)$, where t is an integer-valued index. Let us recall that in this scheme, the $x(t)$ and m_i may be vectors, strings of symbols, or even more general items. Compare each $x(t)$ with all the m_i and copy each $x(t)$ into a sublist associated with that node, the model vector of which is most similar to $x(t)$ relating to the general distance measure.

When all the $x(t)$ have been distributed into the respective sublists in this way, consider the neighborhood set N_i around model m_i. Here N_i consists of all nodes up to a certain radius in the grid from node i. In the union of all sublists in N_i, the next task is to find the "middlemost" sample $\bar{x}_i$, defined as that sample that has the smallest sum of distances from all the samples $x(t)$, $t \in N_i$. This sample $\bar{x}_i$ is now called the *generalized median* in the union of the sublists. If $\bar{x}_i$ is restricted to being one of the samples $x(t)$, we shall indeed call it the *generalized set median*; on the other hand, since the $x(t)$ may not cover the whole input domain, it may be possible to find another item $\bar{x}'_i$ that has an even smaller sum of distances from the $x(t)$, $t \in N_i$. For clarity we shall then call $\bar{x}'_i$ the *generalized median.*

Also notice that for the Euclidean vectors the generalized median is equal to their arithmetic mean if we look for an arbitrary Euclidean vector that has the

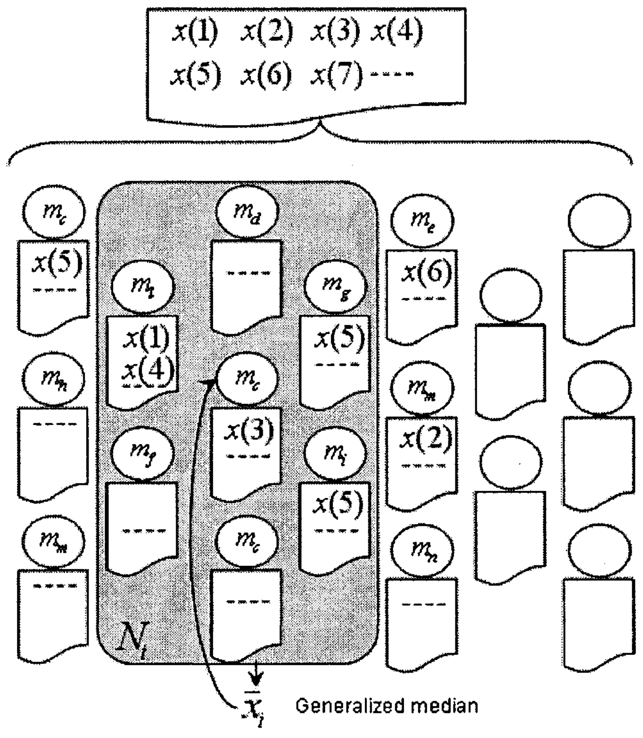


Figure 1.13: Illustration of the batch process in which the input samples are distributed into sublists under the best-matching models, and then the new models are determined as (generalized) medians of the sublists over the neighborhoods N_i.

smallest sum of squares of the Euclidean distances from all the samples $x(t)$ in the union of the sublists.

The next phase in the process is to form $\bar{x}_i$ or $\bar{x}'_i$ for each node in the above manner, always considering the neighborhood set N_i around each node i, and to replace each old value of m_i by $\bar{x}_i$ or $\bar{x}'_i$, respectively, in a simultaneous operation.

The above procedure shall now be iterated: in other words, the original $x(t)$ are again distributed into the sublists (which now change, because the m_i have changed), and the new $\bar{x}_i$ or $\bar{x}'_i$ are computed and made to replace the m_i, and so on. This is a kind of regression process.

Chapter 2

Context dependent neural networks

The wealth of information from the neuronal morphology of the brain is primarily the motivation for such an exciting state of the research in neural networks. One observation that was strongly employed in neural networks is the context dependency of the biological neural system, this observation simply states that the biological brain reacts differently to the same inputs if they were applied in different contexts.

The results of Wrobel A,(1998) in [20] of measuring potential in rat barrel cortex evoked by vibrissa stimulation are reported the conclusion as follows, "We hypothesize that neuromodulatory action elicited by contextual stimulation activates all neurons in the principal barrel column, including those providing an output to the surrounding barrels. This mechanism may lead to experience-dependent changes within intracortical network."

A simple example was given in [2] about that context dependency, stated that "Our reception is narrower when we are frightened or angry".

In this chapter we first introduce the definitions of context by Peter Turney (1996) and the strategy of identifying the context sensitive features in Section 2.1. The strategies of managing the context sensitive features are then described in Section 2.2, then we focus on two different techniques of managing the context sensitive features in neural networks to establish better performance are explained in Section 2.3 and 2.4 (see [16] and its references).

2.1 Introduction to contextual features

"Context" is a will-defined term. Here we are concerned with a specific type of context that influences decision making or any type of information processing of a contextual problem. In general, researches that involve contextual features are mainly concerned with two issues. The first issue is identifying such contextual features among the whole feature space of a problem. The second issue is managing these contextual features, in which researchers are concentrating on developing different techniques of managing these features and benefit from them.

The classification of such contextual features can help in creating symmetric environments for all primary data within one context class. Recent work has demonstrated that, the strategies for exploiting contextual information can improve the performance and efficiency of machine learning algorithms.

In this section we describe the strategy of identifying contextual features for a specific type of context. In particular, the contextual features in supervised concept learning.

Assume the standard machine learning model of concept learning, where data are represented as vectors in a multi dimensional feature space. The feature space is partitioned into a finite set of classes. And the training data are labelled according to its association with the different classes. In many concept learning problems, it is possible to use common-sense knowledge to divide the features into three classes: primary features, contextual features, and irrelevant features.

Primary features are useful for classification even when they are considered in isolation, without the other features. *Contextual features* are useful for classification only when they are considered in combination with other features. And *irrelevant features* are not useful, either in isolation or in combination with other features. For more understanding of these three types, example, "When classifying spoken vowels, the primary features are based on the sound spectrum. The accent of the speaker is a contextual feature. The color of the speakers hair is irrelevant."

Surprisingly, the identification problem has received little attention in the research, perhaps because common-sense makes the problem seem trivial. However, learning systems that can both identify and manage contextual features may have a substantial advantage over the systems that only manage them. A precise definition of context is the first step in the construction of such identification systems.

2.1.1 Definition of context

Peter Turney's definition for context in (1993), did not consider the the possibility of weakly relevant features. In the light of the definitions given by John et al. (1994), Turney introduced new definition that does not have this problem in (1996).

Suppose we have m dimensional feature space $F_1 \times F_2 \times \cdots \times F_m$ where F_i is the domain of the i-th feature. Let C be a finite set of classes. A training instance is in the form $\langle \vec{X}, Y \rangle$ where $\vec{X} \in F_1 \times F_2 \times \cdots \times F_m$ and $Y \in C$.

Assume that instances are sampled from $F_1 \times F_2 \times \cdots \times F_m \times C$ identically and independently with a probability distribution p:

$$p : F_1 \times F_2 \times \cdots \times F_m \times C \rightarrow [0, 1]. \tag{2.1}$$

In an instance of form $\langle \vec{X}, Y \rangle$, where $\vec{X} = \langle X_1, X_2, \cdots, X_m \rangle$ and X_i represents the i-th feature , x_i represents the value of the i-th feature and similarly y is the value of Y.

Given the feature values for a new instance, $X_1 = x_1, \cdots, X_m = x_m$ the learning algorithm can predict the class of the instance, $Y = y$.

Let S_i be the set of all features except X_i , i,e.

$$S_i = \{X_1, \cdots, X_{i-1}, X_{i+1}, \cdots, X_m\}. \tag{2.2}$$

Let s_i be an assignment of values to all of the features in S_i

Definition 2.1. *Suppose that X_i is either strongly relevant or weakly relevant. By definition there is a subset of features $S_i^{'}$ of S_i and an assignment of values $s_i^{'}$ to $S_i^{'}$ such that:*

$$p(Y = y | X_i = x_i, S_i^{'} = s_i^{'}) \neq p(Y = y | S_i^{'} = s_i^{'}). \tag{2.3}$$

$$\phi \subseteq S_i^{'} \subseteq S_i. \tag{2.4}$$

There may be several subsets that satisfy (Equation 2.3 and 2.4). Each such subset $S_i^{'}$ defines a context in which the feature X_i is (strongly or weakly) relevant. Let α_i be the cardinality of the smallest subset (or subsets) for which X_i is relevant. Let β_i be the cardinality of the largest subset (or subsets) for which X_i is relevant. α_i is called the minimum context size and β_i is called the maximum context size. When X_i is irrelevant, both α_i and β_i and are undefined.

It follows from Definition 2.1 that $0 \leq \alpha_i \leq \beta_i \leq m - 1$. It is easy to see that X_i is strongly relevant when $\beta_i = m - 1$ and weekly relevant when $\beta_i < m - 1$.

Definition 2.2. *The feature X_i is primary if and only if $\alpha_i = 0$.*

A primary feature is relevant even when the context is the empty set. That is, if X_i is primary, then there exists some x_i and y for which $p(X_i = x_i) > 0$ such that:

$$p(Y = y | X_i = x_i) \neq p(Y = y). \tag{2.5}$$

Definition 2.3. *The feature X_i is contextual iff $\alpha_i > 0$.*

A contextual feature is only relevant when considered in some(non-empty) context. A contextual feature is irrelevant when considered in isolation, that is, if X_i is contextual feature, then for all x_i and y:

$$p(Y = y | X_i = x_i) = p(Y = y). \tag{2.6}$$

A contextual feature may be either strongly or weakly relevant.

The distinction between primary and contextual is dual to the distinction between weakly relevant and strongly relevant. As illustrated in Table 2.1.

Term	Definition	Dual
strongly relevant	$\beta_i = m - 1$	primary
weakly relevant	$\beta_i < m - 1$	contextual
primary	$\alpha_i = 0$	strongly relevant
contextual	$\alpha_i > 0$	weakly relevant

Table 2.1: Duality of relevance and context-sensitivity

We have defined primary features and contextual features. Now lets see what it means for one feature to be context-sensitive to another. Let $S_{i,j}$ be the set of all features except X_i and X_j, i.e.

$$S_i = \{X_1, \cdots, X_{i-1}, X_{i+1}, \cdots X_{j-1}, X_{j+1}, \cdots, X_m\} \tag{2.7}$$

Let $s_{i,j}$ be an assignment of values to all of the features in $S_{i,j}$.

Definition 2.4. *The feature X_i is weakly context-sensitive to the feature X_j if and only if there exists a subset of features $S'_{i,j}$ of $S_{i,j}$ for which there exists some x_i,x_j,$s'_{i,j}$ and y for which $p(X_i = x_i, X_j = x_j, S'_{i,j} = s'_{i,j}) > 0$ such that the following two conditions hold:*

$$p(Y = y|X_i = x_i, X_j = x_j, S'_{i,j} = s'_{i,j}) \neq p(Y = y|X_j = x_j, S'_{i,j} = s'_{i,j}). \tag{2.8}$$

$$p(Y = y|X_i = x_i, X_j = x_j, S'_{i,j} = s'_{i,j}) \neq p(Y = y|X_i = x_i, S'_{i,j} = s'_{i,j}). \tag{2.9}$$

In this definition, the first condition (Equation 2.8) means that, the feature X_i must be relevant in some context that includes the feature X_j . The second condition (Equation 2.9) means that, the feature X_j is an essential (non-redundant) component of the context. The symmetry of these two conditions implies that X_i is weakly context-sensitive to X_j iff X_j is weakly context-sensitive to X_i.

Definition 2.5. *The feature X_i is strongly context-sensitive to the feature X_j iff X_i is a primary feature, X_j is a contextual feature and X_i is weakly context-sensitive to X_j.*

2.1.2 Illustration example of the contextual features definitions

In this simple example,the features and the class are boolean:

$$F_1 = F_2 = F_3 = C = \{0, 1\}. \tag{2.10}$$

Class	Primary	Contextual	Irrelevant	Probability
Y	X_1	X_2	X_3	p
0	0	0	0	0.03
0	0	0	1	0.03
0	0	1	0	0.08
0	0	1	1	0.08
0	1	0	0	0.07
0	1	0	1	0.07
0	1	1	0	0.07
0	1	1	1	0.07
1	0	0	0	0.07
1	0	0	1	0.07
1	0	1	0	0.07
1	0	1	1	0.07
1	1	0	0	0.03
1	1	0	1	0.03
1	1	1	0	0.08
1	1	1	1	0.08

Table 2.2: Example of the different types of features

Table 2.2 shows the probability distribution and illustrates the above definition.$p : F_1 \times F_2 \times F_3 \times C \rightarrow [0,1]$.
From the table $p(Y = 1) = 0.5$ and $p(Y = 1|X_1 = 1) = 0.44$, Since,

$$p(Y = 1) \neq p(Y = 1|X_1 = 1) \tag{2.11}$$

It follows that X_1 is a primary feature. If the value of X_1 is unknown, then the class Y may be either 0 or 1 with equal probability ($p(Y = 1) = 0.5$). If X_1 is known, then we can guess the class Y with better accuracy than random guessing. If $X_1 = 1$, then Y is most likely to be 0 because $p(Y = 1|X_1 = 1) = 0.44$. If $X_1 = 0$, then Y is most likely to be 1. The feature X_1 is primary because it gives us information about the class Y, even when we know nothing about the other features, X_2 and X_3.

Since $p(Y = y|X_2 = x_2) = p(Y = y)$ for all values y and x_2, it follows that X_2 is not a primary feature. However, X_2 is not an irrelevant feature, since,

$$p(Y = 1|X_1 = 1, X_2 = 1, X_3 = 1) \neq p(Y = 1|X_1 = 1, X_3 = 1). \tag{2.12}$$

Therefore X_2 is a contextual feature. Furthermore, the primary feature X_1 is (strongly) context-sensitive to the contextual feature X_2 , since,

$$p(Y = 1|X_1 = 1, X_2 = 1) = 0.53. \tag{2.13}$$

$$p(Y = 1|X_1 = 1) = 0.44. \tag{2.14}$$

That is, if we know only that $X_1 = 1$, then our best guess is that $Y = 0$ (by Equation 2.14). However, if we know that $X_1 = 1$ in the context of $X_2 = 1$, then our best bet is that $Y = 1$ (by Equation 2.13). The feature X_2 is contextual because it gives us information about the class Y, but only when we know the value of the primary feature X_1. Finally, X_3 is an irrelevant feature, since, for all values y,x_1 ,x_2,and x_3:

$$p(Y = y|X_1 = x_1, X_2 = x_2, X_3 = x_3) = p(Y = y|X_1 = x_1, X_2 = x_2). \tag{2.15}$$

The feature X_3 does not give us any information about the class, even when we know the values of the other features.

2.1.3 Identification of context sensitive features

In general the probability distribution p of instances in the training data set is unknown. So we need to estimate p from the training data. Let D be a sequence of training instances $\langle \vec{X}, Y \rangle$ selected from $F_1 \times F_2 \times \cdots \times F_m \times C$ identically and independently with probability distribution p. Let d be an empirical estimate of p, based on the frequencies of occurrence observed in the training data D (see [16] and its references).

It is likely, due to random variation in D, that every feature X_i will appear to be primary, if we naively apply definition 2.2 to the estimate d. Random noise will cause the following inequality to be true, even when X_i is not actually primary:

$$d(Y = y|X_i = x_i) \neq d(Y = y). \tag{2.16}$$

To apply the above definitions, we need to allow for the presence of noise in the training data D. Let ε be a small positive real number. We may say that the feature X_i appears to be primary when there is a value x_i of X_i and a value y of Y, such that:

$$|d(Y = y|X_i = x_i) - d(Y = y)| > \varepsilon. \tag{2.17}$$

This inequality allows for noise. We can adjust our sensitivity to noise by altering the value of ε. When ε is very close to zero, the implication is that there is little noise in the data. For a fixed sample D, as we increase ε, the number (apparently) of identified primary features decreases. Given a certain desired level of statistical significance (say 95%), we can use standard statistical techniques to calculate the required value of ε.

This concludes that, in addition to the problem of estimating the probability distribution p from the data set D, there is another problem of searching through all possible subsets S_i' of S_i. In general, it is not computationally tractable to examine every possible subset of features in order to determine which features are contextual and which are primary. In practice, it will be necessary to use heuristic search procedures.

2.2 Managing context-sensitive features

In Section 2.1 we have shown the definition of the contextual features and discussed the strategy of identifying these features, the identification of these features is the first main issue with context dependency problem , the other main issue is the management of these identified features. In this section we show the different strategies or heuristics in ([17] and its references).

Assume the standard machine learning framework, where examples are represented as vectors in a multidimensional feature space. We assume that set of training examples is partitioned into a finite set of classes.

As explained earlier, we may distinguish three different types of features: primary, contextual, and irrelevant features.

Primary features are often context-sensitive. That is, they may be useful for classification when considered in isolation, but the learning algorithm may perform even better when we take the contextual features into account.

In this section we introduce a survey by [17] of strategies for taking contextual features into account. We will also list five heuristic strategies for managing context.

We will review evidence that hybrid strategies can perform better than the sum of the component strategies.

Table 2.3 lists some of the examples of contextual features that have been examined in the machine learning literature. Many standard machine learning data sets (Murphy & Aha, 1996) contain contextual features, although this is rarely (explicitly) exploited. For example, in medical diagnosis problems, the patients gender, age, and weight are often available. These features are contextual, since they (typically) do not influence the diagnosis when they are considered in isolation (see [17]).

2.2.1 Strategies for managing context

Suppose we are attempting to distinguish healthy people (class A) from sick people (class B), using an oral thermometer. Context 1 consists of temperature measurements made on people in the morning, after a good sleep. Context 2 consists of

Task	Primary Features	Contextual Features	Reference
image classification	local properties of the images	lighting conditions (bright, dark)	Katz et al. (1990)
speech recognition	sound spectrum information	speakers accent	Pratt et al.(1991)
gas turbine engine diagnosis	thrust, temperature, pressure	weather conditions	Turney&Halas(1993)
speech recognition	sound spectrum information	speakers identity and gender	Kubat(1996)
hepatitis prognosis	medical data	patients age	Turney (1993)
heart disease diagnosis	electrocardiogram data	patients identity	Watrous (1995)
tonal music harmonization	meter, tactus, local key	to be discovered by the learner	Widmer (1996)

Table 2.3: Some examples from machine learning literature

temperature measurements made on people after heavy exercise. Sick people tend to have higher temperatures than healthy people, but exercise also causes higher temperature. When the two contexts are considered separately, diagnosis is relatively simple. If we mix the contexts together, correct diagnosis becomes more difficult. Fig 2.1 illustrates the intuition about this common type of context-sensitivity.

Katz et al. (1990) listed four strategies for using contextual information when classifying. Turney. (1993) named these strategies contextual normalization, contextual expansion, contextual classifier selection, and contextual classification adjustment.

Strategy 1 *Contextual normalization*
Contextual features can be used to normalize context-sensitive primary features, prior to classification. The intent is to process context-sensitive features in a way that reduces their sensitivity to context. For example, we may normalize each feature by subtracting the mean and dividing by the standard deviation, where the mean and deviation are calculated separately for each different context. See Fig 2.2.

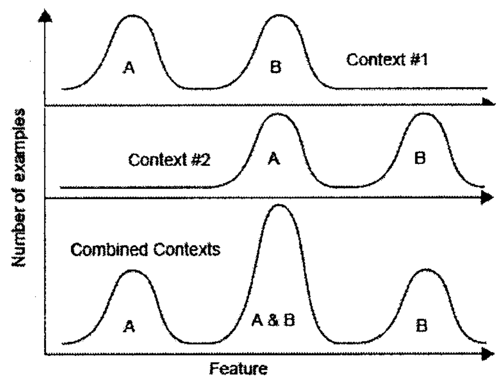


Figure 2.1: The result of combining samples from different contexts.

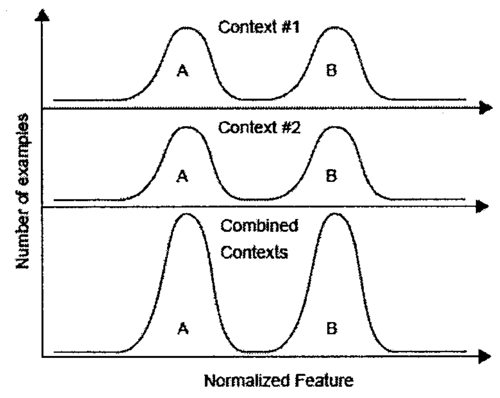


Figure 2.2: *Contextual normalization*: The result of combining normalized samples from different contexts.

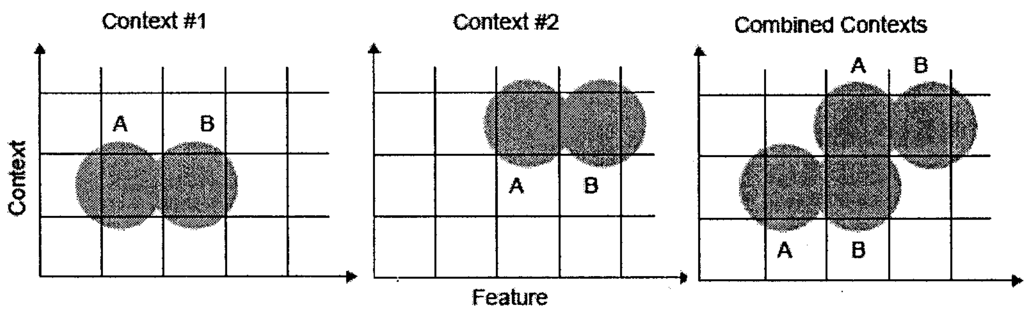


Figure 2.3: *Contextual expansion*: The result of combining expanded samples from different contexts.

Strategy 2 *Contextual expansion*
A feature space composed of primary features can be expanded with contextual

features. The contextual features can be treated by the classifier in the same manner as the primary features. See Fig 2.3.

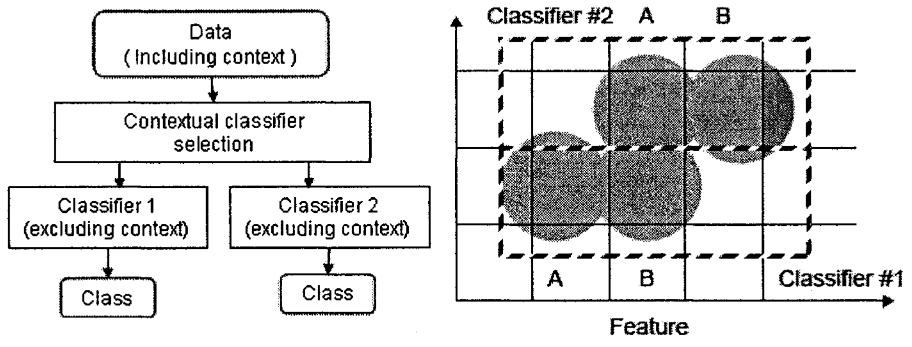

Data
(Including context)
Contextual classifier selection
Classifier 1 (excluding context)
Classifier 2 (excluding context)
Class
Class
Classifier #2 A B
A B Classifier #1
Feature

Figure 2.4: *Contextual classifier selection*: Different classifiers are used in different contexts.

Strategy 3 *Contextual classifier selection*
Classification can proceed in two steps: First select a specialized classifier from a set of classifiers, based on the contextual features, then apply the specialized classifier to the primary features. See Fig 2.4.

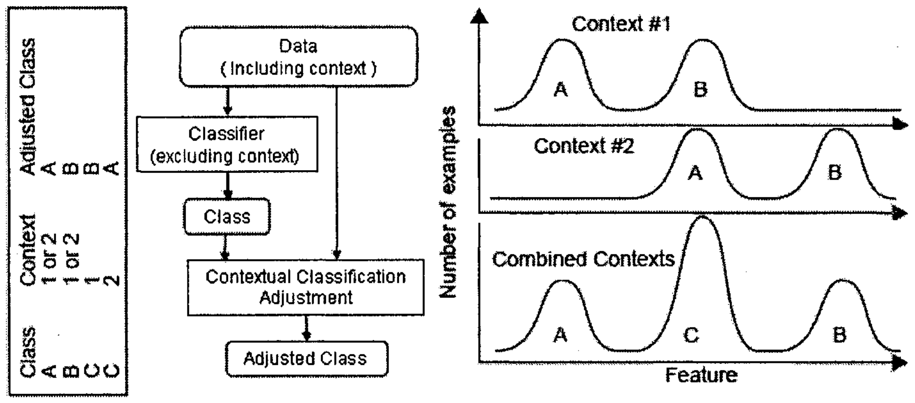


Figure 2.5: *Contextual classification adjustment*: The classification is adjusted for different contexts.

Strategy 4 *Contextual classification adjustment*
The two steps in contextual classifier selection can be reversed: First classify, using only the primary features. Then make an adjustment to the classification, based on the contextual features. The first step (classification using primary features alone) may be done by either a single classifier or multiple classifiers. For example, we might combine multiple specialized classifiers, each trained in a different context. See Fig 2.5.

Turney. (1993) discussed another strategy called contextual weighting.

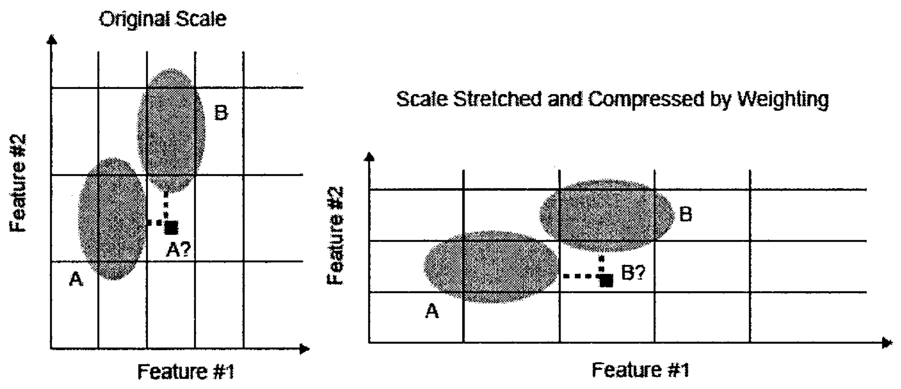


Figure 2.6: *Contextual weighting* : The impact of weighting on classification.

Strategy 5 *Contextual weighting*
The contextual features can be used to weight the primary features, prior to classification. The intent of weighting is to assign more importance to features that, in a given context, are more useful for classification. Contextual selection of features may be viewed as an extreme form of contextual weighting: the selected features are considered important and the remaining features are ignored. See Fig 2.6.

2.2.2 Hybrid strategies

Various combinations of the above strategies are possible. For example, [17] experimented with all eight possible combinations of three of the strategies (contextual normalization, contextual expansion, and contextual weighting) in two different domains, vowel recognition and hepatitis prognosis (Turney 1993a, 1993b). In the vowel recognition task, the accuracy of a nearest neighbor algorithm with no mechanism for handling context was 56%. With contextual normalization, contextual expansion, and contextual weighting, the accuracy of the nearest-neighbor algorithm was 66%. The sum of the improvement for the three strategies used separately was 3%, but the improvement for the three strategies together was 10% (Turney, 1993a, 1993b). There is a statistically significant synergetic effect in this domain. In the hepatitis prognosis task, the accuracy of a nearest neighbor algorithm with no mechanism for handling context was 71%. With contextual normalization, contextual expansion, and contextual weighting, the accuracy of the nearest-neighbor algorithm was 84%. The sum of the improvement for the three strategies used separately was 12%, but the improvement for the three strategies together was 13% (Turney, 1993b). The synergetic effect is not statistically significant in this domain.

2.2.3 Context dependency in neural networks

Broadly speaking, context dependent neural networks means neural networks which can change their way of functioning in a context-sensitive mode. In other words, a context dependent neural network may react differently for the same sequence of inputs, depending on external conditions these conditions is expressed by the contextual variables.

Using context dependency in neural networks is an important issue in many cognitive situations. It opens another horizons to neural network solutions. In the last decade neural network researchers have paid attention to contextual aspects. Many of them have given attention to such aspects to improve systems performance by identifying and managing the context sensitive features among input data. [12] introduced an extension to the standard error back propagation algorithm that enables it to train for the context dependent information by multi-layer feed forward neural networks, a special error function is used in the extension. In [18] a probabilistic framework was presented to incorporate context dependent auditory models in hybrid segment based neural network speech recognition. Some researchers presented new neural network structures to achieve such context dependency. [3] presented a neural approach of using two layer recurrent attractor network which receives external input on one of its layers. Recently, [2] presented a context dependent neural network model. This model allows nets weights to change according to changes of some environmental factors even after completing the learning process.

In this section we have seen different strategies for managing context sensitive features in any learning algorithm. The rest of this chapter will be zooming more into some neural network solutions for learning context sensitive features.

2.3 Overcoming the slow convergence problems

In complex systems systems like for example, robotics and other control systems, learning the control mappings between inputs and outputs requires large size of neural networks. This can make the learning process and convergence prohibitively slow.

The problem size can be approximately quantified as the dimensionality of the input space. With an increase in the dimensionality, the input space will experience an exponential growth in size. Very often, the increase in dimensionality also increases the nonlinearity of the control mappings.

In order to acquire a highly nonlinear control mapping through learning, a large and complex network is required in order to approximate the mapping up to a certain degree of accuracy. The direct consequence of using large networks is the increase in time required to learn the appropriate network parameters using a given learning neural network. Learning becomes so slow that the problem may prove to be intractable.

Context-dependent learning can usually simplify a learning problem significantly. The segmentation of the problem different context makes the segmented subproblems to be reduced to simpler ones under each fixed context.

Of course,the system (natural or artificial) also needs to remember the relationship between the learning subproblem and its associated context.

In [21], Yeung D.Y (1993) had explained the modulation technique of a big problem using the principle of divide-and-conquer to decompose a learning problem to be solved into a set of smaller subproblems corresponding to different contexts. The solutions to the individual subproblems in context-dependent learning are then integrated to give the solution to the entire problem. Decomposing a problem into smaller subproblems might be very expensive and difficult for some problems.

However, the principle of divide-and-conquer is very useful for handling a large variety of problems, especially those whose subproblems do not have very tight mutual interactions.

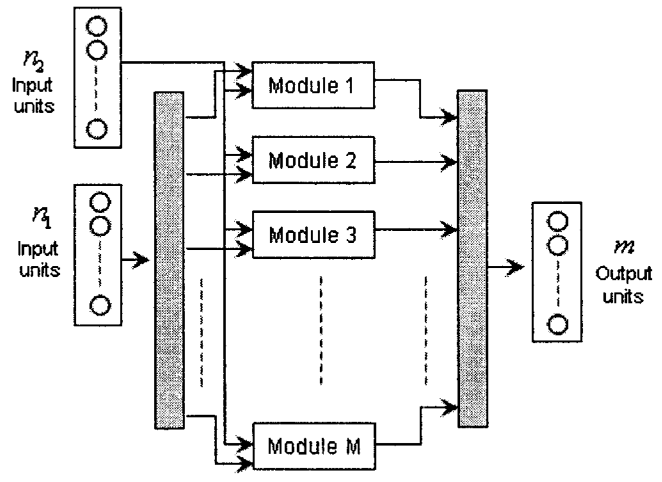


Figure 2.7: A simple neural network model based on modulation to implement the idea of context sensitivity .

Consider now the simple network model with $n = n_1 + n_2$ input units and m output units as shown in Fig 2.7. The set of input units is divided into two groups. The first group consists of n_1 units. The activation values of these n_1 units in combination constitutes an address that can be used to access one of the M modules. If these units have binary (0 or 1) or bipolar (-1 or 1) activation values, a maximum of 2^{n_1} modules can be addressed using the binary coding scheme.

However, these input units can also take continuous activation values, as long as there is some mechanism to guarantee that only one module will get activated. The activated module is then provided with the actual input by the second group of n_2 input units. After some internal processing, the activated module gives an output with m values. One can view this model as a collection of M sub neural network modules, each subneural network has n_2 inputs and m output. The first group of input units serves to specify the context (hence the corresponding subnetwork module) under which further processing is to be carried out. The original problem of forming an $(n \rightarrow m)$ mapping is thus decomposed into M smaller subproblems of forming $(n_2 \rightarrow m)$ mappings. It is important to mention here that the selection of these $n - 1$ inputs for the context network should based on a specific identification of contextual features out of the whole feature space.

This solution we have shown now is simply a modulation of the big problem which is considered the simplest way of applying context dependency, the disadvantage of this technique, is its one-to-one mapping between contexts and subnetwork modules, which probably require huge capacity hardware to process all the different submodules, especially in case of big number of modules that would be impossible to apply this technique.

Yeung D.Y (1993) in [21], developed another enhanced model that mostly doesn't require such huge hardware capacity although in some cases e.g. the very large neural networks, the model may fail to reduce the hardware capacity as shown later, the model is explained in the following subsection.

2.3.1 Context sensitive neural network for problem segmentation

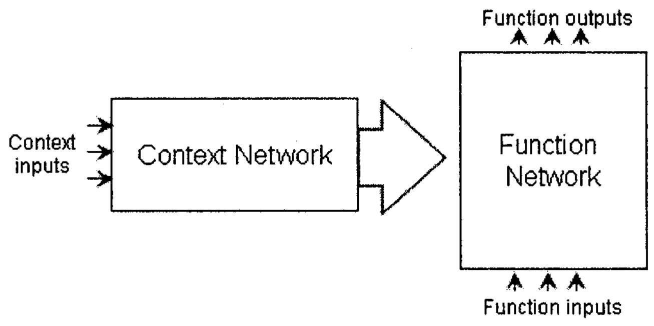


Figure 2.8: Schematic diagram of a context-sensitive network model.

A context-sensitive network is shown schematically in Fig 2.8. It consists of two feed forward neural networks, the context network and the function network. Context network is responsible about mapping the relation between the contextual variables as input and the weight setup of the other neural network as outputs. The function network is responsible to mapping the relation between the set of primary features as inputs and the problem solution as output.

The set of input variables is partitioned into two sets. One set (context input) acts as the input to a context network. The function inputs acts as the inputs to a function network. Depending on the context input provided by the current input pattern, the function network represents different functions at different times based on different contexts.

The feed-forward class of context sensitive neural networks is considered here for both neural networks, with the back-propagation learning neural network.

The context network has as many output units as the total number of adjustable weights in the function network. In general, the number of adjustable weights in the function network may be very large, hence the context network has to learn a large number of parameters. It is thus desirable that the function network be as simple as possible. The ideal case refers to the class of context-sensitive networks whose function networks are linear. This corresponds to the class of functions which are decomposable into parameterized families of linear functions.

Using the activation values of the output units in the context network directly as the weights of the function network is inappropriate, as the activation values are restricted to the range $[-1, 1]$. This problem can be solved by introducing a coupling function to map these activation values from $[-1, 1]$ to their corresponding values which span a wider range. For output unit l in the context network, its activation value, y_l is coupled with a weight in the function network through a coupling function, g_l . One simple choice is to let g_l be the inverse of a sigmoid function,e.g. $g_l = f_l^{-1}$, so that the weights of the function network can take values from $(-\infty, \infty)$.

The use of one network to modulate the behavior of another network is a very useful property. In particular, a single piece of hardware (function network) can behave differently depending on the output of the context network. Hardware reusability is crucial to the design of networks for solving complex problems, so that in most cases the networks will not grow to an unmanageable size. Configuring a system to behave differently in a context dependent manner is a desirable property for robust systems.

With its programmability through the context network, the function network can compute different functions at different times. This helps in separating the network semantically into two different levels of abstraction, each of which plays a different role in network computation. While learning in the function network aims at generalization of the usual type, learning in the context network tries to achieve a better generalization.

This context dependent neural network model have two disadvantages. The context network is trained to produce the weight matrix of the function network , which mean that the output of the context network is usually too large, with a little increase in the size of the function network. The context network increases dramatically. In this case the context network will require extremely long training and probably more hardware resources.

Also the possibility of error mapping of the context network is not small for that large number of outputs , and in this case, selecting wrong weight matrix will let the function network output be too far from the desired. Even if the output of the context network is the proper weight matrix but with some notable error in some weights , that would lead to error in propagating the output of the function network.

In the following subsection we show different technique of imposing the context sensitive features in the neural network solution that doesn't have the mentioned disadvantages while learning complex nonlinear mappings.

2.4 Context dependent neuron model

The model we are showing here also introduces the idea of learning complex nonlinear mappings in a context dependent manner. In the previous model the contextual features were separated from the primary features on a neural network level, i.e. Contextual features were applied to separate neural network other than the neural network that primary features are applied to. In The following model both contextual and primary features are applied with the same neural network. More than that, they are both applied to the same neuron.

Hence, the model of context dependent neural networks we are showing in this section is considered a generalization of the traditional neuron model. The mapping adjustment is performed by contextual "fine-tuning" of weights obtained from a well trained traditional neural networks.

The model of Piotr Ciskowski. (2004) in [2], we are showing here assumes that the features are already identified and doesn't involve any identification technique of the contextual features. It only concentrates on processing contextual data by context dependent neural networks. One of the aims of this models is to cover the case of continuous contextual variables.

2.4.1 Context dependent neuron model structure

Having $\bar{Z}$ denotes the vector of contextual variables, the neuron model shown in Fig 2.9, can be expressed in the form

$$y = \Phi\left[w_0(\bar{Z}) + \sum_{s=1}^{S} W_s(\bar{Z})x_s\right]. \tag{2.18}$$

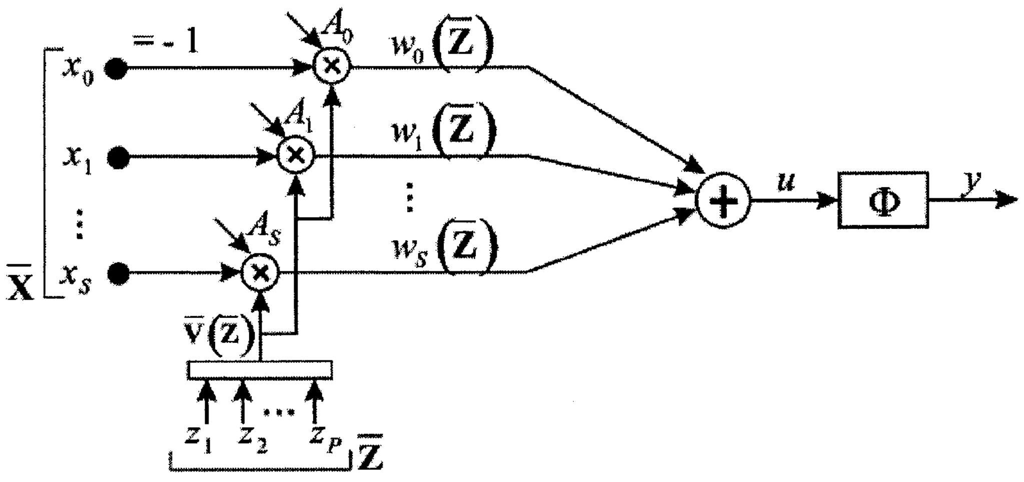


Figure 2.9: Context dependent neuron model

where x_s denotes the s-th primary input, y is the neurons output, and Φ is the activation function.
The first difference between this model and the traditional one is the division of inputs into the two groups of: primary and context-sensitive inputs grouped in vector $\bar{X}$, and contextual inputs in vector $\bar{Z}$. The second difference is that the weights between neurons and primary inputs, depend on the vector of contextual variables, i.e. $w_s = w_s(\bar{Z})$ where $s = 0, 1, \cdots, S$, as presented in Fig 2.9

The neural network model that is used here to solve a problem characterized both by primary and context-sensitive features a hybrid network, in which some weights are context dependent and others are traditional , traditional weight means that the weights are constant after training, i.e. they don't change in all contexts.
In general,$\bar{Z}$ may functionally depend on x_s as being contextual sensitive to them. This model excludes such a possibility, that is, both functional and even stochastic independence of x_s and $\bar{z}$ will be assumed.
In other words by classifying contextual variables into the following categories,

- External contextual variables, which are provided to the network as parallel to input variables, without any functional or statistical dependencies between the context and input variables.
- Internal contextual variables, which are generated from input and/or output signals of the same network.

Then, this model concentrates on the external contextual variables, because its technique of incorporation of the contextual features to the output of the problem, is not depending on such relation between the contextual features among each other.

2.4.2 Dependence of weights on context variables

As mentioned earlier, the second difference between this neuron model and the traditional one is that in this model weights are dependent on the contextual variables. This dependency can be expressed as follows.

Assume that in the model of a context dependent neuron in Equation 2.18, components $w_s(\bar{Z})$ of its weight vector $\bar{W}(\bar{Z})$, dependent on the vector $\bar{Z} = [z_1, z_2, \cdots, z_p]^T$ of contextual variables, in the form,

$$w_s(\bar{Z}) = \bar{A}_s^T \cdot \bar{V}(\bar{Z}), s = 0, 1, \cdots, S. \tag{2.19}$$

where $\bar{V}(\bar{Z}) = [v_1(\bar{Z}), v_2(\bar{Z}), \cdots, v_M(\bar{Z})]^T$ is a vector of basis functions, chosen by the networks designer.

In Equation 2.19 $\bar{A}_s = [a_{s,1}, a_{s,2}, \cdots, a_{s,M}]^T \in R^M$ where $s = 0, 1, \cdots, S$ are vectors of parameters, which specify the dependence of weights on context variables. In other words, components of $\bar{V}(\bar{Z})$ are functions spanning the context dependent vector of weights and our aim is to choose the vectors of coefficients $\bar{A}_s$ where $s = 0, 1, \cdots, S$. As components of $\bar{V}(\bar{Z})$.

2.4.3 Mathematical model of a context dependent net

According to the provided form of dependence of weights on the contextual variables, here we show the mathematical model of the context dependent neuron.

For a random vector of input variables $\bar{X}_{in} = [x_1, x_2, \cdots, x_S]^T$ and for a random vector $\bar{Z}$ of contextual variables, the output of the network is given by

$$Y = \Phi[\bar{W}^T(\bar{Z}) \cdot \bar{X}]. \tag{2.20}$$

where Φ is the activation function, ^{T} is the transposition, $\bar{X} \overset{\text{def}}{=} [1, \bar{X}_{in}]^T$. The k-th neurons weight vector is given by

$$\bar{W}_k(\bar{Z}) = [w_{k,0}(\bar{Z}), w_{k,1}(\bar{Z}), \cdots, w_{k,S}(\bar{Z})]^T. \tag{2.21}$$

where each weight $w_{k,s}(\bar{Z})$ is approximated by the column vector of coefficients $\bar{A}_{k,s}$ and the vector of basis functions $\bar{V}(\bar{Z})$. as $w_{k,s}(\bar{Z}) = \bar{A}_{k,s}^T \cdot \bar{V}(\bar{Z})$.

For the k-th neuron, the coefficient vectors $\bar{A}_{k,s}$ are concatenated into one column vector

$$\bar{A}_k = [\bar{A}_{k,0}^T, \bar{A}_{k,1}^T, \cdots, \bar{A}_{k,S}^T]^T. \tag{2.22}$$

The layers weight matrix is constructed the same way as the weight matrix of a traditional network, only now the weights are dependent on the context vector as shown in 2.23

$$\bar{W}(\bar{Z}) = [\bar{W}_1(\bar{Z}), \bar{W}_2(\bar{Z}), \cdots, \bar{W}_K(\bar{Z})]. \tag{2.23}$$

Φ is assumed to be strictly monotone on R. Thus, $\Psi(t) = \Phi^{-1}(t)$ exists and $\Psi(U) = \Phi^{-1}(U)$ is well-defined random variable, if U is a random variable. The only important activation function, which is excluded by this assumption, is the step function, which however, can be arbitrarily closely approximated by strictly monotone functions.

2.4.4 Minimizing the error of a context dependent Network

The desired output of the net given by Equation 2.20 is represented by a random variable Y. In summary, the existence of a probability distribution of $(\bar{X}, \bar{Z}, Y)$ is assumed. This distribution is unknown. In this learning algorithm we use a learning sequence $(\bar{X}_i, \bar{Z}_i, Y_i), i = 1, 2, \cdots, n$, instead of the unknown common distribution of $(\bar{X}, \bar{Z}, Y)$.

Lets consider that this probability distribution is known, and then we replace the unknown moments of this distribution by their empirical counterparts, based on the learning sequence. Thus, we shall use a variant of the classical method of moments, known to provide efficient estimators when underlying distributions are Gaussian, since then the method of moments coincides with the maximum likelihood approach. Usually, the measure of fit between Y and $\Phi[\bar{w}^T(\bar{Z}) \cdot \bar{X}]$ is considered as

$$E_{(\bar{X},\bar{Z},Y)}\{Y - \Phi[\bar{w}^T(\bar{Z}) \cdot \bar{X}]\}^2. \tag{2.24}$$

where E denotes the expectation with respect to random variables specified below this operator. Here, we consider an alternative approach to choose the weights in such a way that the following criterion is minimized:

$$E_{(\bar{X},\bar{Z},Y)}\left[\Phi^{-1}(Y) - \bar{w}^T(\bar{Z}) \cdot \bar{X}\right]^2. \tag{2.25}$$

which is later called the activation-error criterion, since the desired output Y is transformed back to the interior of the nets' output neuron and then compared with its activation signal $(\bar{w}^T(\bar{Z}) \cdot \bar{X})$.

The advantage of Equation 2.25 in comparison with Equation 2.24, is that Equation 2.25 is a quadratic form with respect to the vector of weights $\bar{W}$. The following result additionally justifies the assumed mathematical model of the cd neuron.

2.4.5 Learning algorithm for feed-forward back-propagation context dependent networks

Piotr Ciskowski in [2] had presented many learning algorithms such as, nonrecursive Least Squares, Recursive Least Squares, Stochastic Approximation, and back-propagation algorithm.

Here we only present his extension to the back-propagation algorithm to train the network for such contextual features.

Hence, in this subsection, a way of calculating the error functions gradient with respect to coefficient vectors of output and hidden layers neurons will be presented. The output-error function will be used, while results for activation-error function are analogical.

Let us now consider a two-layer network containing K neurons in the first, and L neurons in the second layer. $\bar{X}^{(1)} = [1, x_1^{(1)}, x_2^{(1)}, \cdots, x_S^{(1)}]^T$ is the vector of the first layers primary inputs. The first layers outputs $\bar{Y}^{(1)} = [y_1^{(1)}, y_2^{(1)}, \cdots, y_K^{(1)}]^T$, and the bias are primary inputs for the second layer $\bar{X}^{(2)} = [1, y_1^{(1)}, y_2^{(1)}, \cdots, y_K^{(1)}]^T$. $\bar{Y}^{(2)} = \left[[y_1^{(2)}, y_2^{(2)}, \cdots, y_K^{(2)}]^T\right]$ is the vector of the second layers outputs. Both layers are supplied with the same vector of contextual inputs $\bar{Z} = [z_1, z_2, \cdots, z_P]$. Each layer has its weight matrix:

$$\bar{W}^{(1)}(\bar{Z}) = \left[w_{k,s}^{(1)}(\bar{Z})\right]_{(S+1)\times K}. \tag{2.26}$$

where $k = 1, 2, \cdots, K$ and $s = 0, 1, \cdots, S$, and

$$\bar{W}^{(2)}(\bar{Z}) = \left[w_{l,k}^{(2)}(\bar{Z})\right]_{(K+1)\times L}. \tag{2.27}$$

where $l = 1, 2, \cdots, L$ and $k = 0, 1, \cdots, K$.

Let us consider the second layer as the output layer of the network and, thus, assume that the desired output values are given as $\bar{Y}_d^{(2)} = \left[y_{d,1}^{(2)}, y_{d,2}^{(2)}, \cdots, y_{d,L}^{(2)}\right]^T$.

The error function for the network (in [2]) is given by

$$Q = Q^{(2)}\left(A^{(1)}, \bar{A}^{(2)}\right) = \sum_{l=1}^{L} \left[Q_l^{(2)}\left(A^{(1)}, \bar{A}_l^{(2)}\right)\right]. \tag{2.28}$$

where $Q_l^{(2)}\left(A^{(1)}, \bar{A}_l^{(2)}\right)$ is the error function for the second layers l-th neuron, given by

$$Q_l^{(2)}\left(A^{(1)}, \bar{A}_l^{(2)}\right) = \frac{1}{2} E_{(\bar{X}^{(2)}, \bar{Z}, y_{d,l}^{(2)})} \left\{y_{d,l}^{(2)} - \Phi\left(u_l^{(1)}\right)\right\}^2. \tag{2.29}$$

where $u_l^{(2)} = \Phi\left\{(\bar{A}_l^{(2)})^T \cdot [\bar{X}^{(2)} \otimes \bar{V}(\bar{Z})]\right\}$,

and $\bar{Y}^{(1)} = \Phi\left\{(A^{(1)})^T \cdot [\bar{X}^{(1)} \otimes \bar{V}(\bar{Z})]\right\}$ is the vector of the first layers outputs, $\otimes$ denotes the Kronecker product of matrices.

Thus, the error functions gradient with respect to the second layers l-th neurons coefficients is given by

$$grad_{\bar{A}_l^{(2)}} Q = -E_{(\bar{X}^{(2)},\bar{Z},y_{d,l}^{(2)})} \cdot \left\{ d_l^{(2)} \cdot \Phi' \left(u_l^{(2)} \right) \cdot [\bar{X}^{(2)} \otimes \bar{V}(\bar{Z})] \right\}. \tag{2.30}$$

The coefficient update (in [2]) is given by

$$\tilde{\bar{A}}_{n+1} = \tilde{\bar{A}}_n + \gamma_n d_l^{(2)} \cdot \Phi' \left(u_l^{(2)} \right) \cdot [\bar{X}^{(2)} \otimes \bar{V}(\bar{Z})]. \tag{2.31}$$

where $d_l^{(2)} = y_{d,l}^{(2)} - \Phi \left(u_l^{(2)} \right)$

The error functions gradient with respect to the first layers k-th neurons coefficient vector is given by

$$grad_{\bar{A}_k^{(1)}} Q = -E_{(\bar{X}^{(1)},\bar{Z},\bar{Y}_d^{(2)})} \cdot \left\{ \left[\sum_{l=1}^{L} w_{l,k}^{(2)}(\bar{Z}) \cdot \left(y_{d,l}^{(2)} - y_l^{(2)} \right) \cdot \Phi' \left(u_l^{(2)} \right) \right] \right.$$

$$\left. \cdot \Phi' \left(u_k^{(1)} \right) \cdot [\bar{X}^{(1)} \otimes \bar{V}(\bar{Z})] \right\}. \tag{2.32}$$

By analogy to the traditional back-propagation rule, the unknown desired value of the neurons output $y_{d,k}^{(1)}$ may be computed from the errors of all the next layers neurons that are connected to this neuron, and weights connecting the k-th neuron in the first layer with all neurons in the second layer. Thus, the estimated value of the first layers k-th neurons error is given by

$$d_k^{(1)} = y_{d,k}^{(1)} - y_k^{(1)} = \sum_{l=1}^{L} \left[w_{l,k}^{(2)}(\bar{Z}) \cdot d_l^{(2)} \right] = \sum_{l=1}^{L} \left[w_{l,k}^{(2)}(\bar{Z}) \cdot \left(y_{d,l}^{(2)} - y_l^{(2)} \right) \right]. \tag{2.33}$$

where $d_k^{(1)}$ and $d_l^{(2)}$ denote error values for k-th neuron in the first layer and l-th neuron in the second layer, respectively.

Remark 2.1. *Context dependent neural networks are, in fact, a generalization of traditional networks. By choosing the basis function vector with one constant function, we may build a context dependent network with the properties of traditional one.*

2.4.6 Example: XOR problem

It is well known that it is impossible to find any weight vector for a single perceptron neural network (Neural network consists of only one neuron) to solve the standard 4-point XOR problem, in the following subsection we show that with the generalized neuron model provided earlier, it is possible to solve five-points XOR problem (see [2]).

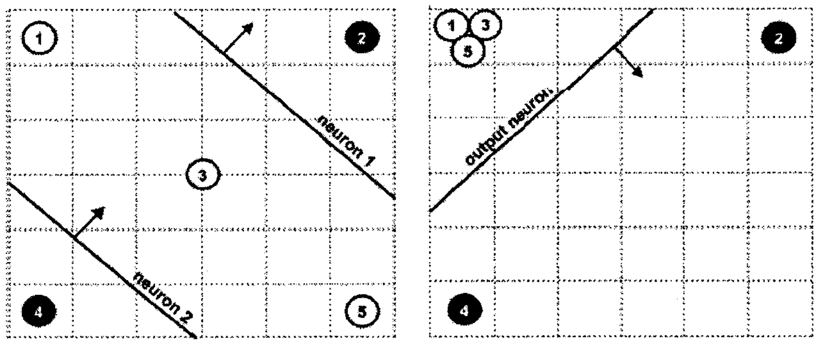


Figure 2.10: Five-point XOR problem solved by a two-layer traditional network. (Left) Input points and discriminating lines of both hidden neurons. (Right) Input points transformed by the first layer and the discriminating line of the output neuron.

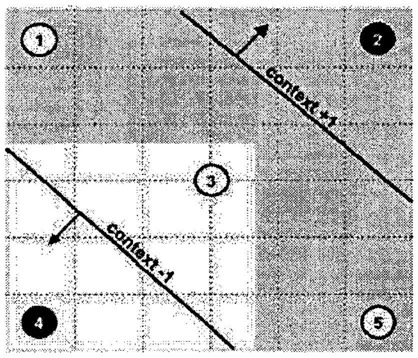
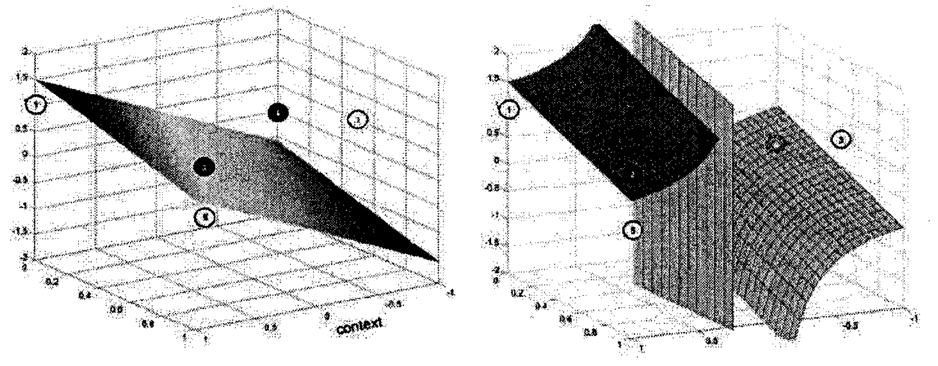


Figure 2.11: Five-point XOR problem solved by one context dependent neuron.

In this subsection, the differences are shown in the way traditional and context dependent networks work. The networks are used to solve the XOR problem enhanced by one additional point in the middle. Traditional networks topology is 2 (Hidden neurons) and 1 (Output neuron). Each neuron in the input layer (not yet "aware" of the points context) is only able to perform linear separation of points, so the division into two classes takes place gradually in both layers.

In Fig 2.10, given five points as inputs : $(0,0)$, $(0,1)$, $(1,0)$, $(0.5, 0.5)$, and $(1,1)$ in two-dimensional input space, the first layer reproduces them into the second layers input space. Three of these points (no. 2, 3, and 4) belonging to the first class are transformed into points close to $(0,1)$. Two other points (from the second class) are positioned in the area that is linearly separable from the transformed points of the first class. Linear separation is then done by the single neuron in the second layer.



(a) Discriminating hyperplane of a traditional functional neuron. (b) Hyper surface of a context dependent neuron.

Figure 2.12: Five-point XOR problem.

Let us now assume that points 1 and 3 belong to one context (for which $z = -1$), while the other three to another $z = 1$. Traditional neuron using the information about points context (supplied on additional primary input or using a functional input multiplying two primary ones) is able to solve the standard 4-point XOR problem (excluding point no. 3). As its decision hyperplane cannot bend through the contexts to perform the desired classification of point no. 3 [Fig 2.12(a)], it cannot solve the enhanced 5-point XOR problem, slightly more complicated than the standard one. Context dependent network (supplied with information about points contexts) works more efficiently.

The 5-point task may be solved by a single neuron. As for each context only one decision line is needed, context dependent neuron produces one discriminating line and adjusts it as the context changes (Fig 2.11). Although lines are parallel, their directions are reversed. Fig 2.12(b) shows the decision hyper-surface of a context dependent neuron in the joint three-dimensional input space and the way it inverses its direction with the context change.

In this chapter we have shown, a formal method to distinguish the three different types of features from the relevance point of view: primary, contextual, and irrelevant features, and an Illustration example was explained.

We have presented the strategy of identifying these context-sensitive features and the five basic strategies for managing them. Combining these strategies appears to be beneficial, as well as five different methods of managing these contextual features.

We have presented a context sensitive model for overcoming the slow convergence problems this technique uses context sensitively between features to provide a segmentation to the problem solution. Another model of context dependent neural nets has been presented and its basic training algorithms, taking advantage of contextual dependencies in training data. It has been shown that context dependent neural networks, being the generalization of traditional networks, have better transformation abilities.

Chapter 3

Overlapped neural networks

Neural network overlapping is known as one of the practical techniques of achieving better generalization and recognition rate. Overlapped Multi-Neural Networks (OMNN) have been used for feed forward neural networks by Hu & Hirasawa. (2000) in [4] and self organized maps by Atukorale & Suganthan. (1999) in [1] and by Suganthan & Winter. (1999) in [14] for this purpose. These systems impose a function distribution over the partially shared weight vector of a multi neural network in which some neurons react only to some inputs but not to the others.

Neural network overlapping is also used in shared weight neural networks (SWNN) in which the weight sharing or overlapping reduces the number of free weights while produces better performance on test sets by Yonggwan & Gader. (1995) in [22] and by Khabou & Gader. (2000) in [7].

Analysis presented by Jinwook & Chulhee.(1999) in [6] and by Stevenson & Winter. (1990) in [13] of the weight distribution and its error sensitivity concluded that the weight vectors of a trained neural network is not unique as there are many possible weight vector solutions based on the initial setup. They also concluded that such weight solutions tend to form concentrated groups in R^N dimensional weight space. This analysis helps to understand the concept of reducing the network freedom by using overlapped neural networks to decrease the neural network function complexity and achieve better generalization.

In an ordinary neural network, individual units do not have any special relations with the input patterns. However, according to recent knowledge of brain science, it is suggested that there exists function localization in a human brain, which means that specific neurons are activated corresponding to certain sorts of sensory information the brain receives. Therefore, a brain-like neural network should have the capabilities of function localization as well as learning. Such a brain-like model may be more efficient because its individual units are mainly used to remember certain input patterns. To obtain such a brain-like model, the main problem is how to guide a training algorithm to realize the function localization.

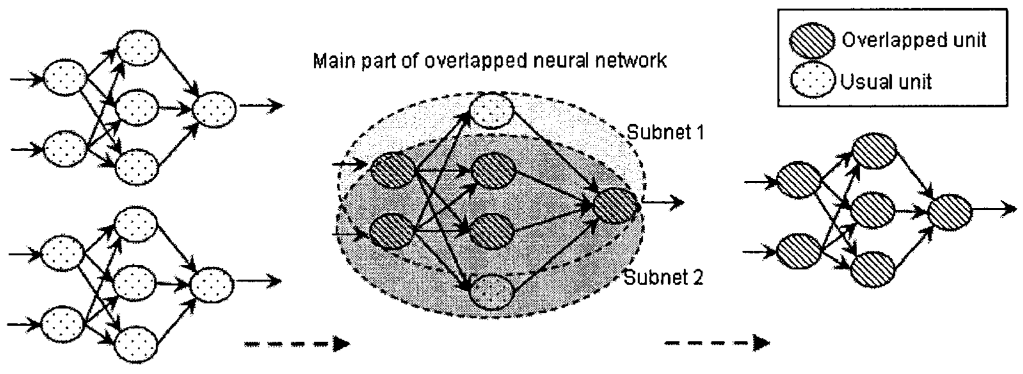


Figure 3.1: Multiple network, overlapped-multi-network and ordinary feed-forward network.

In the following we show an overlapped neural network model by Hu & Hirasawa. (2000)in [4]. In this model, a simple implementation of such brain-like model is considered by using an overlapped multi-neural-network (OMNN). This model OMNN consists of two parts: main part and partitioning part. The main part, structurally, is the same as an ordinary feed forward neural network, but it is considered as one neural network that consists of a class of sub-nets, all the sub-nets have the same input-output units but some different hidden units. The partitioning part, can be any structure that can make unsupervised classification, the main function of this part is to divides the input space into several parts, each of which is associated with one sub-net.

Various clustering or classification methods and algorithm may be used to implement the partitioning part. In his work, Hirasawa has introduced a competitive network for this part in [4] and introduced SOM Model for the partitioning part in [11], However, a competitive network that has the same number of outputs as the number of parts of the input space divided is used here. Each output of competitive network represents one input space part. For an input pattern, only one of competitive network outputs gives 1, and only nodes of the sub-net associated with this output are fired, while all other nodes remain inactive. This realizes function localization of OMNN.

On the other hand, from a viewpoint of multiple network, the main part of OMNN is a multiple network with overlapped units, when the number of overlapped units is zero, it becomes an ordinary multiple network, while it is an ordinary feed-forward neural network when the number of overlapped units is equal to the total hidden units. In this sense, an OMNN can be seen as a learning network between an ordinary feed forward neural network and an ordinary multiple network. Fig 3.1 shows an image of such relationship. Moreover, it is well-known that multi model approach is based on a divide-and-conquer strategy. The bias variance trade-off is an important issue for the divide and-conquer strategy. In general, dividing the training data set into subsets for process identification tends to increase the variance and decrease the bias. In multi model approach, soft or fuzzy splits of data are often used to ease the bias-variance dilemma.

3.1 Overlapped neural network for function localization

An OMNN has the capabilities of both learning and function localization. As shown in Fig 3.2, it consists of two parts: main part and partitioning part. The main part realizes the capability of learning and the partitioning part classifies the input space so as to realize the capability of function localization (see [4] and [11] and their references).

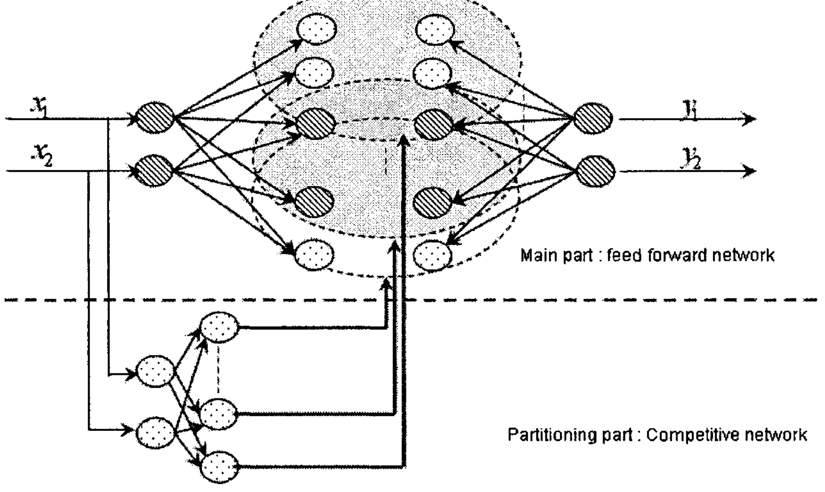

x_1
x_2
y_1
y_2
Main part : feed forward network
Partitioning part : Competitive network

Figure 3.2: An OMNN consisting of two parts: main part and partitioning part.

Partitioning Part

The role of this part is partitioning of operating region. Let us consider problems such as system identification and pattern recognition. The operating region is defined as Z. An operating point $z \in Z$ is a vector of variables. The operating region is partitioned into M operating regimes $Z_i : (i = 1, \cdots, M)$ which is a subset of Z, on the basis of certain prior knowledge.

The input and output vectors of the model are called x and y and consist of n and m different variables respectively, where $x = [x_l, x_2, \cdots, x_n]$ and $Y =$

$[Y_l, Y_2, \cdots, Y_m]$.

Here, competitive neural networks is used for this part. The network selects a winner, via a competitive learning process, highlighting the "winner-take-all" schema. That is, the output unit receiving the largest input is assigned a value of 1, whereas all other units are suppressed to a 0 value. As shown in the lower part of Fig 3.2, the competitive learning network has one layer of input neurons and one layer of output neurons. An input pattern x is a sample point in the n-dimensional real vector space. Binary-valued $\{0, 1\}$ *local representations* are used for the output nodes. That is, there are as many output neurons as the number of classes (M) and each output node represents a pattern category.

Main Part

Structurally, the main part is an ordinary feed forward multi-layer neural network, but it is considered to consist of M overlapped sub-nets.

If we denote the set of input units by I, the set of output units by O, and the set of i-th hidden layer units by $N_i : \{i = 1, 2, \cdots\}$, then the j-th sub-net can be described by $\{I, S_{lj}, S_{2j}, \cdots, O\}$ where the set of i-th hidden layer units, S_{ij} is a subset of N_i. That is, $S_{ij} \subset N_i : \{j = 1, 2, \cdots, M\}$.

These M sub-nets are associated with the M operating regimes (class). For the input and output vectors $\{x, y\}$ of operating regime Z_j, only the units corresponding to the j-th sub-net of OMNN are active, while all other units are inactive and have zero output. The sets of hidden layer units of sub-nets, S_{ij}, are determined based on the prior knowledge used in operating region partition such that all hidden layer units available in the sets N_i are used in subsets S_{ij}.

To establish this structure, there are several parameters to be determined: the number of sub-nets, the number of hidden units for each sub-net, the number of hidden units that overlap or the number of total hidden units.

The values of these parameters are certainly problem depended. Hirasawa. (2000) gives an estimation for these numbers as follows:

1. *The number of sub-nets.* It is equal to the number of parts of input space divided. In many cases, prior knowledge is available for determining the number of input space to be divided. When there is no prior knowledge available, it is recommended to divide the input space into 4 to 6 parts gives better results.

2. *The number of hidden units for each sub-net.* This depends on the complexity of each part of input space. When no prior knowledge is available, the same number may be used for all sub-nets. It is found that when the number of hidden units for each sub-net is equal to $\frac{1}{3}$ to $\frac{2}{3}$ of the total number of hidden units, OMNN gives better results.

3. *The number of hidden units that overlap.* This is rather difficult to determine. It seems that it is easier to determine the total number of hidden units first, then the number of hidden units that overlap by assigning the number of hidden units for each sub-net based on the $\frac{1}{3}$ to $\frac{2}{3}$ rule.

Structure example

Here we give a simple example to show the relation between the partitioning part and the main part in an OMNN.

In the e.g. shown in Fig 3.3, the main part has 2 input units, 6 hidden units and 1 output unit. It is divided into two sub-nets with the same input-output units, the first sub-net contains the first to fourth hidden units, and the second sub-net contains the third to sixth hidden units. The partitioning part is a competitive network containing two output units. It divides the input space into two parts. The outputs of the competitive network control the firing of the hidden units of main part.

When an input set from the first part of input space appears, the competitive network gives $O_1 = 1$ and $O_2 = 0$. This fires hidden units 1 to 4, while hidden units 5 and 6 contribute 0 to the output of OMNN. When an input set is from the second part, then only hidden units 3 to 6 are fired, and hidden units 1 and 2 will be kept inactive. From this example, it is clear that OMNN has not only learning capability, but also function localization capability.

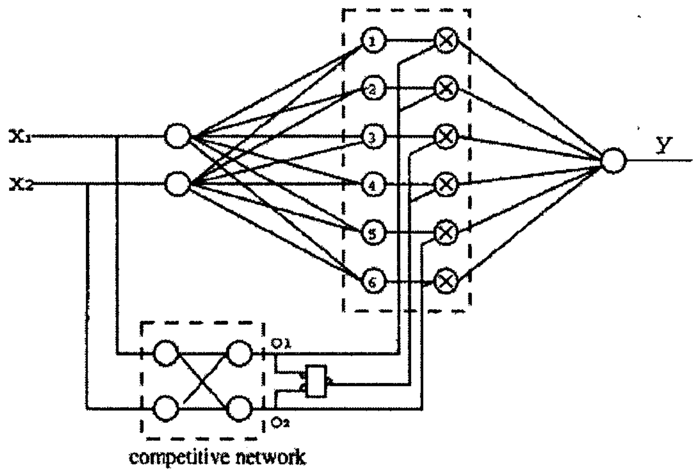


Figure 3.3: An example of OMNN showing the relationship between partitioning part and main part.

3.1.1 Overlapped multi-neural network training

The training process here is two steps: first,training of the partitioning part, second training of the main part. It is clear that the former usually must be done before carrying out the latter training.

A competitive learning algorithm is used for training the partitioning part using the well known winner-take-all algorithm. In the following we discuss training of the main part.

The training of the main part of OMNN is formulated as a nonlinear optimization defined by,

$$\Theta = arg \min_{\Theta}\{E\}, \text{ where } \Theta \in W. \tag{3.1}$$

where E is the error function, Θ is the weight vector and W denotes a compact region of weight vector space. Let y is the OMNN output corresponding to the input vector x. Then the error function E is defined by

$$E = \sum_{i \in D}(\|\hat{y}_i - y_i\|). \tag{3.2}$$

where D is the set of training data, and $\hat{y}$ is the desired output.

Training algorithm

The ordinary random search method algorithm can be employed to find the required weight vector and hence the solution. Some modifications have been done by Hirasawa (2000) to improve the efficiency of the random search algorithm. However, the following is the modified algorithm.

Let $\Theta(k) = [\lambda_1(k), \cdots, \lambda_l(k), \cdots]^T$ be the weight vector $\Theta \in W$ denoting the weight vector corresponding to the k-th search, and $\Delta\Theta(k)$ be the random vector $\Delta\Theta(k) = [\Delta\lambda_1(k), \cdots, \Delta\lambda_l(k), \cdots]^T$ generated based on a probability density functions after the k-th search. Then the random search algorithm can be described as follows.(see [4] and [11])

Algorithm

1. **Step 1:**
    - Choose an initial value $\Theta(0) \in W$
    - calculate $E(\Theta(0))$
    - set $k = 0$.
2. **Step 2:**
    - Generate a random search vector $\Delta\Theta(k)$

- If $\Theta(k) + \Delta\Theta(k) \notin W$ then let $\Theta(k+1) = \Theta(k)$ and go to **Step** 3, else
- calculate $E(\Theta(k)+\Delta\Theta(k))$, If $E(\Theta(k)+\Delta\Theta(k)) < E(\Theta(k))$, (the current search is successful), then set $y^{(k)} = 1$ And $\Theta(k+1) = \Theta(k) + \Delta\Theta(k)$, else
- calculate $E(\Theta(k)-\Delta\Theta(k))$, If $E(\Theta(k)-\Delta\Theta(k)) < E(\Theta(k))$, (the current search is successful), then set $y^{(k)} = 1$ And $\Theta(k+1) = \Theta(k) - \Delta\Theta(k)$, otherwise
- (The search is Failure), Then set $y^{(k)} = 0$ and

$$\Theta(k+1) = \begin{cases} \Theta(k) & \text{if } k_{er}^{+} > k_{er} \text{ and } k_{er}^{-} > k_{er} \\ \Theta(k) + \Delta\Theta(k) & \text{if } k_{er}^{+} < k_{er}^{-} \\ \Theta(k) - \Delta\Theta(k) & \text{if } k_{er}^{+} \geq k_{er}^{-} \end{cases}$$

where $k_{er} \geq 1$ is the maximum error ratio, k_{er}^{+} and k_{er}^{-} are defined by

$$k_{er}^{+} = \frac{E(\Theta(k) + \Delta\Theta(k))}{E(\Theta(k))}$$

$$k_{er}^{-} = \frac{E(\Theta(k) - \Delta\Theta(k))}{E(\Theta(k))}$$

3. **Step 3:**

   Stop if pre-specified conditions are met, else set $k = k + 1$ and go to **Step 2**.

In a conventional random search algorithm, $\Delta\lambda_l$ is usually generated by using a Gaussian probability density function, the modification done by Hirasawa is put better strategy to find $\Delta\lambda_l$ based on a sophisticated probability density function.

Example of OMNN

Hirasawa had considered a benchmark problem. The problem is providing the separability of two nested spirals and he used OMNNs to for that purpose.

The training sets consist of 152 associations formed by assigning the 76 points belonging to each of the nested spirals into two classes. This is a nontrivial classification task, which has been extensively used as a benchmark for evaluation of neural network training. We use the example to discuss generalization ability of OMNN (see [4]).

The OMNN he used in the simulations is denoted by $N_{n-r-m} \times M/n_T$ where N_{n-r-m} is a sub-net with n input units, r hidden units and m output units, M is the number of parts of input space divided, and n_T is the total number of hidden layer units. Since all sub-nets have the same number of hidden units, obviously when $n_T = r \times M$, that is, there is no overlapped units, the OMNN becomes a multi neural network, and when . $n_T = r$ the OMNN is equivalent to an ordinary feed-forward neural network.

As described earlier in chapter 1 the network generalization is depending on three factors: The size of the data , the algorithm complexity, and network structure, and many methods is been proposed for achieving better generalization, we mentioned the weight decay approach for third factor (The network structure), and the VC dimension for the second factor (The algorithm complexity), it is also known that, soft or fuzzy splits of data are often used to this purpose and to ease the bias-variance dilemma. In this example, overlapping hidden units has the same impact so that it improves generalization ability of multi neural network, Hirasawa. (2000).

3.2 Overlapped self organizing maps

In this section we introduce another implementation of neural network overlapping. Suganthan in [14], had developed a model of Hierarchical Overlapped SOM's (HOSOM) for pattern classification, in which the overlapping is achieved by duplicating every training sample to train several upper-level SOM's. That is, the winning neuron as well as a number of runners-up neurons make use of the same training sample to train the higher-level maps grown from those neurons using Kohonen's SOM.

In general every class is represented by several neurons. Further, as every training sample is used to develop a number of different upper layer maps, a degree of overlap in the upper-level SOM's was achieved. This multiplicity allows to make the final decision by fusing the classification of several maps for every training and testing sample. In addition, every higher-level map is trained using a different subset of the training data. As the top-layer maps are pruned by merging and removing neurons, the problem of over-training is curtailed. The resulting HOSOM network offers the best performance for any SOM-based classifier and somewhat comparable performance to the best multi-layer back-propagation network-based classifier. Further, the HOSOM may be regarded as an efficient alternative to the k- nearest neighbor type algorithms.

3.2.1 HOSOM structure

Hierarchical SOM had been intensively researched and is known to provide lower solution cost for the large problems than the standard SOM, Here another feature is added, then the added structural features to the HOSOM relative to the standard SOM, is the the former is both hierarchical and overlapped. However, the network is initialized with just one layer first. The number of neurons in the layer has to be chosen. If there are too few neurons in the first layer, the network may have to be grown to have several layers. If there are too many neurons in the first layer, computational advantage of hierarchical architecture may be compromised. In pattern recognition applications, the number of training samples may be considered in the selection of initial lattice size.

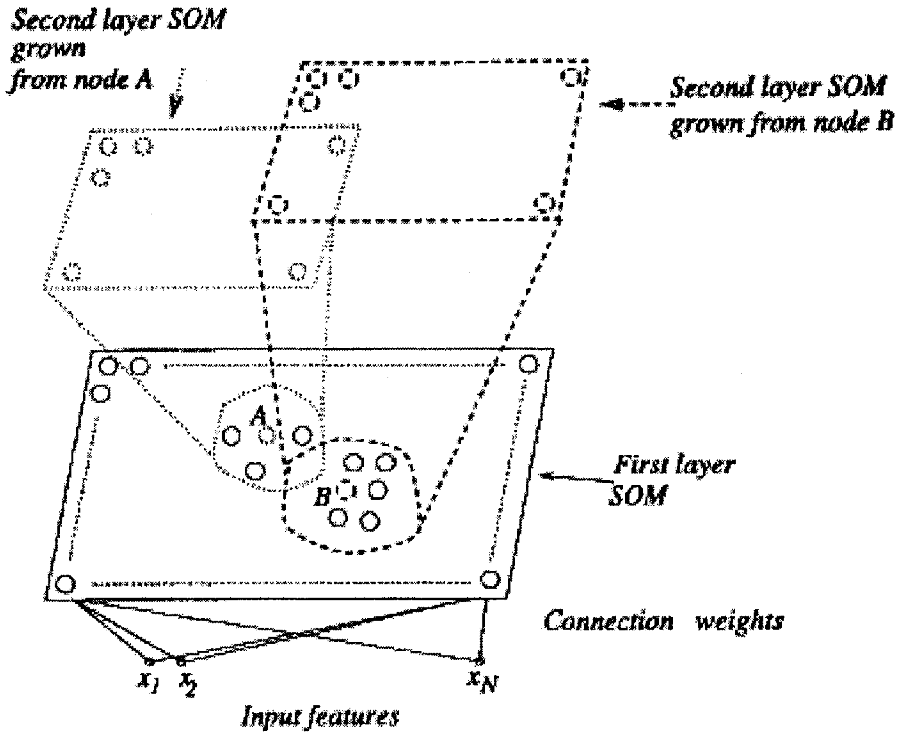

Second layer SOM grown from node A
Second layer SOM grown from node B
A
B
First layer SOM
Connection weights
x_1 x_2 x_N
Input features

Figure 3.4: The HOSOM structure.

Fig 3.4 shows the first-layer SOM and two instances of second-layer SOM's grown out of nodes A and B. The figure also shows the overlap in the feature space of the two second-level maps conceptually.

3.2.2 HOSOM training

For the initial adaptation of the synaptic weight vectors, Kohonen's SOM algorithm is employed. The algorithm is applied to the topmost layers which were grown during the last structure adaptation iteration.

Lets list the following three algorithms, for unsupervised, supervised learning and HOSOM algorithms (see [14] and its references).

Table I: *The unsupervised SOM algorithm*

- **USOM1** Initialize the weight for the given size map. First layer weights are randomly initialized. Subsequent layers are initialized around the root node.

Initialize the learning rate parameter, neighborhood size and set the number of unsupervised learning iterations.

- **USOM2** Present the input feature vector $x = [x_1, x_2, \cdots, x_N]$ in the training data set of the root neuron.
- **USOM3** Determine the winner node c such that $\|x - w_c\| = \min_i\{\|x - w_i\|\}$
- **USOM4** Update the weights within the neighborhood of node c, $N_c(t)$ using the standard update rule: $w_i(t+1) = w_i(t)+\alpha(t)[x_n - w_i(t)]$ where $i \in N - c(t)$. The neighborhood wraps around at edges, i,e., column and row indices are in modulo representation.
- **USOM5** Update learning rate and neighborhood size. According to $\alpha(t + 1) = \alpha(0)\{1 - \frac{t}{K}\}$, and $|N_c(t+1)| = |N_c(0)\{1 - \frac{t}{K}\}|$, where K is a constant and usually set to be equal to the total number of iterations in the self organizing phase.
- **USOM6** Repeat **USOM2-5** for the specified number of unsupervised learning iterations.

Table II: *The supervised LVQ 2 learning algorithm for SOM*

- **SSOM1** Present the input feature vector $x = [x_1, x_2, \cdots, x_N]$ in the training data set.
- **SSOM2** Locate the winner node c such that $\|x - w_c\| = \min_i\{\|x - w_i\|\}$
- **SSOM3** If the winning neuron has the same level as the training example, update weights of the winning neuron only using the standard update rule: $w_c(t + 1) = w_c(t) + \beta[x_n - w_c(t)]$. If the winning neuron has a different label, then 1) update the weights of the winning neuron only using a small negative learning rate β_1 as follows: $w_c(t + 1) = w_c(t) + \beta_1[x_n - w_c(t)]$ and 2) Locate the closest neuron with the same label as the training sample and update its weights using the update equation with a positive learning rate of β_2.
- **SSOM4** Repeat **SSOM1-3** for the specified number of supervised learning iterations.

Table III: *The HOSOM algorithm*

- **HOSOM1** Apply USOM
- **HOSOM2** Label all output nodes using a simple voting scheme.
- **HOSOM3** Apply SSOM

- **HOSOM4** Merge/remove neurons
- **HOSOM5** Apply SSOM
- **HOSOM6** Obtain recognition rates on training data.
- **HOSOM7** Grow an additional layer and repeat **HOSOM 1-6** until satisfactory recognition rate is achieved or maximum complexity level is reached.

The SOM algorithm is summarized in Table I. Having completed the unsupervised SOM learning, the neurons in the topmost layers are labelled using a simple voting mechanism. Then the supervised LVQ algorithm given in Table II is applied to fine-tune the prototype vectors. We apply the following structure adaptation techniques just after applying the supervised LVQ 2 algorithm.

1. *Growing a Layer* : The network may be grown, until either a satisfactory recognition rate is achieved or a predefined level of structural complexity is reached by the network. The complexity may be defined in terms of number neurons or layers.
2. *Merging/Removing Neurons* : The merging operation is essential in particular in the final layer which is not to be grown further.

   Consider the following simple scheme. If an end-node neuron represents a few training samples, that neuron is merged with another neuron which is the closest with the same label. If there is no other neuron with the same label, the neuron is removed. It should be noted that merging operation improves the performance on test data set, in case the network has been over-trained or over-specialized, Kohonen,1997.

It was indicated earlier that overlapped SOM's are obtained at the completion of training. The overlapping is achieved by duplicating every training sample to train several upper-level SOM's. That is, the winning neuron as well as a number of runners-up neurons make use of the same training sample to train the higher-level maps grown from those neurons. By duplicating the training samples in the upper-level SOM's, we obtain overlapped SOM's. The testing samples are also duplicated, but to a lesser degree. Hence, the testing samples fit well inside the feature maps developed using the best matching and several runners-up in the training data.

In addition, this duplication of samples allows us to employ a voting scheme to obtain the final classification. The HOSOM algorithm for pattern recognition is presented in Table III. Fig 3.4 shows the first-layer SOM and two instances of second-layer SOM's grown out of nodes A and B. The figure also shows the overlap in the feature space of the two second-level maps conceptually.

3.3 Shared weight neural networks

In this section we just give a brief about the shared-weight neural networks, many researchers had used shared-weight neural networks for the problem of automatic target detection for its ability to improve the network generalization, for example by Yonggwan & Gader. (1995) in [22] and by Khabou & Gader. (2000) in [7].

It was mentioned earlier that many techniques exist to improve the generalization capability of a neural network by imposing predefined constraints on its weights. Such techniques include regularization, weight pruning and structural constraints. Regularization uses an added term to the neural-network cost function to reduce the effect of non useful weights or to impose a priori knowledge on its structure. Pruning eliminates weights that are deemed redundant.

Structural constraints reduce the number of independent weights by using a locally connected structure or sharing the same weights on many connections. This dramatically reduces the number of free weights while producing better performance on test sets. The standard shared-weight neural network (SSNN) , the morphological shared-weight neural network (MSNN) and the Entropy Optimized Shared-Weight Neural Networks (ESNN) are examples of such networks.

3.3.1 Shared-weight neural network architecture

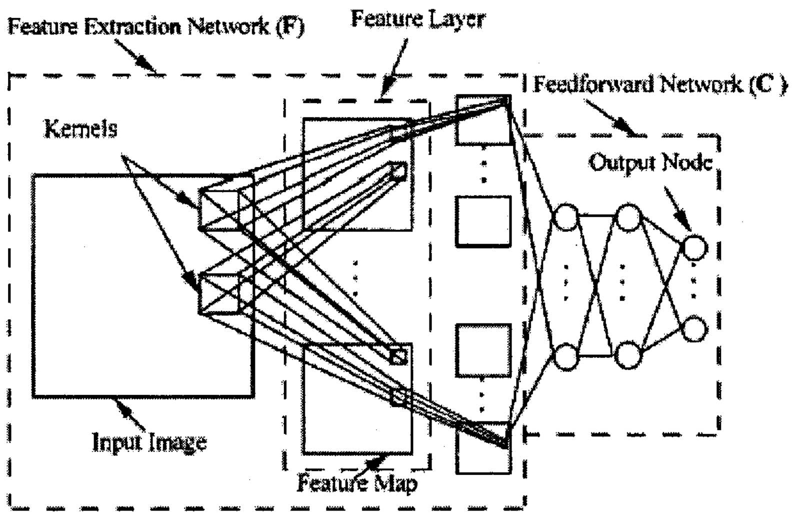


Figure 3.5: Standard shared-weight neural network architecture.

The structure of SSNN is not far from the structure of the previous mentioned OMNN, a SSNN consists of two cascaded sub-networks, called stages: a feature extraction stage F followed by a feed-forward stage C. The feature extraction stage

F usually has a two-dimensional input and has local, translation invariant connections. The layers in this stage perform feature extraction by linear convolution of their inputs with the kernels defined by the local connections.

Each layer is partitioned into subsets called feature maps. Each feature map in a feature extraction layer has one kernel for each feature map in the previous layer. The nodes of the last feature extraction layer are the inputs to C (see Fig 3.5).

The morphological shared-weight neural network, MSNN has the same architecture as the SSNN except that the kernels in the feature extraction layers perform gray-scale hit-miss transform on their inputs instead of convolution performed by the SSNN. For ESNN, Entropy defines a measure on the space of probability distributions, such that those of high entropy are in some sense favored over others.

During training in automatic target recognition, a shared-weight neural network takes a sub image as input, and produces two output values: target or nontarget.

Chapter 4

Novel context sensitive model

In the previous chapters, we have seen the context dependency usage in different models and the benefits that we may gain by employing them in a neural network solution, the major advantages of employing contextual features is actually depending on the way it is used, one of the advantages we have shown here is simplifying the problem while providing more efficiency to it, we also have described neural network overlapping, the major advantages of overlapped neural network, that it performs a function localization and improves the neural network generalization ability.

In this chapter we introduce a novel context dependent model for solving complex problems regardless of sufficiency or accuracy of their historical observations or lab simulation data.

Our approach in this model is based on imposing a context of the problem performance metrics into networks and gaining the enhancement towards its satisfactory state.

We use an overlapped system of back propagation neural networks for our purpose. A main neural network is responsible for mapping input and output relation while a regulatory neural network evaluates the performance metrics satisfaction.

We provide special training and testing algorithms for the overlapped system that guarantees a synchronized solution for both neural networks. By the end of this chapter we present an example of traffic control problem. The result of simulation shows a great enhancement of the solution using our approach.

Context Dependent Controller by Overlapped Neural Networks for Performance Metrics Revision

4.1 Introduction

The theory and design of artificial neural networks have advanced significantly during the past 20 years. Lots of attentions were given to it for its efficient capabilities in pattern recognition, classification, regression, decision making and other tasks of information processing. Many progresses are focused on establishing new techniques and designs of network structures to increase the ability and the efficiency of solving complex problems and to add new features to neural networks.

In this chapter, we consider context dependent neural networks using a different model. Our model is based on a neural structure in which we control and benefit from the weight distribution of two overlapped neural networks to allow imposing the context dependent features into the final output. Different from the result of [2], our contextual variables are not independent on input variables in general. On the other hand, the contextual variables are not necessary generated from input and/or output of the same network. Therefore our approach is also different from the model of Elman's network. These characteristics of our new model allows some special applications which are not fit the previous models.

Analysis in [6] and [13] of the weight distribution and its error sensitivity concluded that the weight vectors of a trained neural network is not unique as there are many possible weight vector solutions based on the initial setup. They also concluded that such weight solutions tend to form concentrated groups in R^N dimensional weight space. This analysis helps to understand the concept of reducing the network freedom by using overlapped neural networks to decrease the neural network function complexity and achieve better generalization.

Our model adapted some advantages of overlapped multi-neural networks for the purpose of context dependency.

The rest of this chapter is organized as follows. Section 4.2.1 describes our concept of applying performance metrics as a context dependency and give an example of a problem analysis according to this concept. In section 4.2.2 we explain the model structure and define the function of its components. In section 4.2.3 the output and the error computation based on the definition of SPM state vector and the provided structure is shown. Section 4.4 and 4.2.5 provide the training and SPM incorporation algorithms. Section 4.3 shows a simulation example and its results and finally, section 4.4 is our conclusion.

4.2 Model and structure of our network

In this section, we describe a new model of context dependent neural networks.

4.2.1 Performance metrics state vector as context dependency

Most of the problems (simple or complex) have a set of dependent performance metrics which are related to each other and dependant on the inputs and solution algorithm. Each performance metric works as a meter to the performance of the solution algorithm, and has a value or a set of values that constitutes the optimal or satisfactory states.

In our model we employ a criterion of context dependent parameters that is derived from some environmental observations of the performance metrics. It will be used to enforce the neural network to produce an optimal output towards the Satisfying Performance Metrics (SPM) state vector.

The study of this performance metrics satisfaction is very important in establishing an efficient and integral solution of problems in which an optimum satisfaction of the dependent performance metrics is required.

This study is also important for cases when the historical data of a problem can't provide the best consultations that guarantees the requested quality.

Example of that can be a mortgage calculation, where the historical data might seem good (case by case) but it doesn't provide the maximum benefits for all the parties. In this problem, the study of the performance metrics as a context dependency would be able to re-evaluate the historical data from the performance point of view and redefine the decision boundaries to achieve better estimations.

Performance metrics incorporation can also play a good role in fixing the lack of the historical information. It will help in providing enhanced solutions in spite of the lack of information.

Neural solutions that the training data is based on a laboratory or experimental data or that is based on a software computation might also use our technique to overcome possible human mistakes or insufficient study of the problem.

Definition 4.1. Satisfactory Performance Metrics (SPM) *: In this model we define SPM as the state vector of the performance metrics that describes the satisfactory states of these metrics.*

Each factor or dimension of the SPM state vector defines the requested quality of a specific performance metric. We can view it as a magnet which creates an attraction force to let the neural network output towards or against its value. For

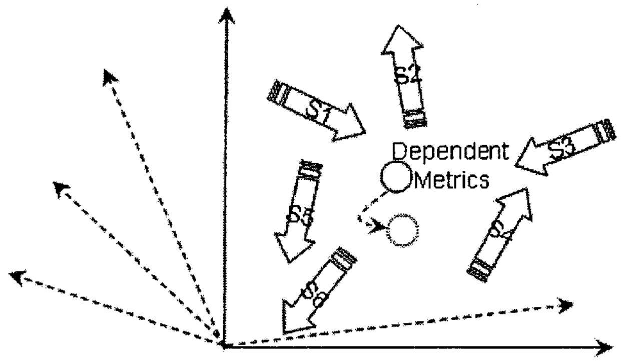


Figure 4.1: In R^N dimensional space, every SPM factor pushes or pulls the output vector towards its satisfactory state as possible.

all the SPM factors, the resultant force would be the enhancement to the solution towards satisfying that performance metrics.

SPM state vector is different from the neural network input and output vectors. It defines a subset of the problem dependent performance metrics.

To explain the SPM, let's give an example of the bandwidth allocation problem for a new network connection request in ATM Networks. Let

Inputs

- (i_1) Network node buffer size
- (i_2) Total available bandwidth
- (i_3) current network traffic conditions
- (i_4) Requested connection quality

Output

- (o_1) Allocated Bandwidth

In this example, the performance metrics based on the total occupied bandwidth after such allocation are

Dependent Performance Metrics

- (m_1) The cell loss rate (CLR).
- (m_2) The cell delay variation (CDV).
- (m_3) The network resources utilization.

It is clear that these metrics (m_1) through (m_3) are all depending on the decision of the bandwidth allocation algorithm as well as the current traffic conditions or the current inputs.

Now we can identify the SPM for this performance metrics as the best possible combination of these metrics as follows.

SPM state vector

- (s_1) Minimum possible cell loss rate. (Min m_1)
- (s_2) Minimum possible cell delay variation. (Min m_2)
- (s_3) Maximum possible network resources utilization. (Max m_3)

And we can define SPM for this problem as the state vector

$$SPM = \langle s_1, s_2, s_3 \rangle = \langle 0, 0, \max(bandwidth) \rangle.$$

In our model, employing the SPM state vector will force the neural network to obtain the possible best output towards satisfying all the metrics based on the defined state vector.

If we employ a specific SPM state vector that gives more attention to a specific performance metric than the other or that ignores some performance metrics, then the neural network will be forced to give more attention to these specific metrics than the others.

In the previous example, more attention is given to cell loss rate than cell delay variation in text data transfer, while giving more attention to cell delay variation than cell Loss rate in voice transfer.

Definition 4.2. Partial SPM : *We define a partial SPM state vector as a SPM vector that ignores one or more performance metrics.*

For example,

$$\langle s_1, s_2, s_3 \rangle = \langle 0, -, \max(bandwidth) \rangle.$$

is a partial SPM state vector that doesn't give any attention to the second metric.

In some cases it might be impossible for the network to reach the SPM state, because the complex relationship between these metrics makes satisfying one metric is against the other. In our example, satisfying the cell loss rate (m_1) and the cell delay variation (m_2) will be against the complete satisfaction of the network resource utilization (m_3). This is because achieving maximum ATM network utilization (m_3) means increasing the network traffic and the allocated bandwidth, which increases the cell loss rate (m_1) and cell delay variation (m_2).

4.2.2 Neural network structure

Our model consists of two back propagation neural networks, partially overlapped in the hidden layers and sharing the same input layer. The output layer of one neural Network plays two roles as it acts also as a part of a hidden layer of the other neural network.

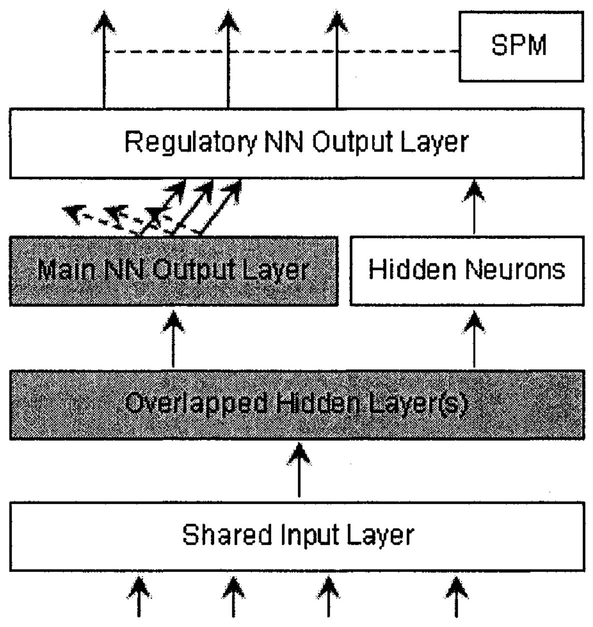


Figure 4.2: Model Structure, two overlapped back-propagation neural networks.

Main Neural Network (MNN)

This is a 3(+) layers back propagation neural network with one or more hidden layers, this neural network is responsible for learning the relation between the inputs and the outputs of the problem. This neural network belongs entirely to the other bigger neural network called regulatory neural network (RNN).

Training of this neural network will be performed in parallel with the training of RNN as shown later. The output of this neural network will be pumped again to the regulatory neural network as shown in Figure 4.2.

Regulatory Neural Network (RNN)

This is a 4(+) layers back propagation neural network with two or more hidden layers. It is responsible for learning the relation between the inputs and the performance metrics of the historical observations, and to impose the SPM state vector into the output of the main neural network.

It includes all the hidden layers of MNN as well as the output layer which constitutes part of the top level hidden layer of RNN.

Advantages of the model

Since MNN entirely belongs to RNN, all of its weights are actually shared by the two neural networks. That will limit its weight freedom which leads to better generalization of the unseen data by providing less error for each simulation, comparing to the totally free weights neural networks.

RNN, as well, will have less number of free weights as big part of its weights are shared with MNN which also leads to less error decisions.

The combination of RNN and MNN will control the weight distribution over the whole system, and then we can benefit from that weight distribution to impose an external context into the behavior of MNN in an efficient way.

Simplicity of the model structure which is based on back propagation algorithms makes it fast and easy to be implemented.

4.2.3 SPM and SSE

For a specific problem, we can express its historical data as a set of patterns in the form $\langle X, Y, M\rangle$ where $X = \langle x_1, x_2, \cdots, x_n\rangle$ is the input vector, $Y = \langle y_1, y_2, \cdots, y_p\rangle$ is its output vector, and $M = \langle m_1, m_2, \cdots, m_q\rangle$ is the observed performance metrics based on the input vector X and its decision or output Y. SPM defined as the vector $S = \langle s_1, s_2, \cdots, s_q\rangle$ is the state of the performance metrics which we are looking forward to achieve.

Then we can express the sum square error (SSE) between the output of RNN and SPM as

$$E_{M,S} = \frac{1}{2}\sum_{j=1}^{D}\sum_{k=1}^{q}(m_{kj} - s_{kj})^2. \tag{4.1}$$

Where D is the number of training patterns,q is the number of neurons in the output layer of RNN.

RNN being well trained means that it is able to provide an output that is close enough to its desired output.

So we can take the limit over the error equation in (4.1) when $M \overset{Training}{\longrightarrow} \hat{M}$, M is the set of actual outputs, $\hat{M}$ is the set of desired outputs. Then we get

$$\lim_{M \longrightarrow \hat{M}} E_{M,S} = E_{\hat{M},S}. \tag{4.2}$$

From (4.2) it is clear that for a trained neural network the error to the SPM state vector is tending to the error between the desired outputs and this SPM state vector. For a finite set of training data and a constant state vector the error $E_{\hat{M},S}$ is constant. For infinite set of training data or in the online training, $E_{\hat{M},S}$ would be the SSE of $\hat{M}$ to S.

In our model we expect to decrease the effect of this error over the incoming decisions by employing a set of virtual data patterns $\langle X, Y, S \rangle$ in which the output of RNN is always the same vector S. It is clear that we can't use this virtual set of data patterns to train the neural network, because it will result in mapping a relation from any vector to the constant vector S. In our model we use this virtual set of data patterns to test RNN for new input vectors in a very sensitive way in which we don't allow the weight vector of RNN to change dramatically by the training of such virtual data patterns. In this technique we always take a step back to the set of weights that provided the best mapping of $\langle X, M \rangle$.

This restoration of the weight vector protects the system from getting distorted. On the other hand the output Y of MNN will be produced according to the imposing of the vector S.

The output of a neuron i in the output layer of RNN is

$$m_i = f^s \left(\sum_{j=1}^{H} h_j w_{ji} \right). \tag{4.3}$$

Where H is the number of neurons in the topmost hidden layer of RNN, h_j is the output of neuron j in this hidden layer, w_{ji} is the connection weight between neurons j and i, and f^s is the output sigmoid function.

Since the topmost hidden layer consists of two parts (as the output layer of MNN belongs to it), we can rewrite (4.3) as

$$m_i = f^s \left(\sum_{j=1}^{p} y_j w_{ji} + \sum_{k=p+1}^{H} h_k w_{ki} \right), \tag{4.4}$$

where $y_j = h_j \; \forall j \in \{1, 2, ..., p\}$ and p is the number of neurons in the output layer of MNN. Then we can write

$$m_i = f^s \left(G_1(Y, W^N) + G_2(h^N, W^N) \right), \tag{4.5}$$

where G_1, G_2 are functions expressing the two summations in (4.4) respectively, h^N is the set outputs from the non overlapped hidden neurons of RNN topmost layer, and W^N is the non overlapped part of RNN weight vector.

Since h^N is propagated from preceding overlapped layers, we can make the following substitution.

$$m_i = f^s \left(G_1(Y, W^N) + G_2(X, W^O, W^N) \right), \tag{4.6}$$

where W^O is the overlapped part of RNN weight vector. According to (4.6) we can see that G_1, G_2 indicate the dependency of the performance metrics over the input X and the decision Y. And that dependency was established using the provided model weight redistribution.

If the number of the non overlapped neurons in the topmost layer is small enough, then the dependency function G_1 will be incorporated more than G_2 in producing the final output of RNN. Hence Y as the output of MNN will be more adapted towards satisfaction of the performance metrics during the error back propagation of RNN.

4.2.4 Training algorithm

Training of this overlapped system using back propagation algorithms has to be done in parallel. We will minimize the training of both neural networks and try to get a reasonable balance between them. Extra consecutive training for any one of them will certainly result in a distortion of the weight vector of the neural networks.

The ideal case is when every training iteration for one neural network is followed by a training iteration for the other net. Consecutive training over the whole system in such a manner will force the weight vector of the RNN to perform a specific distribution in which the overlapped section will plot the relation between X and Y, while the whole weight vector is also able to plot the relation between X and M.

Since it is not guaranteed that both neural networks provide an acceptable solution at the same time, we implemented the following technique to adjust the consecutive training so that allowing a neural network perform more training than the other.

We trigger the system to take such decision when one neural network reaches a good convergence state while the other one still not. In this case the slow neural network will have the opportunity to be trained more while the other one will wait for it or probably perform a little divergence until they both be in a same solution phase. If the system couldn't come to a solution for both neural networks, then more neurons should be added to the non overlapped and/or overlapped part of RNN.

Before introducing the training algorithm, lets define R_{RNN} as the training rate of RNN, and R_{MNN} as the training rate of MNN. Also let $R_{Max} = Max(R_{RNN}, R_{MNN})$ and $R_{Min} = Min(R_{RNN}, R_{MNN})$. The training rate is the number of training iterations for one training cycle, where the training cycle equals to R_{Max}. Define E^D_{RNN} to be the SSE of RNN, and similarly E^D_{MNN} to be the error for MNN.

Algorithm

1. Start
2. Set $R_{RNN} = R_{MNN} = 1$.
3. Initialize RNN with random weights.

4. Pick a random training pattern $\langle X, Y, M \rangle$.
5. If $R_{RNN} \geq R_{MNN}$ or R_{RNN} $mod(R_{Max} - R_{Min}) \neq 0$ then train RNN for $\langle X, M \rangle$.
6. If $R_{MNN} \geq R_{RNN}$ or R_{MNN} $mod(R_{Max} - R_{Min}) \neq 0$ then train MNN for $\langle X, Y \rangle$.
7. Repeat (4) to (6) for (R_{Max}) times.
8. Repeat (4) to (7) for (D/R_{Max}) times.
9. Repeat (4) to (8) for a number of epochs.
10. If (E^D_{RNN} is acceptable but E^D_{MNN} is not acceptable) then increase R_{MNN}, Else if (E^D_{RNN} is not acceptable and E^D_{MNN} is acceptable) then increase R_{RNN}.
11. If $(R_{MNN} = R_{RNN})$ then Set $R_{RNN} = R_{MNN} = 1$.
12. Repeat (4) to (11) until E^D_{RNN} is acceptable and E^D_{MNN} is acceptable.
13. End

It is known that the error function for the output layer neurons in back propagation is different from the error function of the hidden neurons.

According to the provided training algorithm, the first hidden layer(s) will be treated as hidden layer in both neural networks, while this is not the case for MNN output layer as it will be trained for error as an output layer in MNN using the learning equation

$$\Delta w_{ij} = -\eta(\hat{y}_j - o_j)o_j(1 - o_j)o_i.$$

where $\hat{y}_j$ is the desired output for neuron j in the output layer of MNN, While get trained for error as a hidden layer in RNN using the learning equation

$$\Delta w_{ij} = -\eta \left(\sum_{k=1}^{q} -\delta_k w_{jk} \right) o_j(1 - o_j)o_i.$$

Where Δw_{ij} is the change of the connection weight between neuron i and j according to the back propagation algorithm, η is the training factor, o_k is the output of neuron k in the upper layer, δ_k is the output error of neuron k in the upper layer.

4.2.5 SPM incorporation algorithm

Lets define a degree of dissatisfaction σ_k^{SPM} for the output of a neuron k in RNN output layer as follows

$$\sigma_k^{SPM} = \begin{cases} s_k - m_k & \text{if } s_k \neq \text{``}-\text{''} \\ 0 & \text{otherwise} \end{cases}$$

In case of partial SPM, σ_k^{SPM} will equal to 0 for all ignored metrics and will equal to the error between the neuron output and the requested satisfaction if specified.

The following simple test algorithm is able to provide the output of MNN according to the required satisfaction criterion. The regulatory neural network is first consulted to estimate the performance metrics of such input data and then evaluate it against the SPM state vector. The resulted error will be back propagated in the whole system to allow the main neural network to provide a suitable solution for such satisfaction.

Algorithm

1. Start.
2. Backup the weight vector of RNN.
3. Test RNN for the given input.
4. Compute the degree of dissatisfaction σ_k^{SPM} for all output neurons of RNN.
5. Back propagate the degree of dissatisfaction to RNN.
6. Test MNN for the same given input.
7. Restore the weight vector of RNN.
8. Repeat step (3) to (7) for all the test data.
9. End.

Steps (2) and (7) are used to backup the weight vector of RNN and restore it again after the testing is done. The need for that is because our testing includes a step of error back propagation for RNN. If we ignore its commutation that would cause divergence of the MNN.

4.3 A Simulation Example

Street traffic control is chosen as an example for its well known structure. In this example, we want to control the intersection flow using information of the previous 3 traffic cycles.

Training data were obtained from a software program (Traffic flow controller) developed using C++ language based on real traffic rules. Traffic flow control and the performance metrics were obtained based on setup data such as intersection structures, average vehicle speed and dimensions, etc.

For simplicity we simulate two intersected streets constituting four traffic directions. Each street has three lanes and the middle lane is for left turning (see Figure 4.3).

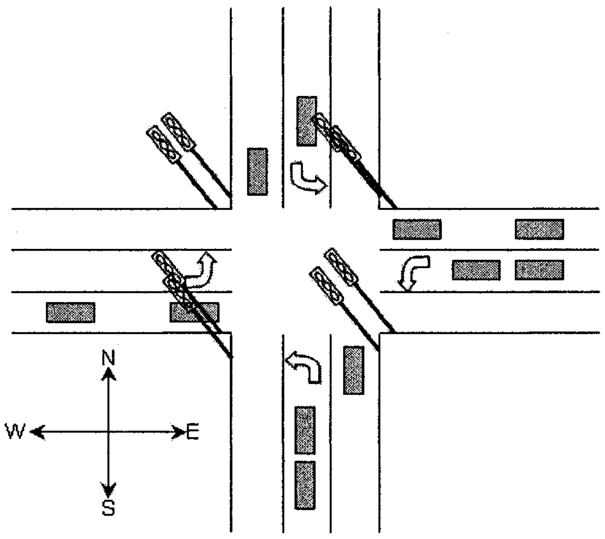


Figure 4.3: An intersection of two streets with three lanes each, the middle lane is left turn lane.

We denoted the four directions by ⟨ North, South, East, West ⟩, or simply ⟨ N, S, E, W ⟩. Traffic Light has five statuses ⟨ Red, Yellow, Green, Left Arrow, flashing Green ⟩. *A traffic load* is an ordered set of numbers, $\langle N_{NS}, N_{NL}, \cdots, N_{WL} \rangle$, where N_{NS} is the number of vehicles heading north straight and N_{NL} is the number of vehicles heading north left, and so on.

A vector of the intersection traffic lights at any moment is ⟨ N Light, S Light, E Light, W Light ⟩. For example, the traffic light state vectors for the Intersection ⟨ Green, Green, Red, Red ⟩ and ⟨ Red, Red, Red, Red ⟩ are valid vectors, while ⟨ Green, Green, Green, Green ⟩ is not a valid state vector. There are 14 valid state vectors in a traffic cycle. Each vector is associated with a *time interval* which indicates the duration of that state vector. After that time interval, the traffic lights changes to the next state vector.

A solution for a given traffic load is a vector of ordered time intervals corresponding to each of the 14 traffic light state vectors, which constitutes a complete traffic cycle. A Traffic cycle is set as 120 seconds.

The performance metrics in this example are cumulative numbers of vehicles which were delayed for at least one traffic cycle, while crossing the intersection. So the *performance metrics vector* $M = \langle D_{NS}, D_{NL}, \cdots, D_{WL}\rangle$, where D_{NS} is the number of delayed vehicles for North-Straight direction , and so on.

4.3.1 Neural network design

The main neural network has three layers of $\langle 32, 32, 14\rangle$ neurons consisting 1472 connection weights between them, with an output sigmoid function.

The 32-input vector describes the last three cycles traffic loads (24 numbers) and the numbers of current waiting vehicles. The 14-output vector is the sequence of the traffic light state time intervals, which is a solution estimated for the incoming traffic cycle, based on previous three cycles traffic load.

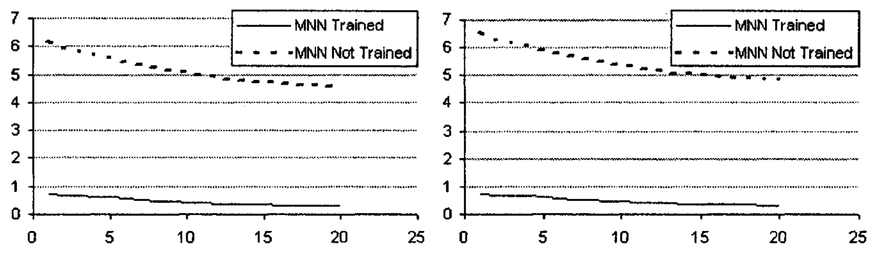
The Regulatory neural network has four layers of $\langle 32, 32, 32, 8\rangle$ neurons consisting 2304 connection weight between them, with an output sigmoid function.

The 32-input vector is similar to that of the main neural network while the 8-output vector is the ordered set of the estimated performance metrics vector defined earlier.

It is clear that RNN is sharing about 2 thirds of its weight vector with MNN.

4.3.2 Simulation results

This example uses a set of 10,000 data patterns $\langle X, Y, M\rangle$ which are from the C++ program. 5,700 of them were used as training set while 4,300 were used for testing as unseen set of data. The neural networks were trained for 1,140,000 iterations. SSE (Sum square error) between the neural network outputs and the desired outputs was read for training and testing for both neural networks.

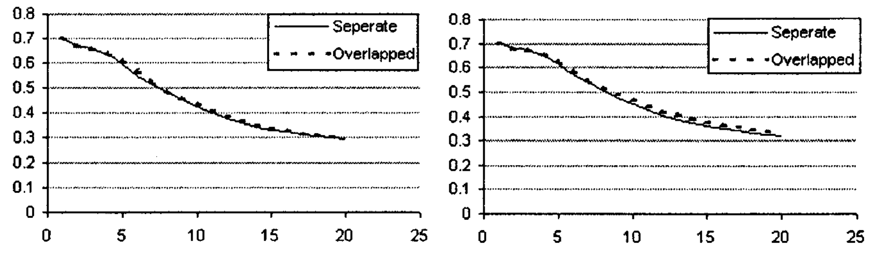


(a) Training (b) Testing

Figure 4.4: SSE of MNN sharing all weights with training of both neural networks (Solid), only training RNN (Dashed)[X-axis is the training, Y-axis is the error].

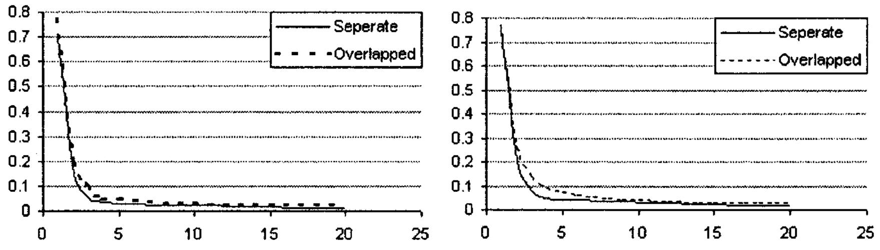
The error on the Y-axis of Figures 4.4, 4.5 and 4.6 was computed as the SSE of the actual and the desired output.

In Figure 4.4(a) and 4.4(b), we compared the results from different settings. The dashed line shows the error of MNN which is a part of RNN without any training, its weights change only according to the training of RNN. In this case its output layer was only acting as a part of the hidden layer of RNN. Training RNN certainly affects the weight vector of MNN but doesn't produce any specific weight distribution. The error for training set in Figure 4.4(a) and test set in Figure 4.4(b) respectively is very big. The neurons in the output layer of MNN is only trained as hidden neurons in RNN. The Training algorithm provided allows them to provide solutions together as well as provide the requested weight distribution. The solid line shows the error when both RNN and MNN gets trained according to our training algorithm.



(a) Training (b) Testing

Figure 4.5: SSE of MNN sharing no weights (Solid), sharing all weights with RNN (Dashed) [X-axis is the training, Y-axis is the error].



(a) Training (b) Testing

Figure 4.6: SSE of RNN sharing no weights (Solid), partially sharing weights with MNN (Dashed)[X-axis is the training, Y-axis is the error].

In Figures 4.5(a) and 4.5(b), we tested the effect of the network overlap. We considered two situations for the system , separate and overlapped. In the separate status there is no weight sharing, i.e. MNN is totally isolated from RNN and has its own different set of weights. In the overlapped status MNN shares all its weight vector with RNN according to our model structure. The solid line shows the error of MNN in separate status and the dashed line line shows the error in overlapped status. It clearly shows that MNN provided the requested convergence although it shares all its weights with RNN. So overlapped system reduces the number of free weights without reducing the function of the system.

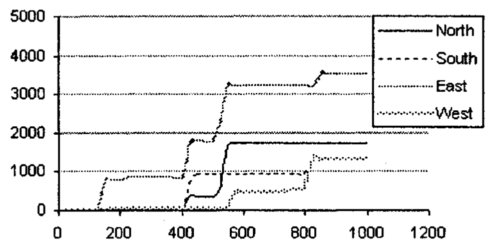
Figures 4.6(a) and 4.6(b), shows the error of RNN in the separate and overlapped situations in the solid and dashed lines respectively, we see that RNN also provided the requested convergence in the overlapped status.

In this case we see that the overlapped system is able to provide the performance metrics as the output of RNN, while being able to provide a solution to the given inputs from MNN output layer which is also part of RNN topmost hidden layer. We forced the weight change of RNN to perform such specific weight distribution that allows us to impose the SPM state vector over MNN behavior.

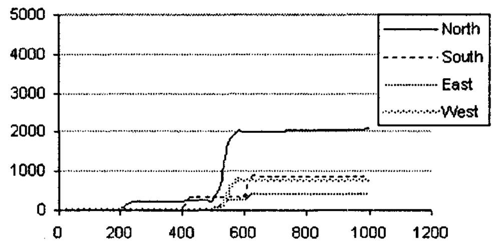
After establishing the requested weight distribution a test for the overlapped system reaction to different SPM state vectors was done over a test set of 1000 unseen traffic cycles which forms 33 hours in terms of time. Testing of the 1000 unseen patterns was done by the provided SPM incorporation algorithm. The traffic light state time intervals was obtained from MNN as its output.

According to the obtained MNN output and the intersection traffic load we computed the cumulative number of delayed vehicles using the traffic flow controller software. In this way we can measure the effect of imposing the SPM state vector on the traffic performance of the intersection.

For simplicity we express the number of cumulative delayed vehicles as 4 dimensional vector. We take our measures for the four straight directions only.



(a) No SPM incorporation



(b) SPM = $\langle 0,0,0,0 \rangle$

Figure 4.7: Cumulative delayed vehicles over the time resulted by traffic computations over MNN output, [X-axis is the time, Y-axis is the number of cumulative delayed vehicles.].

Figure 4.7(a) shows the performance metrics over 33 hours without any SPM state vector incorporation. The measure of the cumulative delayed vehicles in the four directions was ⟨ 1718, 1323, 3526, 1320 ⟩ for ⟨ Left, South, East, West ⟩ respectively,

Figure 4.7(b) shows the incorporation of a SPM state vector $\langle 0,0,0,0 \rangle$,the measure of the cumulative delayed vehicles over the 33 hours was ⟨ 2074 , 867 , 415 , 748 ⟩. It is clear that the system provided a great enhancement of the traffic flow over the 33 hours by decreasing the cumulative delayed in the south, east and west directions, with a little increase in the north direction. This is considered the best performance metrics over the whole traffic system in these conditions.

The $\langle 0,0,0,0\rangle$ SPM state vector acted like a set of magnets, each one of these magnets attracts the output of the main neural network towards the satisfaction of the corresponding performance metric separately.

The total force of attraction of these magnets could achieve such reduction of the cumulative delayed vehicles in the south, east and west directions. It also caused a little increase in the north direction.

The total enhancement in the four directions i,e. the summation of all the cumulative delayed vehicles has been reduced from 7887 to 4104 delayed vehicle in a period of 33 hours.

According to the training data of MNN, it was trained to give more attention to the directions that are loaded with traffic. For example if the traffic load for north direction is expected to be high, MNN will produce a set of traffic light intervals for the next cycle in which more time will be assigned to the north-south Green light.

We can see that the overlapped system has two sets of forces that affects the final output of MNN. These forces are solution force and the regulation force.

The solution force is produced from MNN training and its tendency to provide the desired solution. In our example, this force works to free the loaded traffic in any direction without paying attention to specific directions.

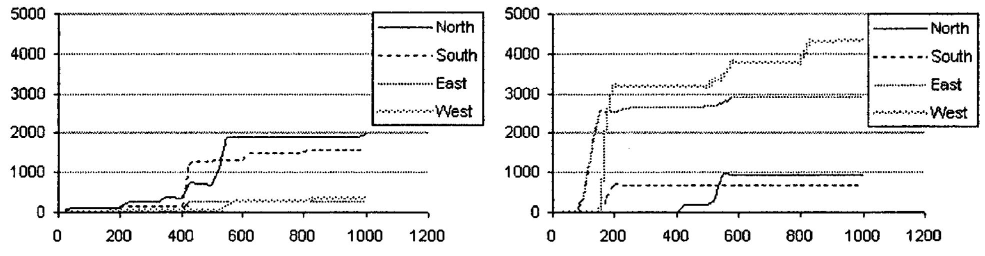
The regulation force is produced from the SPM incorporation to the output of MNN. In our example, it works to free the traffic of the four directions according to the requested satisfaction of the SPM state vector without paying attention to which one has more traffic load.

Hence we get the output of MNN that is balanced using the solution force towards the desired solution and the regulation force towards the SPM state vector.

In Figure 4.8 we are testing the system reaction to the partial SPM state vectors. In Figure 4.8(a) we gave priority to free the east and west directions by choosing SPM state vector $= \langle -,-,0,0\rangle$ and in Figure 4.8(b) we gave priority to free the north and south directions by choosing SPM state vector $= \langle 0,0,-,-\rangle$.

The results of imposing both partial SPM state vectors show an excellent reaction from the system, and we could obtain the requested prioritization over the selected traffic directions.

Another set of results based on different SPM state vector is provided in the following table.



(a) SPM = $\langle -, -, 0, 0 \rangle$ (b) SPM = $\langle 0, 0, -, - \rangle$

Figure 4.8: Cumulative delayed vehicles over the time resulted by traffic computations over MNN output using partial SPM, [X-axis is the time, Y-axis is the number of cumulative delayed vehicles.].

SPM state vector	Result Performance metrics
No SPM	⟨ 1718 , 1323 , 3526 , 1320 ⟩
$\langle 0,0,0,0 \rangle$	⟨ 2074 , 867 , 415 , 748 ⟩
$\langle 0,0,-,- \rangle$	⟨ 944 , 678 , 2923 , 4314 ⟩
$\langle -,-,0,0 \rangle$	⟨ 1971 , 1560 , 261 , 346 ⟩
$\langle -,-,-,0 \rangle$	⟨ 1454 , 1169 , 2037 , 399 ⟩
$\langle -,-,0,- \rangle$	⟨ 2561 , 2255 , 1000 , 6 ⟩
$\langle 0,-,0,- \rangle$	⟨ 1567 , 2546 , 772 , 386 ⟩
$\langle 0,-,-,0 \rangle$	⟨ 942 , 517 , 3411 , 223 ⟩
$\langle 0,-,0,0 \rangle$	⟨ 2355 , 3081 , 712 , 145 ⟩
$\langle 0,10,0,0 \rangle$	⟨ 2216 , 3671 , 1062 , 41 ⟩

Table 4.1: Performance metrics after imposing different SPM state vectors

4.4 Conclusion

Our novel approach of solving problems is based on satisfying its dependent performance metrics. This approach allows us to provide an integral and efficient solution regardless of sufficiency or accuracy of its historical observations or lab simulation data.

We measure these dependent performance metrics of the problem and perform regulation to the neural network output to gain better satisfactory to these performance metrics.

We employed a state vector (SPM) of the problem to express the required problem satisfaction and we used it to enhance the final solution of the neural network. The partial SPM vector also can be used to control the solution behavior by prioritizing the satisfaction of some specific metrics over the others.

Our model consists of two overlapped back-propagation neural networks that possesses a specific weight distribution among their weight vectors. The first is the main neural network which is responsible for mapping the input to output. The main neural network belongs entirely to the second neural network. The second network called regulatory neural network is responsible for evaluating the performance metrics satisfaction and imposing the degree of dissatisfaction to the output of the main neural network.

Training of such system of overlapped multi neural networks required a specific algorithm that prevents extra consecutive training cycles to any of them and let them provide solutions in the same time. In this algorithm we trigger the slow neural network to train more than the faster one and also keep flipping training cycles between them.

Our algorithm for incorporating SPM state vector to the final solution is based on error back-propagation in which we let the regulatory neural network estimate the performance metrics for the unseen inputs, evaluates the degree of dissatisfaction and then back propagate it through the overlapped neurons. It allows the main neural network benet from the such regulation and provide better solution towards the required satisfaction.

Simulation results of a traffic system show that the overlapped system of neural networks could establish convergence to their desired solutions while the main neural network is sharing its entire weight vector with the regulatory neural network. The results also shows that the system provided a great enhancement to the solution by incorporating SPM state vector so that the cumulative delayed vehicles for all directions had decreased dramatically during a test period of 33 hours. We could control the behavior of the solution and give higher priority of freeing some traffic directions than others by incorporating partial SPM state vector that expresses the required satisfactory state.

Chapter 5

Conclusion

Context dependent neural networks mean neural networks which can change their way of functioning in a context-sensitive mode. Using context dependency in neural networks is an important issue in many cognitive situations. In this report we introduced a novel context dependent neural network model based on overlapped multi-neural network structure. For that we gave details about contextual features and some of its applications in neural networks. We also presented some different strategies for applying overlapping in neural networks.

First, we started by introducing the basic knowledge and concepts of neural networks, and explained its main constituting components. The most common neuron model is McCulloch and Pitts neuron. Many different neuron output formulation functions were listed. The sigmoidal function is the most common one. We explained the multi-layer perceptron (MLP) as the most common structure of neural networks. The well-known method of back-propagation learning was also briefly presented.

The generalization ability of a neural network determines how well the mapping surface of the network will renderer the unseen inputs to the output space. Generalization is mainly influenced by three factors: the number and performance of the learning data samples, the complexity of the learning algorithm employed, and the network size. We have shown the weight decay as one the approaches may be obtained for the complexity measure. We briefly touched upon Vapnik Chervonenkis (VC) theory as a concept of complexity measure that can be captured.

We highlighted the radial basis function (RBF) networks and self organizing maps (SOM). Radial basis function networks can be regarded as a very useful addition to the toolbox of neural networks. In general, it can be seen that for many applications, RBF networks can provide a fast and accurate means of approximating a nonlinear mapping based on observed data. Due to the locally acting nature of RBFs, they have a tendency to require more data than a comparable multi-layer sigmoidal network.

Our study of the contextual features involved a formal method to distinguish the three different types of features from the relevance point of view: primary, contextual, and irrelevant features. Primary features are useful for classification when considered in isolation without regard for the other features. Contextual features are not useful in isolation, but can be useful when combined with other features. Irrelevant features are not useful for classification, either when considered alone or when combined with other features. An illustration example was explained.

Researches that involve contextual features are mainly concerned with two issues. The first issue is identifying such contextual features among the whole feature space of a problem. The second issue is managing these contextual features, in which researchers are concentrating on developing different techniques of managing these features and benefit from them.

We have presented the strategy of identifying these context-sensitive features and five basic strategies for managing them. Combining these strategies appears to be beneficial.

We have presented a context sensitive model for overcoming the slow convergence problems, this technique uses context sensitively between features to provide a segmentation to the problem solution.

A context dependent neuron model was also presented. This neuron is considered a generalization of the traditional neuron, according to the mapping adjustment which is performed by contextual "fine-tuning" of weights obtained from traditional networks. The mathematical model and the learning algorithm of this context dependent neural networks was presented. A sample five-point XOR problem was presented as well.

Neural network overlapping is one of the practical techniques of achieving better generalization and recognition rate. It is been used in feed-forward neural networks as well as in self organizing maps. It has also been used in shared weight neural networks (SWNN) in which the weight sharing or overlapping reduces the number of free weights while produces better performance on test sets. Examples of overlapped multi feed forward neural networks to perform function localization was presented, and a benchmark problem example had shown better generalization based on the function localization performed using the overlapping technique.

We have introduced a novel approach of solving problems that is based on satisfying its dependent performance metrics. This approach allows us to provide an integral and efficient solution regardless of sufficiency or accuracy of its historical observations or lab simulation data.

In our approach we measure the dependent performance metrics of the problem and perform regulation to the neural network output to gain better satisfactory to these performance metrics. For that we employed a state vector (SPM) that stands for (Satisfactory performance metrics) of the problem to express the required problem satisfaction and we used it to enhance the final solution of the neural network. We also introduced a partial SPM vector which also can be used to control the solution behavior by providing a control function of prioritizing the satisfaction of some specific metrics over the others.

The novel model consists of two overlapped back-propagation neural networks that possess a specific weight distribution among their weight vectors. The first is the main neural network which is responsible for mapping the input to output. The main neural network belongs entirely to the second neural network. The second network called regulatory neural network. And it is responsible for evaluating the performance metrics satisfaction and imposing the degree of dissatisfaction to the output of the main neural network.

Training of the system of overlapped multi neural networks required a specific algorithm that prevents extra consecutive training cycles to any of them and let them provide solutions in the same time. In this algorithm we trigger the slow neural network to train more than the faster one and also keep flipping training cycles between them.

We presented a test algorithm that allows incorporating the SPM state vector to the final solution. And it is based on the error back-propagation algorithm. We let the regulatory neural network estimate the performance metrics for the unseen inputs and then evaluates the degree of dissatisfaction and back propagate it through the overlapped neurons. This allows the main neural network to benefit from such regulation and provide better solution towards the required satisfaction.

Our simulation results of a traffic system show that the overlapped system of neural networks could establish convergence to their desired solutions while the main neural network is sharing its entire weight vector with the regulatory neural network. The results also shows that the system provided a great enhancement to the solution by incorporating SPM state vector. So that the cumulative delayed vehicles for all directions had decreased dramatically during a test period of 33 traffic hours. We could control the behavior of the solution and give higher priority of freeing some traffic directions than others by incorporating partial SPM state vector that expresses the required satisfactory state.

References

[1] Atukorale A.S and Suganthan P.N, "Combining multiple HONG networks for recognizing unconstrained handwritten numerals," IEEE Neural Networks, 1999. IJCNN '99. International Joint Conference, vol 4, pp: 2928 - 2933, 10 July 1999.

[2] Ciskowski P and Rafajlowicz E, "Context-Dependent Neural networks-Structures and Learning," IEEE Trans. Neural Networks, vol 15, pp: 1367-1377, 6, Nov 2004.

[3] Doboli S, Minai A and Best P.J, "Generating smooth context-dependent neural representations," IEEE Neural Networks, 1999. IJCNN '99. International Joint Conference, vol 1, pp: 12 - 15, 10 July 1999.

[4] Hu J and Hirasawa K, "Overlapped multi-neural-network: a case study, " IEEE Neural Networks, 2000. IJCNN 2000, Proceedings of the IEEE-INNS-ENNS International Joint Conference, vol 1, pp: 120 - 125, 24 July 2000.

[5] Hu J, Hirasawa K and Xiong Q, "Overlapped Multi- Neural-Network and Its Training Algorithm" IEEJ Japan, 2001. Electronics, Information and Systems Society, vol 121, pp.1949-1956, Dec 2001.

[6] Jinwook G and Chulhee L, "Analyzing weight distribution of neural networks," IEEE Neural Networks, 1999. IJCNN '99. International Joint Conference vol 2, pp: 1154 - 1157, 10 July 1999.

[7] Khabou M.A and Gader P.D, "Automatic target detection using entropy optimized shared-weight neural networks," IEEE Trans. Neural Networks, vol 11, pp: 186 - 193, 1 Jan 2000.

[8] Kohonen T,"Self- Organizing Maps," Springer, pp:1 - 260, 2001.

[9] Madan M.G , Liang j and Noriyasu H "Static and Dynamic Neural networks from fundementals to advanced theory," IEEE Press pp:1 - 751, 2003.

[10] Michie D., Spiegelhalter D.J., Taylor C.C. "Machine Learning, Neural and Statistical Classification," University Forvie, pp:1 - 274, 1994.

[11] Sasakawa T, Hu J and Hirasawa K, "Self-Organized Function Localization Neural Network" Fuji Technology Press, Japan, The 20th Fuzzy Systems Symposium, vol 1, pp. 27-30, Dec 2004.

[12] Spreeuwers L.J, Van Der Zwaag B.J and Van Der Heijden F,"Context dependent learning in neural networks," IEEE Fifth International Conference, pp: 632 - 636, 4 Jul 1995.

[13] Stevenson M, Winter R and Widrow B, "Sensitivity of feed-forward neural networks to weight errors," IEEE Trans. Neural Networks, vol 1, pp: 71 - 80, March 1990.

[14] Suganthan P.N, "Hierarchical Overlapped SOM's for Pattern Classification," IEEE Trans. Neural Networks, vol 10, pp: 193 - 196, 1 Jan 1999.

[15] Tibshirani R and Hinton G, "Coaching variables for regression and classification," Statist. Comput, vol 8, pp: 2533, 1998.

[16] Turney P, The identification of context-sensitive features:Aformal definition of context for concept learning, in Proc. 13th Int. Conf. Machine Learning (ICML96), Bari, Italy, 1996, pp: 5359

[17] Turney P, The management of context-sensitive features: A review of strategies, in Proc. 13th Int. Conf. Machine Learning (ICML96), Bari, Italy, 1996, pp. 6066.

[18] Verhasselt J and Martens J.P, "Context modeling in hybrid segment-based/neural network recognition systems," Acoustics, Speech, and Signal Processing, ICASSP '98. IEEE, vol 1, pp:501 - 504, 12 May 1998.

[19] Vojislav K , "Learning and Soft Computing," The MIT Press pp:1 - 576, 2001.

[20] Wrobel A, Kubik E and Musial P, Gating of the sensory activity within barrel cortex of the awake rat, Exp. Brain Res, vol 123, pp: 117123, 1998.

[21] Yeung D.Y and Bekey G, "On reducing learning time in context-dependent mappings," IEEE Trans. Neural Networks, vol 4, pp: 3142, Jan 1993.

[22] Yonggwan W and Gader P.D, "Morphological shared-weight neural network for pattern classification and automatic target detection," Neural Networks, 1995. Proceedings., IEEE International Conference, vol 4, pp: 2134 - 2138, 1 Dec 1995.

[23] Yu H,H and Jenq-Neng H "Handbook of Neural network signal processing," CRC Press pp:1 - 384, 2002.

Appendix

```
// This the C++ Source code of the simulation example in this report
// Example : Traffic Controller

#include "stdafx.h"
#include <stdlib.h>
#include <stdio.h>
#include <math.h>

#define Min_Double     -HUGE_VAL
#define Max_Double     +HUGE_VAL
#define MIN(x , y)     ((x)<(y) ? (x) : (y))
#define MAX(x , y)     ((x)>(y) ? (x) : (y))
#define LO             (0.1)
#define HI             (0.9)
#define BIAS           (1  )
#define sqr(x)         ((x)*(x))
#define Y              (0  )

typedef struct {                          /* A LAYER OF A NET:                     */
        int      Number_Of_Neurons;      /*- number of units in this layer        */
        double*         Output;           /* - output of ith unit                  */
        double*         Error;            /* - error term of ith unit              */
        double**        Weight;           /* - connection weights to ith unit      */
        double**        WeightSave;       /* - saved weights for stopped training  */
        double**        Delta_Weight;/* - last weight deltas for momentum */
} LAYER;

typedef struct {                          /* A NET:                                */
        LAYER**         Layer;            /* - layers of this net                  */
        LAYER*          InputLayer;       /* - input layer                         */
        LAYER*          OutputLayer;      /* - output layer                        */
        double          Alpha;            /* - momentum factor                     */
        double          Eta;              /* - learning rate                       */
        double          Gain;             /* - gain of sigmoid function            */
        double          Error;            /* - total net error                     */
} NET;

#define Traffic_Cycle (120) //Seconds
#define      History_Impact (3)  // Number of History rows to be taken into
                                 // consideration when estimating new row
#define Traffic_Dimensions    (8)// Number of Data Columns
#define Traffic_Light_Intervals (14)
#define Number_Of_Patterens   (1000)// Number of Number_Of_Patterens to estimate a Day is
                                    // 720 * (120 Sec)
#define MNN_Number_Of_Outputs (Traffic_Light_Intervals)
#define RNN_Number_Of_Outputs (Traffic_Dimensions)

int File_Output_Pointer[2] ={2*Traffic_Dimensions , Traffic_Dimensions };
int Number_Of_Layers[2] =   {3 , 4};
int Number_Of_Inputs[2] =   {32 , 32};
int Number_Of_Hidden[2] =   {Number_Of_Inputs[0] , Number_Of_Inputs[1]};
int Number_Of_Outputs[2] =  {MNN_Number_Of_Outputs , RNN_Number_Of_Outputs};
int Number_Of_Neurons[2][4]={{Number_Of_Inputs[0] , Number_Of_Hidden[0] ,
                              MNN_Number_Of_Outputs , 0} ,
                             {Number_Of_Inputs[1] , Number_Of_Hidden[1] ,
                              Number_Of_Hidden[1] , RNN_Number_Of_Outputs}};
int Data_In_Integer [Number_Of_Patterens][2*Traffic_Dimensions +
                     Traffic_Light_Intervals];
double Data_In_Double [Number_Of_Patterens][2*Traffic_Dimensions +
                      Traffic_Light_Intervals];
```

```
#define Training_Start_Pointer      (History_Impact)
#define Training_End_Pointer        (4 * Number_Of_Patterens/ 5)
#define Test_Start_Pointer          (Training_End_Pointer+1)
#define Test_End_Pointer            (Number_Of_Patterens)
int Loaded_Traffic_Indicator[Traffic_Dimensions]= {2,11,2,12,2,15,2,14};// Car
#define Can_Cross_In_Yellow_Percent (50)
#define Impact_On_Opposite_Direction_Percent (95) // The Change towards the other side
#define Green_NS (4)  // The index of the green Interval for North -South
#define Green_EW (11) // The index of the green Interval for East -West The Change towards the
                     // same side
#define Traffic_Impact (100- Impact_On_Opposite_Direction_Percent)
double Max_Speed_For_Running_Car =16.6;// Max Speed for running car ->16.6 M/Sec
double Speed_for_Starting_Car =8.3; // Speed for Starting Car->30 KM/Hour->5.5 M/Sec
double Avg_Distance_Between_Stopping_Cars =5;//Average Minimum Distance between Stopping
                                          //  cars->5 M
double Average_Car_Length =4;             // Average Car Length  ->5  M
double Spped_Gain_Time =5;                // SppedGainTime ->5  Sec
double Car_Start_Delay_Rate =2;           // Car Start Delay rate -> 2 Sec/Car
double Intersection_Leangth =20;          // Intersection Leangth ->30  M
double Avg_Distance_Between_Running_Cars =20; // Average Minimum Distance
                                              //  Between Running cars ->20 M
double Speed_Accelaration; //((Max_Speed_For_Running_Car- Speed_for_Starting_Car)/
                           // Spped_Gain_Time) Speed Increase Rate (Accelaration)->(16.67.6)/5
                           //   ->2.22    M/Sec^2
double Running_Car_Crossing_Time;//((Intersection_Leangth
                                 //+Average_Car_Length)/Max_Speed_For_Running_Car)
                                 // Running Car Crossing time->34/16.67->2.04Sec
double Max_Intersection_Flow_Rate; //(((Avg_Distance_Between_Running_Cars
                                   //+Average_Car_Length)/Intersection_Leangth
                                   // )/Running_Car_Crossing_Time)  Max
                                   //Intersection flow Rate for Running Cars->/2.04->Car/Sec
double Min_Time_Interval_Between_Running_Cars; //
int Performance_Metrics [Traffic_Dimensions]={0,0,0,0,0,0,0,0};  // Car
int Cars_In_Queue [Traffic_Dimensions]= {0,0,0,0,0,0,0,0};       // Car
int Waiting_In_Queue[Traffic_Dimensions]= {0,0,0,0,0,0,0,0};     // Car
long Performance_Metrics_Totals [Traffic_Dimensions]= {0,0,0,0,0,0,0,0}; // Car
int    SPM_Integer [Traffic_Dimensions]= {0,0,0,0,8,8,8,8};      // Car
double SPM_Double  [Traffic_Dimensions]= {SPM_Integer[0] , SPM_Integer[1] ,
                    SPM_Integer[2] , SPM_Integer[3] , SPM_Integer[4] ,
                    SPM_Integer[5] , SPM_Integer[6] , SPM_Integer[7] };  // Car
int Index[2][4]  = {{2,3,0,1},{6,7,4,5}};                 // Direction
int Vector[2][16]={{-1,2,2,2,1,0,-1,0,1,-1,-1,2,1,-1,1,0},{-1,9,9,9,8,7,-
1,7,8,-1,-1,9,8,7,8,7}}; // Vector (Status)
int Assign[16]   = {0,6,6,10,10,10,0,4,6,0,0,5,8,4,4,6}; // Out of 10 (Percent of Max
                                                         // Vector Time)
#define Traffic_Cycle (120) // in Seconds
#define Starter_Max (10)   // This is the Max Interval for the first Three vectors
#define Starter_Yellow_Max (3)
#define Yellow_Max (4)
#define Red_Max (1)  // Because it will turn yellow in the other side
                     // This "1" is for the Interval that they are all yellow
#define Green_Min (20)
// Max Vector Time
Int Stage_Max[Traffic_Light_Intervals]=
              {Starter_Max,Starter_Max,Starter_Max,Starter_Yellow_Max,
               Traffic_Cycle-(2 * (Starter_Max+Yellow_Max+Red_Max)),
               Yellow_Max,Red_Max, Starter_Max,Starter_Max,
               Starter_Max,Starter_Yellow_Max,Traffic_Cycle-(2 *
               (Starter_Max+Yellow_Max+Red_Max)),Yellow_Max, Red_Max};
// Min Vector Time
int Stage_Min[Traffic_Light_Intervals]={0,0,0,0, Green_Min,Yellow_Max,Red_Max,
                            0,0,0, 0, Green_Min,Yellow_Max,Red_Max};
// Max Vector Time
int Interval_B[Traffic_Light_Intervals];  // Output Vector Intervals
int Interval[Traffic_Light_Intervals];
// Mapping array between the vectors and the affected Traffic Directions
// [0],[1] for the major two affected Directions
// [2],[3] for the minor two affected Directions
// -1 Means no effect on any direction
//                  Noth      South   East      West
//                  L  S      L  S    L  S      L  S
int VDMap[4][Traffic_Light_Intervals]=
                  {{0,0,2,0,1,1,-1,4,4,6,4,5,5,-1},
                   {2,1,3,2,3,3,-1,6,5,7,6,7,7,-1},
                   {-1,-1,-1,-1,0,0,-1,-1,-1,-1,-1,4,4,-1},
                   {-1,-1,-1,-1,2,2,-1,-1,-1,-1,-1,6,6,-1}};
int VDMapY[Traffic_Light_Intervals]  = {0,0,0,1,0,1,0,0,0,0,1,0,1,0 };
double Waiting_Factor_NS ;
double Waiting_Factor_EW ;
double  Mean[2][Traffic_Light_Intervals];
double  TrainError[2];
double  TrainErrorPredictingMean[2];
double  TestError[2];
double  TestErrorPredictingMean[2];
double Input_Max[2][2 * Traffic_Dimensions];
double Input_Min[2][2 * Traffic_Dimensions];
double Output_Max[2][Traffic_Light_Intervals];
double Output_Min[2][Traffic_Light_Intervals];
FILE*  File_NN_Performance;
FILE*  File_Inputs;
FILE*  File_Outputs;
FILE*  File_Totals;
FILE*  File_Outputs_Formated;
FILE*  File_NN_Weights;
//------------------------------------------------------------------------
void InitializeRandoms()
{
  srand(4711);
}
//------------------------------------------------------------------------
void Load_Inputs()
{
       int  i,j,Step,GVar;
       long Lng;
       char S[28];
       int Chrr;
       for ( i =0; i <=Number_Of_Patterens-1; i ++)
       {
       Step =0;
       // Read The Actual Traffic Part of the file
       fscanf(File_Inputs , "%s" , S);
          fscanf(File_Inputs , "%d" , &Lng);
          fscanf(File_Inputs , "%s" , S);
          for ( j =0; j <=Traffic_Dimensions-1; j ++)
          {
             fscanf(File_Inputs , "%d" , &GVar);
                Data_In_Integer[i][j +Step]  = GVar ;
                Data_In_Double[i][j +Step]  =double(GVar);
          }
       // Ignore The Estimated Traffic Part of the file
       fscanf(File_Inputs , "%s" , S);
          for ( j =0; j <=Traffic_Dimensions-1; j ++)
          {
             fscanf(File_Inputs , "%d" , &Chrr);
          }
       // Read The Intervals Part of the file
       Step =2 * Traffic_Dimensions ;
       fscanf(File_Inputs , "%s" , S);
          for ( j =0; j <=Traffic_Light_Intervals-1; j ++)
          {
             fscanf(File_Inputs , "%d" , &GVar);
                Data_In_Integer[i][j +Step]  = GVar ;
                Data_In_Double[i][j +Step]    =double(GVar);
          }
          // Ignore the Cars waiting (Before) Part of the file
          fscanf(File_Inputs , "%s" , S);
          for ( j =0; j <=Traffic_Dimensions-1; j ++)
          {
                fscanf(File_Inputs , "%d" , &Chrr);
          }
          // Read the Cars waiting Part of the file
          Step =Traffic_Dimensions ;
          fscanf(File_Inputs , "%s" , S);
          for ( j =0; j <=Traffic_Dimensions-1; j ++)
          {
             fscanf(File_Inputs , "%d" , &GVar);
                Data_In_Integer[i][j +Step]  = GVar ;
                Data_In_Double[i][j +Step]    =double(GVar);
          }
     }
}
//------------------------------------------------------------------------
int RndInt(int Low , int High)
{
  return rand() % (High-Low+1) + Low;
}
//------------------------------------------------------------------------
double Rnddouble(double Low , double High)
{
  return ((double) rand() / RAND_MAX) * (High-Low) + Low;
}
//------------------------------------------------------------------------
void Normalize_Data()
{
  int  i , j ;
  // For Inputs
  // Initialize Minimums and Maximums
  for ( j =0;  j <2*Traffic_Dimensions;  j ++) {
       Input_Max[0][j] =
          Input_Max[1][j] = 0;

       Input_Min[0][j] =
          Input_Min[1][j] = 1000;
  }
  // Calculate Minimums and Maximums Per data Column
  for ( i =0;  i <Number_Of_Patterens-1;  i ++) {
        for ( j =0;  j <2*Traffic_Dimensions;  j ++) {
                Input_Max[0][j] =
                Input_Max[1][j] = MAX(Input_Max[1][j] , Data_In_Double[i][j]);

                Input_Min[0][j] =
                Input_Min[1][j] = MIN(Input_Min[1][j] , Data_In_Double[i][j]);
        }
  }
   // Initialize the Mean array
  for ( j =0;  j <Traffic_Light_Intervals;  j ++) {
      Mean[0][j] = 0;
         Mean[1][j] = 0;
  }
  // Normalize Inputs
  for ( i =0;  i <Number_Of_Patterens;  i ++) {
        for ( j =0;  j <2*Traffic_Dimensions;  j ++) {
             if (Input_Max[0][j]-Input_Min[0][j] != 0 )
                      Data_In_Double [i][j] = double(((Data_In_Double[i][j] -
                                Input_Min[0][j] ) / (Input_Max[0][j]-
                                Input_Min[0][j])) * (HI-LO) + LO);
             else
```

```
                    Data_In_Double [i][j] = LO ;
               if ( j >Traffic_Dimensions-1)
                    Mean[1][j -Traffic_Dimensions] += Data_In_Double[i][j] /
                                                       Number_Of_Patterens;
          }
     }

// SPM_Double Array (Ignore -1)
 for ( i =0;  i <Traffic_Dimensions;  i ++) {
    if (SPM_Double [i]!= -1)
       if (Input_Max[0][i +Traffic_Dimensions]-
              Input_Min[0][i +Traffic_Dimensions] != 0 )
          SPM_Double[i] = double(((SPM_Double[i] - Input_Min[0][i
          +Traffic_Dimensions] ) / (Input_Max[0][i +Traffic_Dimensions]-
          Input_Min[0][i +Traffic_Dimensions])) * (HI-LO) + LO);
       else
          SPM_Double[i] = LO ;
 }

 // For Outputs
 // Initialize Minimums and Maximums
 for ( j =0;  j <Traffic_Light_Intervals;  j ++) {
      Output_Max[0][j] = 0;
      Output_Max[1][j] = 0;
      Output_Min[0][j] = 1000;
      Output_Min[1][j] = 1000;
 }

 // Calculate Minimums and Maximums for NN #0
 for ( i =0;  i <Number_Of_Patterens-1;  i ++) {
      for ( j =2*Traffic_Dimensions;  j
           <2*Traffic_Dimensions+Traffic_Light_Intervals;  j ++) {

          Output_Min[0][j -2*Traffic_Dimensions] = MIN(Output_Min[0][j -
              2*Traffic_Dimensions] , Data_In_Double[i][j]);
          Output_Max[0][j -2*Traffic_Dimensions] = MAX(Output_Max[0][j -
              2*Traffic_Dimensions] , Data_In_Double[i][j]);
      }
 }
 // Normalize Outputs for NN #0
 for ( i =0;  i <Number_Of_Patterens;  i ++) {
      for ( j =2*Traffic_Dimensions;  j
           <(2*Traffic_Dimensions)+Traffic_Light_Intervals;  j ++) {
          if (Output_Max[0][j -2*Traffic_Dimensions]-Output_Min[0][j -
              2*Traffic_Dimensions] != 0 )
          {
              Data_In_Double [i][j] = double(((Data_In_Double[i][j]-
                  Output_Min[0][j -2 * Traffic_Dimensions]) /
                  (Output_Max[0][j -2*Traffic_Dimensions]-Output_Min[0][j -
                   2*Traffic_Dimensions])) * (HI-LO) + LO);

             Mean[0][j -2*Traffic_Dimensions] += Data_In_Double[i][j] /
                  Number_Of_Patterens;
           }
           else
           {
              Data_In_Double [i][j] = LO;
           }
      }
 }
 for ( j =Traffic_Dimensions;  j <(2*Traffic_Dimensions);  j ++) {
     Output_Max[1][j -Traffic_Dimensions] =  Input_Max[1][j];
     Output_Min[1][j -Traffic_Dimensions] =  Input_Min[1][j];
 }
```

```
}

void InitializeApplication(void)
{
  File_Inputs     = fopen("Inputs.txt" , "r");
  File_NN_Performance     = fopen("NNPerf.txt" , "w");
  File_Outputs     = fopen("Outputs.txt" , "w");
  File_Totals  = fopen("Totals.txt" , "a");
  File_Outputs_Formated    = fopen("Formatted.txt" , "w");
  File_NN_Weights     = fopen("Weights.txt" , "w");

  Load_Inputs();
  Normalize_Data();

  Speed_Accelaration = 1.0 * ((Max_Speed_For_Running_Car-
Speed_for_Starting_Car)/Spped_Gain_Time); // Speed Increase Rate (Accelaration)->(16.67-
5.6)/5     ->2.22    M/Sec^2
  Running_Car_Crossing_Time = 1.0 *
((Intersection_Leangth+Average_Car_Length)/Max_Speed_For_Running_Car)        ; //
Running Car Crossing time->34/16.67 ->2.04 Sec
  Max_Intersection_Flow_Rate = ((1.0 * ( Avg_Distance_Between_Running_Cars +
Average_Car_Length )/Intersection_Leangth)/Running_Car_Crossing_Time) ; // Max
Intersection flow Rate for Running Cars->0.67/2.04->0.33    Car/Sec
  Min_Time_Interval_Between_Running_Cars =
(Avg_Distance_Between_Running_Cars+Average_Car_Length)/Max_Speed_For_Running_Ca
r ;                  // MinTimeInterval between Running Cars
}
//---------------------------------------------------------------------------
void InitializeNetwork(NET* Net , int Which_NN)
{
  int  Line , i ;
  int NN_Specific;
  double Out , Err;
  Net->Alpha = 0.5;
  Net->Eta   = double(0.05);
  Net->Gain  = 1;

  if (Which_NN==0)
     NN_Specific = 2*Traffic_Dimensions; else NN_Specific = Traffic_Dimensions;
  TrainErrorPredictingMean[Which_NN] = 0;
  for (Line=Training_Start_Pointer; Line<=Training_End_Pointer; Line++) {
    for ( i =0;  i <Number_Of_Outputs[Which_NN];  i ++) {
      Out = Data_In_Double[Line][i +(NN_Specific)];
      Err = Mean[Which_NN][i] - Out;
      TrainErrorPredictingMean[Which_NN] += double(0.5 * sqr(Err));
    }
  }
  TestErrorPredictingMean[Which_NN] = 0;
  for (Line=Test_Start_Pointer; Line<=Test_End_Pointer; Line++) {
    for ( i =0;  i <Number_Of_Outputs[Which_NN];  i ++) {
      Out = Data_In_Double[Line][i +(NN_Specific)];
      Err = Mean[Which_NN][i] - Out;
      TestErrorPredictingMean[Which_NN] += double(0.5 * sqr(Err));
    }
  }
}
//---------------------------------------------------------------------------
double Delay_In_Start(int NC)// Delay in Start for Car Number NC inline->2NC ->2NC Sec
{
       return ((NC-1)*Car_Start_Delay_Rate);
}
//---------------------------------------------------------------------------
double Distance_From_Intersection(int NC) // Distance from Intersection->10NC ->10NC  M
{
    return ((NC-1)*(Avg_Distance_Between_Stopping_Cars+Average_Car_Length));
}
//------------------------------------------------------------------------
double Time_To_Cross(int NC , int Root) // Root #1, #2 for Time Quadratic Equation which is
                        {               // Time to Cross the NC waiting Cars
  double a , b , c , SqrtTerm , Out;
  int Sign;
  if (NC>0)
  {
    if (Root ==1) Sign =1; else Sign =-1;
    a = (double)0.5 * Speed_Accelaration ;
    b = Speed_for_Starting_Car ;
    c = -1 * ( Distance_From_Intersection(NC) + Intersection_Leangth +
        Average_Car_Length ) ;
    SqrtTerm =  ( b * b) - 4 * a * c ;
    if (a==0)
       if (b==0)
           Out  = -1.0 ;
         else
           Out  = Delay_In_Start(NC) + ((-1.0 * b)/c);
    else
         if (SqrtTerm < 0)
           Out  = -1.0 ;
         else
           Out = Delay_In_Start(NC)+(-1 * b +(Sign * sqrt(SqrtTerm))) / (2*a);
  }
  else
  {
     Out=0;
  }
  return Out;
}
//------------------------------------------------------------------------
double Time_To_Cross_Root_Selected(int NC) //Selected Root for Time Quadratic Equation
which is
{                                       // Time to Cross the NC waiting Cars
      double Root1 , Root2;
    if(NC==1)
      {
            NC=1;
      }
      Root1=Time_To_Cross(NC , 1);
    Root2=Time_To_Cross(NC , 2);

      return MAX(Root1 , Root2);
}
//------------------------------------------------------------------------
void FinalizeApplication()
{
  fclose(File_NN_Performance);
  fclose(File_Inputs);
  fclose(File_Outputs);
  fclose(File_Totals);
  fclose(File_Outputs_Formated);
  fclose(File_NN_Weights);
}
//------------------------------------------------------------------------
void Generate_Neural_Network(NET* Net , int Which_NN)
{
  int l , i ;
  Net->Layer = (LAYER**) calloc(Number_Of_Layers[Which_NN] , sizeof(LAYER*));
  for (l=0; l<Number_Of_Layers[Which_NN]; l++) {
    Net->Layer[l] = (LAYER*) malloc(sizeof(LAYER));
    Net->Layer[l]->Number_Of_Neurons      = Number_Of_Neurons[Which_NN][l];
    Net->Layer[l]->Output      = (double*)
        calloc(Number_Of_Neurons[Which_NN][l]+1 , sizeof(double));
    Net->Layer[l]->Error       = (double*)
        calloc(Number_Of_Neurons[Which_NN][l]+1 , sizeof(double));
    Net->Layer[l]->Weight      = (double**)
        calloc(Number_Of_Neurons[Which_NN][l]+1 , sizeof(double*));
    Net->Layer[l]->WeightSave = (double**)
        calloc(Number_Of_Neurons[Which_NN][l]+1 , sizeof(double*));
    Net->Layer[l]->Delta_Weight     = (double**)
        calloc(Number_Of_Neurons[Which_NN][l]+1 , sizeof(double*));
    Net->Layer[l]->Output[0]  = BIAS;
    if (l != 0) {
      for ( i =1;  i <=Number_Of_Neurons[Which_NN][l];  i ++) {
        Net->Layer[l]->Weight[i]      = (double*)
           calloc(Number_Of_Neurons[Which_NN][l-1]+1 , sizeof(double));
        Net->Layer[l]->WeightSave[i] = (double*)
           calloc(Number_Of_Neurons[Which_NN][l-1]+1 , sizeof(double));
        Net->Layer[l]->Delta_Weight[i]     = (double*)
           calloc(Number_Of_Neurons[Which_NN][l-1]+1 , sizeof(double));
      }
    }
  }
  Net->InputLayer  = Net->Layer[0];
  Net->OutputLayer = Net->Layer[Number_Of_Layers[Which_NN] - 1];
  Net->Alpha       = double(0.9);
  Net->Eta         = double(0.25);
  Net->Gain        = 1;
}
//------------------------------------------------------------------------
void Set_Random_Weights(NET* Net , int Which_NN)
{
  int l , i , j ;
  for (l=1; l<Number_Of_Layers[Which_NN]; l++) {
    for ( i =1;  i <=Net->Layer[l]->Number_Of_Neurons;  i ++) {
      for ( j =0;  j <=Net->Layer[l-1]->Number_Of_Neurons;  j ++) {
        Net->Layer[l]->Weight[i][j] = Rnddouble(-0.5 , 0.5);
      }
    }
  }
}
//------------------------------------------------------------------------
void Set_Inputs(NET* Net , int InputLine)
{
  int  i , j , k=0;
  for ( i =0; i <History_Impact; i ++) {
     for ( j =0; j <Traffic_Dimensions; j ++) {
      k++;
      Net->InputLayer->Output[k] =
          Data_In_Double[InputLine-History_Impact+ i][j];
     }
  }
  for ( j =0; j <Traffic_Dimensions; j ++) {
      k++;
      Net->InputLayer->Output[k] =
          Data_In_Double[InputLine-1][Traffic_Dimensions+  j];
  }
}
//------------------------------------------------------------------------
void Get_Output(NET* Net , double* Output)
{
  int  i ;
  for ( i =1;  i <=Net->OutputLayer->Number_Of_Neurons;  i ++) {
    Output[i -1] = Net->OutputLayer->Output[i];
  }
}
//-----------------------------------------------------------------------
void Save_Weights(NET* Net , int Which_NN)
{
  int l , i , j ;
  for (l=1; l<Number_Of_Layers[Which_NN]; l++) {
    for ( i =1;  i <=Net->Layer[l]->Number_Of_Neurons;  i ++) {
      for ( j =0;  j <=Net->Layer[l-1]->Number_Of_Neurons;  j ++) {
        Net->Layer[l]->WeightSave[i][j] = Net->Layer[l]->Weight[i][j];
      }
    }
  }
}
//-----------------------------------------------------------------------
void Copy_Weights(NET* Net0 , NET* Net1 , int Which_NN)
{
  int l , i , j ;

  for (l=1; l<Number_Of_Layers[0]; l++) {
    for ( i =1;  i <=Net0->Layer[l]->Number_Of_Neurons;  i ++) {
      for ( j =0;  j <=Net0->Layer[l-1]->Number_Of_Neurons;  j ++) {
         if (Which_NN==0)
           Net0->Layer[l]->Weight[i][j]      = Net1->Layer[l]->Weight[i][j] ;
         else
           Net1->Layer[l]->Weight[i][j]      = Net0->Layer[l]->Weight[i][j] ;
      }
    }
  }
}
//-----------------------------------------------------------------------
void Print_Weights(NET* Net0 , NET* Net1)
{
  int l , i , j ;
  for (l=1; l<Number_Of_Layers[1]; l++) {
    for ( i =1;  i <=Net1->Layer[l]->Number_Of_Neurons;  i ++) {
      for ( j =0;  j <=Net1->Layer[l-1]->Number_Of_Neurons;  j ++) {
         if (l<Number_Of_Layers[0] &&  i <=Net0->Layer[l]->Number_Of_Neurons &&
              j <=Net0->Layer[l-1]->Number_Of_Neurons)
           fprintf(File_NN_Weights , "\nLayer :%2d >- %2d  Unit :%2d >- %2d
                Weight#0 : %12f   Weight#1 : %12f   #0-#1 : %12f " , l-1 , l ,
                 j , i , Net1->Layer[l]->Weight[i][j] , Net0->Layer[l]-
                 >Weight[i][j] , Net0->Layer[l]->Weight[i][j]- Net1->Layer[l]-
                 >Weight[i][j]);
         else
           fprintf(File_NN_Weights , "\nLayer :%2d >- %2d  Unit :%2d >- %2d
                  Weight#0 : %f " , l-1 , l , j , i , Net1->Layer[l]-
                  >Weight[i][j]);
      }
    }
  }
}
//-----------------------------------------------------------------------
void Restore_Weights(NET* Net , int Which_NN)
{
  int l , i , j ;
  for (l=1; l<Number_Of_Layers[Which_NN]; l++) {
    for ( i =1;  i <=Net->Layer[l]->Number_Of_Neurons;  i ++) {
      for ( j =0;  j <=Net->Layer[l-1]->Number_Of_Neurons;  j ++) {
        Net->Layer[l]->Weight[i][j] = Net->Layer[l]->WeightSave[i][j];
      }
    }
  }
}
```

```
//-------------------------------------------------------------------------
void Propagate_Layer(NET* Net , LAYER* Lower , LAYER* Upper)
{
  int   i , j ;
  double Sum;
  for ( i =1;  i <=Upper->Number_Of_Neurons;  i ++) {
    Sum = 0;
    for ( j =1;  j <=Lower->Number_Of_Neurons;  j ++) {
      Sum += Upper->Weight[i][j] * Lower->Output[j];
    }
    Upper->Output[i] = 1 / (1 + exp(-Net->Gain * Sum));
  }
}
//-------------------------------------------------------------------------
void Propagate_Neural_Network(NET* Net , int Which_NN)
{
  int l;

  for (l=0; l<Number_Of_Layers[Which_NN]-1; l++) {
    Propagate_Layer(Net , Net->Layer[l] , Net->Layer[l+1]);
  }
}
//-------------------------------------------------------------------------
void Compute_Output_Error(NET* Net , int Target , int Which_NN , int ForBest)
{
  int   i , k;
  double Out , Err;

  Net->Error = 0;
  for ( i =1;  i <=Net->OutputLayer->Number_Of_Neurons;  i ++) {
    Out = Net->OutputLayer->Output[i];

      k = File_Output_Pointer[Which_NN] +  i  - 1;
    if (ForBest==false)
         Err = Data_In_Double[Target][k]- Out;
      else
              if (SPM_Double[i -1]==-1)
             Err = Data_In_Double[Target][k]- Out;
              else
             Err = 1 * -1 * (SPM_Double[i -1]- Out);

    Net->OutputLayer->Error[i] = Net->Gain * Out * (1-Out) * Err;
    Net->Error += double(0.5 * sqr(Err));
  }
}
//-------------------------------------------------------------------------
void Back_Propagate_Layer(NET* Net , LAYER* Upper , LAYER* Lower)
{
  int   i , j ;
  double Out , Err;
  for ( i =1;  i <=Lower->Number_Of_Neurons;  i ++) {
    Out = Lower->Output[i];
    Err = 0;
    for ( j =1;  j <=Upper->Number_Of_Neurons;  j ++) {
      Err += Upper->Weight[j][i] * Upper->Error[j];
    }
    Lower->Error[i] = Net->Gain * Out * (1-Out) * Err;
  }
}
//-------------------------------------------------------------------------
void Back_Propagate_Neural_Network(NET* Net , int Which_NN)
{
  int l;
  for (l=Number_Of_Layers[Which_NN]-1; l>1; l--) {
    Back_Propagate_Layer(Net , Net->Layer[l] , Net->Layer[l-1]);
  }
}
//----------------------------------------------------------------------
void Adjust_Weights(NET* Net , int Which_NN)
{
  int  l , i , j ;
  double Out , Err , Delta_Weight;

  for (l=1; l<Number_Of_Layers[Which_NN]; l++) {
    for ( i =1;  i <=Net->Layer[l]->Number_Of_Neurons;  i ++) {
      for ( j =0;  j <=Net->Layer[l-1]->Number_Of_Neurons;  j ++) {
        Out = Net->Layer[l-1]->Output[j];
        Err = Net->Layer[l]->Error[i];
        Delta_Weight = Net->Layer[l]->Delta_Weight[i][j];
        Net->Layer[l]->Weight[i][j] += Net->Eta * Err * Out + Net->Alpha *
            Delta_Weight;
        Net->Layer[l]->Delta_Weight[i][j] = Net->Eta * Err * Out;
      }
    }
  }
}

void Simulate_Neural_Network(NET* Net , int Line , double* Output , int
Training , int Which_NN , int ForBest)
{
  Set_Inputs(Net , Line );
  Propagate_Neural_Network(Net , Which_NN);
  Get_Output(Net , Output);

  Compute_Output_Error(Net , Line , Which_NN , ForBest);

  if (Training) {
    Back_Propagate_Neural_Network(Net , Which_NN);
    Adjust_Weights(Net , Which_NN);
  }
}
//----------------------------------------------------------------------
void Train_Neural_Network(NET* Net0 , NET* Net1 , int Epochs)
{
  int  Line , a;

  double Output0[MNN_Number_Of_Outputs];
  double Output1[RNN_Number_Of_Outputs];

  for (a=0; a<Epochs; a++) {
         for (Line=Training_Start_Pointer; Line<Training_End_Pointer; Line++) {
          if(Line == 535)
          Line =Line;
               Simulate_Neural_Network(Net0 , Line , Output0 , 1 , 0 , false);
          Copy_Weights  (Net0 , Net1 , 1);
               Simulate_Neural_Network(Net1 , Line , Output1 , 1 , 1 , false);
          Copy_Weights  (Net0 , Net1 , 0);
          }
  }
}
//----------------------------------------------------------------------
void Test_Neural_Network(NET* Net0 , NET* Net1)
{
  int  Line;
  double Output0[MNN_Number_Of_Outputs];
  double Output1[MNN_Number_Of_Outputs];
  TrainError[0] = 0;
  TrainError[1] = 0;
  for (Line=Training_Start_Pointer; Line<=Training_End_Pointer; Line++) {
    Simulate_Neural_Network(Net0 , Line , Output0 , 0 , 0 , false);
    TrainError[0] += Net0->Error;
    Simulate_Neural_Network(Net1 , Line , Output1 , 0 , 1 , false);
    TrainError[1] += Net1->Error;
  }

  TestError[0] = 0;
  TestError[1] = 0;
  for (Line=Test_Start_Pointer; Line<=Test_End_Pointer; Line++) {
    Simulate_Neural_Network(Net0 , Line , Output0 , 0 , 0 , false);
    TestError[0] += Net0->Error;
    Simulate_Neural_Network(Net1 , Line , Output1 , 0 , 1 , false);
    TestError[1] += Net1->Error;
  }
  fprintf(File_NN_Performance , "\nNetwork #0# Mean Square Error is %0.3f on
      Training Set && %0.3f on Test Set" , TrainError[0] /
      TrainErrorPredictingMean[0] , TestError[0] / TestErrorPredictingMean[0]);
  fprintf(File_NN_Performance , "\n    #1#      %0.3f    %0.3f  " ,
      TrainError[1] / TrainErrorPredictingMean[1] ,
      TestError[1] / TestErrorPredictingMean[1]);
}
//------------------------------------------------------------------------
int DRound(double fNumber)
{
   if ((fNumber - int(fNumber))>=0.5)
          return int(fNumber)+1;
   else
          return int(fNumber);
}
//------------------------------------------------------------------------
int De_Normalize_Output(double fInterval , int TheJ , int Which_NN )
{
  return DRound( ((fInterval-LO)/(HI-LO)) * (Output_Max[Which_NN][TheJ]-
Output_Min[Which_NN][TheJ])  + Output_Min[Which_NN][TheJ] );
}
//------------------------------------------------------------------------
double Normalize_Output(int iInterval , int TheJ , int Which_NN)
{
  return double(( (iInterval-Output_Min[Which_NN][TheJ]) /
(Output_Max[Which_NN][TheJ]-Output_Min[Which_NN][TheJ]) ) * (HI-LO) + LO);
}
//------------------------------------------------------------------------
int De_Normalize_Input(double fCars , int TheJ , int Which_NN )
{
  return DRound(((fCars-LO)/(HI-LO)) * (Input_Max[Which_NN][TheJ]-
Input_Min[Which_NN][TheJ]) + Input_Min[Which_NN][TheJ]);
}
//------------------------------------------------------------------------
double Normalize_Input(int iCars , int TheJ , int Which_NN)
{
  return double(( (iCars-Input_Min[Which_NN][TheJ]) /
(Input_Max[Which_NN][TheJ]-Input_Min[Which_NN][TheJ]) ) * (HI-LO) + LO);
}
//------------------------------------------------------------------------
void Put_Cars_In_Queue(int Line)
{
       int  i ;
       int WaitingCars=0;
       for ( i =0;  i <=Traffic_Dimensions-1;  i ++)
        {
          if ( i <=5)
               WaitingCars = int(Waiting_Factor_NS * Data_In_Integer[Line][i]);
          else
               WaitingCars = int(Waiting_Factor_EW * Data_In_Integer[Line][i]);
          WaitingCars+=RndInt( 0 , Data_In_Integer[Line][i]- WaitingCars) -
               int(1.0 * (Data_In_Integer[Line][i]- WaitingCars)/2);
          if (WaitingCars<0) WaitingCars =0;
          if (WaitingCars>Data_In_Integer[Line][i])
             WaitingCars =Data_In_Integer[Line][i];

          Performance_Metrics[i] += WaitingCars;
          Cars_In_Queue[i]   = Data_In_Integer[Line][i]-WaitingCars;
          Waiting_In_Queue[i]=Performance_Metrics[i];
    }
}
//-----------------------------------------------------------------------
void Add_Queue_To_Next_Waiting(void)
{
       int  i ;
       for ( i =0;  i <=Traffic_Dimensions-1;  i ++)
        {
          Performance_Metrics[i]+=Cars_In_Queue[i];
        }
}
//-----------------------------------------------------------------------
void Cross_On_Yellow(int  j , int  i )
{
   if (Performance_Metrics[VDMap[j][i]] == 0)
   {
     if (Cars_In_Queue[VDMap[j][i]] > 0)
        {
        if (RndInt(0 , 100)<=Can_Cross_In_Yellow_Percent)
           Cars_In_Queue[VDMap[j][i]]--;
        }
   }
   else
   {
      if (RndInt(0 , 100)<=Can_Cross_In_Yellow_Percent ||
               RndInt(0 , 100)<=Can_Cross_In_Yellow_Percent     )
          Performance_Metrics[VDMap[j][i]]--;
   }
}
//-----------------------------------------------------------------------
void Cross_On_Green_Straight(int  j , int  i )
{
       int c , k , ComingFirst;
       double LastCarStartDelay , IntervalRem;

    LastCarStartDelay = Delay_In_Start(Performance_Metrics[VDMap[j][i]]);
       ComingFirst=0;
       if (Cars_In_Queue[VDMap[j][i]]>=1 && LastCarStartDelay > 0)
       {  // Then Estimate How many Cars will come during the delay in start of the last car in
          // line. Put the Estimated in ComingFirst Variable.
        if (Cars_In_Queue[VDMap[j][i]]>=1)
          {
           for (c=1; c<=Cars_In_Queue[VDMap[j][i]]; c++)
           {
          if(LastCarStartDelay  > ComingFirst *
             Min_Time_Interval_Between_Running_Cars)
             {
                if (Rnddouble(0.0 , Interval[i]) <= (LastCarStartDelay-
                      (ComingFirst * Min_Time_Interval_Between_Running_Cars)) )
                 {
                      ComingFirst++;
                      LastCarStartDelay = Delay_In_Start(Performance_Metrics
                            [VDMap[j][i]] + ComingFirst );
```

```
                  }
              }
              else
              {
                  break;
              }
         }
      }
  }
  IntervalRem = Interval[i];
  if (IntervalRem <= LastCarStartDelay)
  { // The Interval is not enough for crossing the waiting cars so , Some will cross and some will
     // stay and all the incomming cars will stay in the next waiting line.
     for(k=ComingFirst+Performance_Metrics[VDMap[j][i]];k>=1;k--)
     {  // calculate the number of cars that will be able to cross. put it in k and subtract
        // it from the waiting list
        if (IntervalRem >=Time_To_Cross_Root_Selected(k))
        {
           Performance_Metrics[VDMap[j][i]]+=Cars_In_Queue[VDMap[j][i]]-k;
           Cars_In_Queue[VDMap[j][i]] =0;
           break;
        }
     }
  }
  else
  {   // The Interval is enough for all the waiting to Cross the propably some of the incomming
      // or all will pass. Subtract teh crossing time of all the waiting and estimated comming
      // in the delay from the remaining interval
      Performance_Metrics[VDMap[j][i]] = 0;
      IntervalRem -=Time_To_Cross_Root_Selected(ComingFirst+
          Performance_Metrics[VDMap[j][i]]);
      Cars_In_Queue[VDMap[j][i]]-= ComingFirst;
      if (IntervalRem < (Cars_In_Queue[VDMap[j][i]]) *
          Min_Time_Interval_Between_Running_Cars ){
          // The Remaining time is not enough for all incommming running cars to cross
          // Some will be added to waiting and some will pass
          Cars_In_Queue[VDMap[j][i]] =(Cars_In_Queue[VDMap[j][i]]) -
                int(IntervalRem/Min_Time_Interval_Between_Running_Cars);
          Performance_Metrics[VDMap[j][i]]= 0;
      }
       else
       { // The Remaining time is enough for all incomming running cars to cross
          Performance_Metrics[VDMap[j][i]] = 0;
          Cars_In_Queue[VDMap[j][i]] = 0;
      }
   }
  }
  //------------------------------------------------------------------------------
  void Cross_On_Green_Left(int  j , int  i )
  {
    int Depend , CanCross , c ;
    double FreeFactor;
    if ( j ==2) Depend=1; else Depend=0;
    FreeFactor = (Loaded_Traffic_Indicator[VDMap[Depend][i]] -
           Cars_In_Queue[VDMap[Depend][i]])
           /Loaded_Traffic_Indicator[VDMap[Depend][i]] ;
    if (FreeFactor <0 ) FreeFactor=0;
    //   Lets Calculate how many cars can cross if the opposite direction is empty
    //   Then Multiply it by the free Factor to be the actual Cars that can cross.
    CanCross =0;
    // Even The Running cars will have to slow down like a stop
   // to be able to cross , so they will be treated as Performance Metrics cars.
    for (c=1; c<=Performance_Metrics[VDMap[j][i]]+
         Cars_In_Queue[VDMap[j][i]]; c++)
    {
       if (Interval[i] >= Time_To_Cross_Root_Selected(c))
            CanCross=c;
         else
            break;
  }
  // Only Part of them Can cross according to the corresponding traffic
  if (CanCross > 0 ) CanCross = int( CanCross * FreeFactor );

  if (Performance_Metrics[VDMap[j][i]] >= CanCross)
  {
     Performance_Metrics[VDMap[j][i]] -= CanCross;
  }
  else
  {
     Cars_In_Queue[VDMap[j][i]] -= CanCross - Performance_Metrics[VDMap[j][i]];
     Performance_Metrics[VDMap[j][i]] = 0 ;
  }
}
//------------------------------------------------------------------------
void Calc_Metrics(int Line)
{
  int  i , j , TotNS=0;

  for ( i =0;  i <=6;  i ++)
  {
    TotNS+= Interval[i];
  }
  Waiting_Factor_EW = 1.0 * TotNS / Traffic_Cycle;
  Waiting_Factor_NS = 1.0 * (Traffic_Cycle-TotNS) / Traffic_Cycle;
  Put_Cars_In_Queue(Line);
  for ( i =0;  i <=Traffic_Light_Intervals-1;  i ++)
  {
    if (Interval[i]>0)
       {
       for ( j =0;  j <=3;  j ++)
          {
         if (VDMap[j][i]!=-1)
         {
             if (VDMapY[i]==1)
                {  // It is Yellow
                Cross_On_Yellow( j , i );
                }
                else
                { // It is Green
                      if ( j <=1)// Major affected Directions
                            Cross_On_Green_Straight( j , i );
                      Else // Minor affected Directions
                             Cross_On_Green_Left( j , i );
                }
          }
       }
    }
  }
Add_Queue_To_Next_Waiting();
}
//------------------------------------------------------------------------
void Push_Text_Line(int Line)
{
 int  i  ;
 fprintf(File_Outputs , "SoFar___ ");
 fprintf(File_Outputs , "%5d  " , Line);
 fprintf(File_Outputs , "Actual___ ");

 for ( i =0;  i <=Traffic_Dimensions-1;  i ++)
```

```
  {
    fprintf(File_Outputs , "%3d " , Data_In_Integer[Line][i]);
  }
  fprintf(File_Outputs , "  Programmed_Waiting");
  for ( i =Traffic_Dimensions;  i <2*Traffic_Dimensions;  i ++)
  {
         fprintf(File_Outputs , "%3d " , Data_In_Integer[Line][i]);
  }
  fprintf(File_Outputs , "  Programmed_Intervals ");
  for ( i =0;  i <Traffic_Light_Intervals;  i ++)
  {
    fprintf(File_Outputs , "%3d " , Data_In_Integer[Line][i
                  +2*Traffic_Dimensions]);
  }
  fprintf(File_Outputs , " NN Intervals___");
  for ( i =0;  i <=Traffic_Light_Intervals-1;  i ++)
  {
    fprintf(File_Outputs , "%3d " , Interval_B[i]);
  }
  fprintf(File_Outputs , " Adapted Intervals___");
  for ( i =0;  i <=Traffic_Light_Intervals-1;  i ++)
  {
    fprintf(File_Outputs , "%3d " , Interval[i]);
  }
  fprintf(File_Outputs , "  Waiting_Before_Control___");
  for ( i =0;  i <=Traffic_Dimensions-1;  i ++)
  {
    fprintf(File_Outputs , "%3d " , Waiting_In_Queue[i]);
  }
  fprintf(File_Outputs , "  Waiting___");
  for ( i =0;  i <=Traffic_Dimensions-1;  i ++)
  {
    fprintf(File_Outputs , "%3d " , Performance_Metrics[i]);
  }
  fprintf(File_Outputs , "\n");
}
//------------------------------------------------------------------------
void Push_Text_Line_1(int Line)
{
  int  i  ;
  fprintf(File_Outputs_Formated , "SoFar___ ");
  fprintf(File_Outputs_Formated , "%5d  " , Line);
  fprintf(File_Outputs_Formated , "Actual___ ");
  for ( i =0;  i <=Traffic_Dimensions-1;  i ++)
  {
    fprintf(File_Outputs_Formated , "%3d " , Data_In_Integer[Line][i]);
  }
  fprintf(File_Outputs_Formated , " Programmed_Waiting___");
  for ( i =0;  i <=Traffic_Dimensions-1;  i ++)
  {
    fprintf(File_Outputs_Formated , "%3d " , Data_In_Integer[Line][i
         +Traffic_Dimensions]);
  }
  fprintf(File_Outputs_Formated , " NN Waiting___");
  for ( i =0;  i <=Traffic_Dimensions-1;  i ++)
  {
    fprintf(File_Outputs_Formated , "%3d " , Performance_Metrics[i]);
  }
  fprintf(File_Outputs_Formated , "\n");
}
//------------------------------------------------------------------------
void Adapt_Next_Traffic_Loads(int Line , int Which_NN)
{
  int  i ;
  for ( i =0; i <Traffic_Dimensions; i ++)
  {
     Data_In_Double [Line][Traffic_Dimensions +  i]=
     Normalize_Input(Performance_Metrics[i] , i , Which_NN);
  }
}
//------------------------------------------------------------------------
void Adapt_Intervals(void)
{
  int  i , T3=0 , TheOne , Rem=0 , ToT=0 , GiveToNS=0 , GiveToEW=0;

  for ( i =0; i <Number_Of_Outputs[0]; i ++)
  {
       Interval[i] = 0;
  }
  T3= Interval_B[0] + Interval_B[1] + Interval_B[2];
  if (T3<=3)
  {
         Rem=Rem +T3;
         T3=0;
  }
  if (T3>Starter_Max)
  {
         Rem=Rem + (T3 - Starter_Max);
         T3=Starter_Max;
  }
  if (Interval_B[0] >= Interval_B[1]) TheOne = 0; else TheOne = 1;
  if (Interval_B[TheOne] < Interval_B[2]) TheOne = 2;

  Interval[TheOne] = T3;
  if (TheOne == 0 && T3>0)
         Interval[3]= Starter_Yellow_Max;
  else
  {
         Interval[3]= 0;
  }
  Rem=Rem +Interval_B[3]-Interval[3];
  T3=Interval_B[7]+Interval_B[8]+Interval_B[9];
  if (T3<=3)
  {
         Rem=Rem +T3;
         T3=0;
  }
  if (T3>Starter_Max)
  {
         Rem=Rem + (T3 - Starter_Max);
         T3=Starter_Max;
  }
  if (Interval_B[7] >= Interval_B[8]) TheOne = 7; else TheOne = 8;
  if (Interval_B[TheOne] < Interval_B[9]) TheOne = 9;

  Interval[TheOne] = T3;
  if (TheOne == 7 && T3>0)
         Interval[10]= Starter_Yellow_Max;
  else
  {
         Interval[10]= 0;
  }
  Rem=Rem +Interval_B[10]-Interval[10];
  Interval[5] =
  Interval[12]= Yellow_Max;
  Interval[6] =
  Interval[13]= Red_Max;
  Rem = Rem + Interval_B[5] -Interval[5];
  Rem = Rem + Interval_B[12] -Interval[12];
  Rem = Rem + Interval_B[6] -Interval[6];
  Rem = Rem + Interval_B[13] -Interval[13];
  for ( i =0; i <Number_Of_Outputs[0]; i ++)
  {
       ToT+=Interval[i];
  }
  ToT+= Interval_B[4];
  ToT+= Interval_B[11];
  Rem = Rem + Traffic_Cycle -ToT;
  GiveToNS = int((Interval_B[4]/(Interval_B[4]+Interval_B[11])) * Rem);
  GiveToEW = Rem- GiveToNS;
  Interval[4] = Interval_B[4]+ GiveToNS;
  Interval[11] = Interval_B[11]+ GiveToEW;
}
//-----------------------------------------------------------------------
void Push_Totals_Line(char *Comment , int Epochs)
{
       int  i ;
       fprintf(File_Totals , "\n %s : " , Comment);
       for ( i =0; i <Traffic_Dimensions; i ++)
       {
              fprintf(File_Totals , "%5d " , Performance_Metrics_Totals[i]);
       }
}
//-----------------------------------------------------------------------
void Test_Network_Performance(NET* Net)
{
    int  i , Line;
    double Output[MNN_Number_Of_Outputs];

       for (Line=Training_Start_Pointer; Line<=Number_Of_Patterens-1; Line++)
       {
          printf("\nTraffic Shaping....   %3d   ...... " , Line);
          Simulate_Neural_Network(Net , Line , Output , 0 , 0 , false);
          for ( i =0; i <Number_Of_Outputs[0]; i ++)
          {
             Interval_B[i] = De_Normalize_Output(Output[i] , i , 0);
             Data_In_Double[Line][2*Traffic_Dimensions+  i]= Output[i];
          }
          Adapt_Intervals();
          Calc_Metrics(Line);
          Adapt_Next_Traffic_Loads(Line , 0);
          for ( i =0; i <Traffic_Dimensions; i ++)
          {
              Performance_Metrics_Totals[i] += Performance_Metrics[i];
          }
          Push_Text_Line(Line);
          if(Line==Number_Of_Patterens-1)
              Push_Totals_Line("Before" , -1);
       }
       for ( i =0; i <Traffic_Dimensions; i ++)
       {
         Performance_Metrics_Totals[i] = 0;
       }
}
//-----------------------------------------------------------------------
void Performance_Without_SPM(NET* Net)
{
  int  i , Line;
  double Output[RNN_Number_Of_Outputs];
```

```
       for (Line=Training_Start_Pointer; Line<=Number_Of_Patterens-1; Line++)
       {
        if (Line==535)
          Line = Line;
          printf("\nTraffic Shaping.....   %3d   ....... " , Line);
          Simulate_Neural_Network(Net , Line , Output , 0 , 1 , false);
          for ( i =0; i <Number_Of_Outputs[1]; i ++)
          {
             Performance_Metrics[i] =  De_Normalize_Output(Output[i] , i , 1);
          }
          for ( i =0; i <Traffic_Dimensions; i ++)
          {
             Performance_Metrics_Totals[i] += Performance_Metrics[i];
          }
          Push_Text_Line_1(Line);
       }
       fprintf(File_Outputs_Formated , "\n Total Performance_Metrics Is");
       fprintf(File_Outputs_Formated , "\n ----------------\n");
       fprintf(File_Outputs_Formated , "\n>>>>>>>>>>>>>>>>>>>>>>>>>  ");
       for ( i =0; i <Traffic_Dimensions; i ++)
       {
         fprintf(File_Outputs_Formated , "%3d " ,
              Performance_Metrics_Totals[i]);
         Performance_Metrics_Totals[i] = 0;
       }
}
//---------------------------------------------------------------------------
void TrainNetForBest(NET* Net1 , int Epochs)
{
  int  Line , a;
  double Output1[RNN_Number_Of_Outputs];
  for (a=0; a<Epochs; a++) {
      for (Line=Training_Start_Pointer; Line<Test_End_Pointer; Line++) {
         Simulate_Neural_Network(Net1 , Line , Output1 , 1 , 1 , true );
      }
   }
}
//---------------------------------------------------------------------------
void Performance_With_SPM(NET* Net0 , NET* Net1 , int Epochs)
{
  int  Line , a , i ;

  double Output0[MNN_Number_Of_Outputs];
  double Output1[RNN_Number_Of_Outputs];

 // SPM_Double Array (Ignore -1)
  for ( i =0;  i <Traffic_Dimensions;  i ++) {
    if (SPM_Double [i]!= -1)
       {
        Output_Max[1][i] = SPM_Integer [i];
             Output_Min[1][i] = SPM_Integer [i];
       }
  }
  for (Line=Training_Start_Pointer; Line<Test_End_Pointer; Line++) {

     Restore_Weights(Net1 , 1);
     for (a=0; a<Epochs; a++) {
          Simulate_Neural_Network(Net1 , Line , Output1 , 1 , 1 , true );
     }
     Copy_Weights  (Net0 , Net1 , 0);
     Simulate_Neural_Network(Net0 , Line , Output0 , 0 , 0 , false );

     for ( i =0; i <Number_Of_Outputs[0]; i ++)
     {
         Interval_B[i] = De_Normalize_Output(Output0[i] , i , 0);
         Data_In_Double[Line][2*Traffic_Dimensions+  i]= Output0[i];
        }
        Adapt_Intervals();
        Calc_Metrics(Line);
        Adapt_Next_Traffic_Loads(Line , 0);
        for ( i =0; i <Traffic_Dimensions; i ++)
        {
           Performance_Metrics_Totals[i] += Performance_Metrics[i];
        }
        Push_Text_Line(Line);
        if (Line==Test_End_Pointer-1)
           Push_Totals_Line("After" , Epochs);
      }
      for ( i =0; i <Traffic_Dimensions; i ++)
      {
         Performance_Metrics_Totals[i] = 0;
      }
}
//-----------------------------------------------------------------------
int _tmain(int argc , _TCHAR* argv[])
{
  int SoFar=0;
  NET  Net[2];
  int Stop;
  int Which_NN;
  int Increasing=0;
  double MinTestError[2];
  InitializeRandoms();
  InitializeApplication();

  for(Which_NN=0;Which_NN<=1;Which_NN++)
  {
    Generate_Neural_Network(&Net[Which_NN] , Which_NN);
    Set_Random_Weights(&Net[Which_NN] , Which_NN);
    InitializeNetwork(&Net[Which_NN] , Which_NN);
    MinTestError[Which_NN] = double(Max_Double);
  }
  Stop = 0;

  do {
      SoFar++;
        if (SoFar==524)
          SoFar = SoFar;

        Train_Neural_Network(&Net[0] , &Net[1] , 100   );
        Test_Neural_Network (&Net[0] , &Net[1]         );

        Increasing++;

        printf("\nNetwork Trainming.....%3d , Error#0= %f , Error#0= %f..." ,
             SoFar , TestError[0] , TestError[1]);
        if ( TestError[0] < MinTestError[0]  &&  TestError[1] <
           MinTestError[1])
        {
              Save_Weights(&Net[0] , 0);
              Save_Weights(&Net[1] , 1);
        }

        if ( MinTestError[0] - TestError[0] > 0.001 && MinTestError[1] -
             TestError[1] > 0.001 )
        {
         Increasing = 0;
```

```
            fprintf(File_NN_Performance , " - saving Weights ...");
            MinTestError[0] = TestError[0];
            MinTestError[1] = TestError[1];
        }
        else
            if  ( Increasing >= 3 ||
                  (MinTestError[0] - TestError[0] <= 0.001 && MinTestError[0] -
                  TestError[0] > 0.0)|| (MinTestError[1] - TestError[1] <= 0.001
                  && MinTestError[1] - TestError[1] > 0.0)    )
            {
                fprintf(File_NN_Performance , " - stopping Training && restoring
                        Weights ...");
                Stop = 1;
                Restore_Weights(&Net[0] , 0);
                Restore_Weights(&Net[1] , 1);
            }
    } while (! Stop);

    Performance_Without_SPM(&Net[1]);
    fprintf(File_Outputs , "\nPerformance Before doing Plan #1");
    fprintf(File_Outputs , "\n------------------------------\n");
    Test_Network_Performance(&Net[0]);
    fprintf(File_Outputs , "\nPerformance After doing Plan #1");
    fprintf(File_Outputs , "\n------------------------------\n");
    fprintf(File_Totals , "\n DMSS      ");
    Performance_With_SPM(&Net[0] , &Net[1] , 99);
    FinalizeApplication();
    return 0;
}
```