ADDIS ABABA UNIVERSITY

SCHOOL OF GRADUATE STUDIES

ADDIS ABABA INSTITUTE OF TECHNOLOGY (AAiT)

ELECTRICAL & COMPUTER ENGINEERING DEPARTMENT

A GENERAL PYRAMIDAL MODULAR NEURAL NETWORK ARCHITECTURE FOR HIGH DIMENSIONAL INPUT VECTORS

By

Menore Tekeba Mengistu

September 2011

Addis Ababa, Ethiopia

ADDIS ABABA UNIVERSITY

SCHOOL OF GRADUATE STUDIES

ADDIS ABABA INSTITUTE OF TECHNOLOGY (AAiT)

ELECTRICAL & COMPUTER ENGINEERING DEPARTMENT

A General Pyramidal Modular Neural Network Architecture for High Dimensional Input Vectors

By

Menore Tekeba Mengistu

Advisor

Dr. Kumudha Raimond

A thesis submitted to the school of Graduate studies of Addis Ababa University in partial fulfillment of the requirements for the degree of Masters of Science in Computer Engineering

September 2011

Addis Ababa, Ethiopia

ADDIS ABABA UNIVERSITY

SCHOOL OF GRADUATE STUDIES

A GENERAL PYRAMIDAL MODULAR NEURAL NETWORK ARCHITECTURE FOR HIGH DIMENSIONAL INPUT VETCTORS

By

Menore Tekeba Mengistu

ADDIS ABABA INSTITUTE OF TECHNOLOGY

APPROVAL BY BOARD OF EXAMINERS

Dr. Getahun Mekuria ________________

Chairman, Dept. of Graduate Committee Signature

Dr. Kumudha Raimond ________________

Advisor Signature

______________________________ __________________

Internal Examiner Signature

______________________________ __________________

External Examiner Signature

Declaration

I, the undersigned, declare that this thesis work is my original work, has not been presented for a degree in this or any other universities, and all sources of materials used for the thesis work have been fully acknowledged.

Menore Tekeba Mengistu ______________________________

Name signature

Place: Addis Ababa

____

Date of submission

This thesis has been submitted for examination with my approval as a university advisor.

Dr. Kumudha Raimond ____________________________

Advisor's name signature

Acknowledgement

As wisdom is from God, praise and gratitude be given to the **Almighty God** who has nourished me with His mercy and wisdom.

The second sincere thank goes to my advisor and supervisor ***Dr. Kumudha Raimond*** for her continuous guidance, critics, suggestions, motivating advices and all the things she gave me during this thesis work as well as during the course works she has given me that helped me to see what I can do.

In third place, many thank goes to my dear sister, *Abaynesh Dagnachew*, who has helped me in her advice and every aspect of my life during this thesis work. Without her, it would have been very difficult to my thesis work to get completed.

Last but not least, my sincere thanks go to the Department head, *Dr. Getahun Mekuria* and all my staff members of Electrical and Computer Engineering Department. I have seen that my personal concern was also theirs and they all helped me in encouraging and reminding me to work hard.

Contents

Contents....IV
Acronyms and Abbreviations....VIII
Terminology....IX
Abstract....X
Chapter One....1
1. Introduction....1
1.1. Introduction....1
1.2. Background and History....2
Efficiency....3
Training....3
Robustness....3
Generalization Ability....3
1.3. Statement of the Problem....4
1.4. Objective....4
General Objective....4
Specific Objectives....4
1.5. Methodology....5
1.6. Scope....5
1.7. Contribution....6
1.8. Thesis Outline....6
Chapter Two....8
2. Literature Review....8
2.1. Introduction....8
2.2. Landmarks of the concepts of Modular Neural Network Development....8
1. The Concept of Modularity....8
2. On the Network Topology of MNNs....9
2.3. Specific Problem Applications solved using MNNs....12
Chapter Three....14
3. Theoretical Background....14
3.1. Feed-ward MLP and Back Propagation Learning Algorithm....14

3.1.1 An Artificial Neuron 14
3.1.2 A Single Layer Network 15
3.1.3. Multilayer Neural Network 16
3.2. Modularity 19
3.1.1. Modularity in the Nervous System 20
3.2.2 Modular Artificial Neural Networks 21
3.3. Particle Swarm Optimization 22
3.3.1. Introduction 22
3.3.2. Background 22
3.3.3. The Algorithm 23
Chapter Four 25
4. The Design of PMNN 25
4.1. The Evolution of the New Architecture 25
4.2. The Proposed Architecture 31
4.2.1. Design of Input Layer of PMNN 31
4.2.2. Design of Hidden Layers of PMNN 33
4.2.2. Design of Decision Module of PMNN 33
4.2.3. Design Assumptions for both module types 33
4.3. Topology Selection 33
4.3.1 Topology Selection for Overall Architecture 35
4.3.2. Topology Selection for Modules 36
4.4. Training the System 37
4.4.1. The Training Algorithm: 39
4.4. Calculation of Output 41
4.5. Analysis of the New Architecture 41
4.5.1. Training Time and Speed 41
4.5.2. Learning Problems 42
4.5.2. Generalization Ability 43
Chapter 5 44
5. Implementation of the Proposed Architecture 44
5.1. Considerations 44

5.2. The Software Platform ... 45
5.3. Data Preprocessing ... 45
5.4. The Program Structure ... 45
5.4.1. For Evolving the Topology of the Network ... 45
5.4.2. Modular Weight Training Using Back Propagation Algorithm ... 47
Chapter 6 ... 49
6. Experimental Evaluation ... 49
6.1. Introduction ... 49
6.2. Test Methodology ... 50
6.3. Pattern Recognition Problems ... 50
6.3.1. PSO Algorithm in Topology of the PMNN and the modules ... 51
6.3.2. Palm Print Recognition ... 53
6.3.3. Iris Pattern Recognition ... 56
6.3.4. Face Recognition ... 59
6.4. Summary of the Evaluation ... 61
6.5. The Proposed Network for Other High Dimensional Input Vectors ... 63
Chapter 7 ... 64
7. Conclusions and Recommendations ... 64
7.1. Conclusions ... 64
7.2. Future Works and Recommendations ... 65
Appendix A: Code Sample for PSO Algorithm of Module Topology ... 67
Appendix B: Code Sample for PSO Algorithm of Pyramidal Topology ... 76
8. References ... 85

List of Tables

No table entries in the document.

Table of Figures

Figure 2.1: Modular Neural Network Architecture 10

Figure 3.1: An Artificial Neuron 15

Figure 3.2: Example of Multilayer Neural Network Architecture 16

Figure 3.3: The Back Propagation Network 17

Figure 3.4: An Example of MNN Network 22

Figure 4.1: The Block Diagram of General Steps followed to Design the PMNN Architecture 25

Figure 4.2: Examples of Modular Neural Network Architecture 29

Figure 4.3: The proposed Pyramidal Architecture 30

Figure 4.4: PSO based Self-Scaling Pruning-Constructive Algorithm for Pyramidal MNN Development. 37

Figure 5.1: The PSO Algorithm Structure used to search the overall topology and the topology of the modules 47

Figure 5.2: The Back Propagation Training Algorithm structure of the proposed architecture 48

Figure 6.1: Progressive result of the Pyramidal Network for Palm-print as layer increases 54

Figure 6.2: Eight Typical Palm Print Images in the database for one subject 54

Figure 6.3: The results of monolithic MLP, the two layer architecture with input layer modules and the proposed Pyramidal Architecture with palm print recognition 55

Figure 6.4: Progressive result of the Pyramidal Network for Iris Recognition as layer increases 57

Figure 6.5: The Proposed Pyramidal Network Architecture Results on Iris and Face Recognition 58

Figure 6.6: Six Typical Iris Images from the database for one subject 58

Figure 6.7: Progressive result of the Pyramidal Network for Face Recognition as layer increases 60

Figure 6.8: Five Typical cropped face images 60

Acronyms and Abbreviations

ACO: Ant colony optimization

AI: Artificial Intelligence

ANN: Artificial Neural Network

BPA: Back Propagation Algorithm

CALM: Categorizing And Learning Module

CNS: Central Nervous System

GA: Genetic Algorithm

MLP: Multi-Layer Perceptron

MNN: Modular Neural Network

NN: Neural Network

PMNN: Pyramidal Modular Neural Network

PSO: Particle Swarm Optimization

Terminology

Pyramidal Neural Network (PMNN): the type of Modular Neural Network Architecture having two or more number of layers of modules with more number of modules in the input layer and less number of modules in subsequent layers. Its name is derived from its shape that it has wider base from the input side and getting narrower on the output side.

Component Modules: The Multi-Layer Perceptron (MLP) modules trained using Back Propagation Algorithm that composes the Layers of the Pyramidal Modular Neural Network.

Decision Module: The MLP module that accepts the outputs of the last hidden layer of the PMNN modules and gives out the final classes output.

N-Layer Architecture: The type of MNN Architecture with relatively equal number of modules in each layer of the network.

Abstract

In this thesis new modular neural network (MNN) architecture is proposed. The basic building blocks of the architecture are small multilayer feed forward networks trained using the Back propagation algorithm (BPA). The newly proposed MNN Architecture is called Pyramidal MNN (PMNN). It is called Pyramidal for the number of the modules that constitutes the layers of network relatively decreases from the input layer to the output layer.

An Optimization technique called PSO has been used to optimize the topology of the proposed PMNN architecture for typical high dimensional input vector datasets. The optimization technique is used to suit the PMNN architecture for specific problems of high dimensional input vectors depending on the nature of the data input and the nature of the problem. This is done by evolving topology of the modules that constitutes the network and changing the architecture of the overall network to suit the new data set.

The suggested training algorithm works in multiple stages depending on the number of hidden layers of the network. The training of modules in the same layer of the PMNN is easy to implement in parallel. Since the network is not fully connected, the number of weight of connections is less and hence the training is very quick for large input dimensional vectors.

An object-oriented implementation of the proposed PMNN architecture is written to simulate the behavior. The evaluation and optimization of the PMNN architecture for different real world applications is carried out to show the effectiveness of the proposed architecture for high dimensional input vector applications. The evaluation is based on three pattern recognition problems: palm-print recognition, iris recognition and face recognition. In all the three evaluations, it has achieved more than 95% accuracy of the test results. Furthermore, the proposed PMNN architecture performs better than other similar type research works. It is shown that as PMNN is a huge family of several specific architectures, this proposed topology of the neural net can serve wide range of complex domain problems that need to be solved using Artificial Intelligence (AI).

Keywords: Modular Neural Networks, Pyramidal Modular Neural Networks, Particle Swarm Optimization, High Dimensional Input Vectors, General Pyramidal Modular Neural Networks, Modularity in Neural Networks

Chapter One

1. Introduction

1.1. Introduction

MNN is a recent development in the area of AI. The results of the different applications involving MNNs lead to the general evidence that the use of MNNs implies a significant learning improvement comparatively to a single Neural Network (NN) architecture and especially to the back propagation Neural Network. It has special pronounced advantage of high learning rate and additional generalization ability for high dimensional input vectors as indicated in [1].

PMNN is the type of MNN Architecture having two or more number of layers of modules with more number of modules in the input layer and less number of modules in subsequent layers. Its name is derived from its shape that it has wider base from the input side and getting narrower on the output side. There is also some architecture with similar topology. An N-Layer Architecture is the type which has more or less the same number of modules in all layers. But here in this thesis, only PMNN is considered to be dealt with.

To the best of researcher's knowledge, there is no other research work that is proposed and evaluated PMNN architecture. However, similar to PMNN has been dealt in [1] with just two layers. But this research work [1] has not dealt with the general type (only two layers with fixed architecture) and has not also included optimization techniques for the proposed network as well.

The development of new PMNN Architecture for typical high dimensional input datasets is presented in this thesis. The proposed Architecture serves as the general architecture of MNN topology and the means of optimizing this topology for new high dimensional datasets is included.

Although there have been specific problem solving MNN architectures that have been used to solve very specific pattern recognition purposes, there have not been a general type of topology of the network to serve the above mentioned purpose. A new MNN Architecture has been proposed by different researchers at different times as in [1] and [2] and they have worked on different attributes of those topologies. But as they also have pointed out in their research works,

there have been many directions and unexplored areas of new networks and attributes of the already proposed networks as recommended by [1] and [2] in their respective topics of future works.

In this thesis the existence of many best optimized solutions in network topology of the general PMNN architecture for different problems depending on the nature of the problem are explored. Each specific problem finds its best optimized network topology from the large search space of the given general pyramidal network topology and the means of searching the best solution for a typical high dimensional input data is well done.

In this thesis typical homogeneous network architecture is proposed based on a typical high dimensional input dataset. The pattern recognition problems using palm-print recognition, face recognition and iris recognition are shown. Then an effective means of optimizing the architecture for new dataset is also provided. The optimization process of the network topology for the given dataset is done by using Particle Swarm Optimization (PSO) in evolving the number of layers of PMNN, the number of modules in each layer of the PMNN, the detailed structure of these modules. Then the number of weights of connections and the weight values of connections for the modules is done using BPA.

1.2. Background and History

An MNN is an NN characterized by a series of independent neural networks moderated by some intermediary. Each independent NN serves as a module and operates on separate inputs to accomplish some subtask of the complex task the network is designed to perform [2]. As artificial neural network (ANN) research progresses, it is appropriate that ANNs continue to draw on their biological inspiration and emulate the segmentation and modularization found in the brain. The brain, for example, divides the complex task of visual perception into many subtasks [5].

Certainly some tasks that the brain handles, like vision, have a hierarchy of modules. However, it is not clear whether there is some intermediary which ties these separate processes together on a grander scale. Rather, as the tasks grow more abstract, the isolation and compartmentalization breaks down between the modules and they begin to communicate back and forth. [16].

One of the major benefits of a MNN is the ability to reduce a large, unwieldy NN to smaller, more manageable components [1]. There are some tasks it appears are for practical purposes

intractable for a single NN as its size increases. The following are benefits of using a MNN over a single all-encompassing NN [16].

Efficiency

The possible connections increases at a daunting rate as nodes are added to the network. Since computation time depends on the number of nodes and their connections, any increase here will have drastic consequences in the processing time. As the greater task is further compartmentalized, the possible connections each node can make are limited, and the subtasks will hopefully execute more efficiently than trying to tackle the whole task at once.

Training

A large NN attempting to model multiple parameters can suffer from interference as new data can dramatically alter existing connections or just serve to confuse. With some foresight into the subtasks to be solved, each NN can be tailored for its task. This means the training algorithm used, and the training data used for each module can be unique and implemented much more quickly. In large part this is due to the possible combinations of interesting factors diminishing as the number of inputs decreases.

Robustness

Regardless of whether a large NN is biological or artificial, it remains largely susceptible to interference at and failure in any one of its nodes. By compartmentalizing subtasks, failure and interference are much more readily diagnosed and their effects on other modules are eliminated as each one is independent of the other.

Generalization Ability

In addition to the above mentioned benefits, the thesis also targeted to drive an additional generalization ability derived from the connectionist approach and logical network for high dimensional input vectors [1].

The development of a typical general PMNN using a typical high dimensional dataset and employs PSO to optimize the general topology of the network to suit for specific problem in

particular. This enables us to find the best possible solution of a problem from the available homogenous general topology evaluated using the fitness (cost) function.

1.3. Statement of the Problem

As it is clearly discussed in section 1.2 above, the development of MNN is a recent development having those stated advantages over the monolithic NN Architecture. Efficiency, Training, Robustness, and Generalization ability are the advantages of MNN over the monolithic one.

Different research works have been done on MNN so far on network topology and specific applications. Although some architectures are proposed and evaluated by different researchers, these proposed architectures are not evaluated with respect to all the network parameters, like training algorithms used, the types of connections, the number and types of modules used, the optimization techniques applied. Additionally, all possible network architectures are not even proposed and evaluated. There are also specific applications using MNN. But these developments cannot serve as a general architecture for new type of applications. It is too specific and only designed to address one application.

The purpose of this thesis is to develop and examine a general PMNN. This pyramidal topology of MNN can serve as the reference and the key solution for high dimensional datasets and problems. It is consisted of MLPs with BPA Training Algorithm. Similar but without any optimization technique and with only two Layers of PMNN is done by [1]. Here the advantage of this thesis work is to widen the search space of the solutions for different applications with high dimensional inputs. Furthermore, the thesis here also addresses the technique of optimizing this proposed network for entirely new and high dimensional input vectors to identify an optimal topology of the overall architecture as well as the topology of modules that constitute the PMNN.

1.4. Objective

General Objective

The general objective of this work is to propose and examine a new general PMNN architecture for high dimensional input vectors.

Specific Objectives

It has the following specific objectives:

- Investigate the existing architectures of MNNs and propose a general pyramidal and homogeneous MNN having MLP modules as an element.
- Investigate the existing optimization techniques and select an appropriate one. Using the selected technique, optimize the modular and global network parameters
- Simulate and evaluate the network Architecture using pattern recognition problems. Compare the proposed architecture with monolithic MLP and the two-layer MNN architecture in [1].

1.5. Methodology

The methodologies used in this thesis are given below.

❖ **Literature Survey:** Literature survey of theoretical bases and different existing research works on MNNs.

❖ **Propose the Pyramidal MNN Architecture:** By investigating the existing MNN Network Architecture, new PMNN Architecture is proposed.

❖ **Selecting Optimization Technique and optimize the network:** Investigating existing optimization techniques of MNN, an appropriate optimization technique is selected. Then the proposed network architecture is optimized for a typical high dimensional input dataset.

❖ **Simulation & Performance Evaluation**: A simulating object oriented program is written and the network is simulated and evaluated selecting typical high dimensional input datasets.

1.6. Scope

The scope of this thesis is to propose PMNN Architecture and add optimization technique to suit this Architecture for different types of applications of high dimensional input datasets. The proposed network topology is evaluated with self-developed object oriented simulating program with selected high dimensional dataset applications. Additionally, this thesis has compared the

results of the proposed PMNN architecture with monolithic neural network architecture and with two-layer similar architecture proposed by [1].

1.7. Contribution

The contributions of this thesis are given below:

- Widen the search space of solutions for applications of high dimensional input vectors. The number of layers in the PMNN, the number of modules in each layer of the PMNN, the topology of the modules that constitute the PMNN can be set with applied optimization technique. So there is a wide search space to find the optimal solution of an application in this PMNN network compared to other similar type of works.

- Working on new optimization technique to suit the network for new application of high input dataset. The architecture is general type and it can be used for wide range of high dimensional input datasets. The use of the optimization technique applying PSO enabled this general PMNN architecture to be used for different and wide range of high dimensional input data applications.

- A new approach of using high dimensional input vectors without applying dimension reduction techniques and feature selection has been proposed with a very good experimental evaluation result.

1.8. Thesis Outline

The thesis has seven chapters arranged in the following manner:

- Chapter One - Introduction: Gives the overall introduction and descriptions about the thesis.

- Chapter Two – Literature Review: Brief explanations about the literatures of MNNs and its development.

- Chapter Three – Theoretical Background of the thesis: Discusses the theoretical bases of the thesis and fundamental concepts used in the thesis.

- Chapter Four – Design of PMNN: It discusses design steps and methods followed and also different design benchmarks that need to be demarked. Finally it also discusses the procedures and assumptions followed during evolving the network and training each module.
- Chapter Five – Implementation: This chapter discusses the implementation of the simulating program. The overall program structure and program components are discussed here.
- Chapter Six – Experimental Evaluation: This chapter discusses the experimental results of the simulation on different selected high dimensional datasets. It also compares the result of this network with the result of monolithic NN Architecture and formerly developed two layers Architecture by [1].
- Chapter Seven – Conclusion and Recommendations: The conclusions on the thesis work and the result of the experimentation is given in this chapter. It also gives the recommendations on future works on the same Architecture.

Finally, the thesis comes to an end with Appendices and References used in the thesis.

Chapter Two

2. Literature Review

2.1. Introduction

There are a number of researches made on Modular Network. Since it is difficult to cover all the researches, more important research works are compiled here. Many researchers have worked to improve different general structure, topologies, learning algorithms, to incorporate the neurobiological, psycho physiological modularity while others have worked to solve specific problems by using the concept of modularity and other AI concepts. Here it is planned to compile the research works that worked towards the general development of MNN and can serve as landmarks of the concepts of modularity in neural networks in the first part and to present the works of research works that worked to solve specific problems in the second part.

2.2. Landmarks of the concepts of Modular Neural Network Development

In the history of AI, the concept of modularity is taken from the modular and functional parts of anatomy, psycho-physiology, neurobiology, and brain. There are different arguments that have been developed by researchers for more useful features of modularity specially to handle complex tasks. Here under this subtopic, the following concepts are dealt as reviewed from different research works.

1. The Concept of Modularity

As it is briefly sketched above, several lines of evidence indicate that a modular organization of the brain exists at different anatomical scales. This modular organization is paralleled at the functional level of information processing, as has become evident from a wide range of psychological studies [5]. The research work in [5] argues the existence of modularity in different anatomical scales by presenting the parallel processing of different anatomical structures. For example, humans are not in difficulty to listen while driving the car [5]. According to the research work in [5], some tasks are processed in distinct streams of modules

and do not interfere with each other. Other tasks require simultaneous access to a single module and are, therefore, much more prone to mutual interference. It also drives an important argument that the nature of information processing in the brain is modular. Individual functions are broken up into sub-process which can be executed in separate modules without mutual interference [5].

The thesis work in [1] also concludes that modularity is a very important concept in nature. It also defines modularity as subdivision of a complex object into simpler objects. The subdivision is determined either by the structure or function of the object and its subparts [1]. It also gives that replication and decomposition are the two main concepts for modularity [1]. It discusses the presence of modularity in nervous system especially in brain. It explores modularity derived from psychology and presents the parallel processing capability of different tasks in humans as an example.

The concept of modularity for computational systems in [2] is defined as a computational system which can be considered to have a modular architecture if it can be split into two or more subsystems in which each individual subsystem evaluates either distinct inputs or the same inputs without communicating with other subsystems. The overall output of the modular system depends on an integration unit which accepts outputs of the individual subsystems as its inputs and combines them in a predefined fashion to produce the overall output of the system [2]. It also presents some characteristics of modules. After having given some introducing concepts on modularity, it also presented modularity in neural networks and the motivation behind using the concept of modularity in neural networks. It has listed those motivating factors to use MNNs rather than the monolithic neural networks.

2. On the Network Topology of MNNs

Several network topologies of MNNs have been developed and examined by different researchers. Some of those networks are discussed depending on the closeness of the work to be addressed by this thesis.

The research work showed in [5] has been based on some previously developed module (CALM) as the element of its modular network and has used evolutionary method to evolve the specific network required. CALM stands for Categorizing And Learning Module [5]. It has used Genetic Algorithm (GA) for an autonomous and effective search mechanism to find effective NN

architecture [5]. Both the modular structure of a network and a global set of parameters associated with the activation and learning dynamics of CALM were defined by the GA for a chosen categorization of unconstrained, handwritten digits as a case-study [5].

The other research work dealing with MNN Architectures was presented in [1]. It uses forward Multi-Layer Perceptron (MLP) with BPA as modular units of its new architecture [1]. It has dealt with a specific case of general PMNN architecture with only one input layer of modules and one decision (output) module. It has also used a two stage parallel BPA for input and output layers [1]. The network developed by [1] is shown in Fig.2.1.

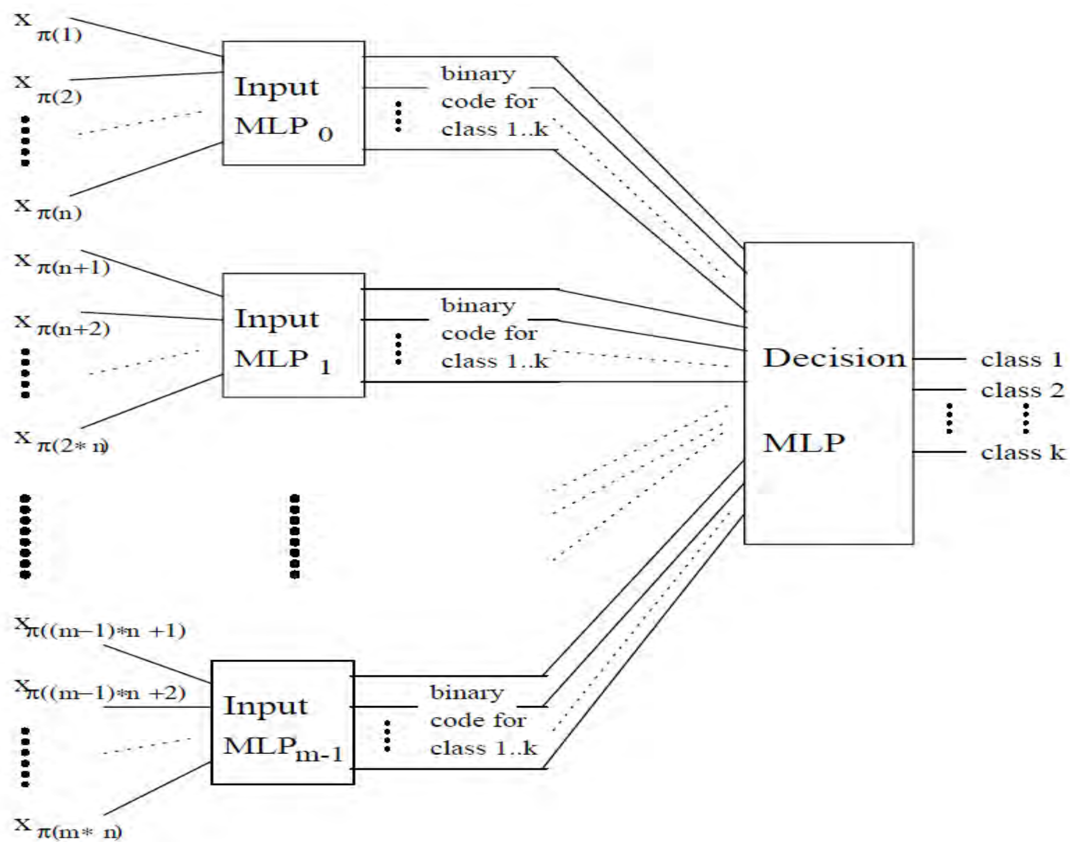

Figure 2.1: Modular Neural Network Architecture

In [1], training occurs in two stages using the BPA. In the first phase, all modules in the input layer of MNN are trained. In the second stage the decision network is trained. The research work argues the importance of the new network for fast learning, more generalization ability for high dimensional input space. But it has also pointed out the existence of limitations due to equal division of input space to the input layer modules [1].

The other research work which dealt with the topology of MNNs is [2]. It has argued first the importance of modularity as stated above and showed different approaches to design the architecture of an MNN. It has presented that the most important issue in designing MNNs is the definition of the modular neural architecture at the abstract level such that it does not lose its relevancy to the application at hand, biological/cognitive adherence and theoretical analysis [2].

According to the approach followed in [2], the task of designing a modular network can essentially be broken down into two major subtasks; firstly, the determination of the optimal number and form of the operating regimes for task subdivision and secondly, the implementation of a methodology by which the individual local modules are integrated to produce an optimal global NN structure. The advantages of a modular structure of a NN cannot be achieved to the fullest if either of these two steps is not implemented optimally [2].

After it has dealt different issues related to the design issues of MNNs, [2] presents task decomposition, structural modularization, training algorithm, combination of specialist modules as the important issues related to the design and implementation of MNNs. It has also presented how self-scaling neural networks can be developed in one chapter. Self-scaling MNN architectures are a class of neural networks that scale their structure to match the complexity of the given learning task in an incremental fashion [2]. It deals the two types of self-scaling mechanisms: the vertical and hierarchical self-scaling neural networks. It has presented the algorithms used for each type of self-scaling types. It has dealt different topologies of MNNs and discussed wide ranges of issues about MNNs in its entire dissertation works.

The research work in [9] deals with proposing some general network topology with three layers. The first layer known as the allocator accepts the input data, determines which modules to be involved and allocates to different modules of the second layer [9]. Generally, a number of modules may be selected to accomplish the given task. Each of the selected modules output the result based on local computation [9]. At third layer known as the coordinator then collects all the results of the selected modules and gives the final output [9]. The general topology is then evolutionally developed through three steps: splitting nodes, determining which nodes are terminal nodes and assigning class labels [9]. It uses simple GA for evolving the network [9].

2.3. Specific Problem Applications solved using MNNs.

The research work presented in [3] addressed specific problems: for pattern recognition of human faces and finger prints. It uses MNNs with a fuzzy logic method for response integration. For the case of human face recognition, it divides the face into three regions and gives to three modules to be processed and finally the outputs from each module was integrated using fuzzy logic. Similar approach was followed for finger print recognition in [3].
The research work carried out in [4] is very similar to [3]. But [4] has included an optimization technique for the proposed topology applied for speaker recognition. It has used hierarchical GA to code the MNN architecture for the specific problem of speaker recognition since MNN topology information is hierarchical, i.e. it need to manage information about the number of modules, number of layers and nodes of the network [4]. These two works claim that the training time as well as the result of the network is far better than the monolithic network architecture.

The research work presented in [7] uses the hybrid technique to forecast the dynamic behavior of complex economic time series. It uses several NN architectures and training algorithms to compare the results and decide at the end, which one is best for the application [7].

The research work presented in [41] is the type of MNN which is used to detect operational problems on urban arterials for Advanced Transportation Management Systems. This thesis dissertation in [41] has applied model selection techniques by exploring several different architectures and a comprehensive system was finally proposed to address the problem of operational problems. Input feature selection has also been applied based data available in traffic management center. The research work has focused on detecting three operational problems: lane blocking incidents, special events and malfunctioning. These different types of operational problems are detected on different modules. The data to be fed here is first preprocessed and then fed to the respective modules. Then the outputs of the modules with different subtasks are then integrated and the incident state is given as output.

The research work in [43] is using MNNs to give stock market predictions. It has three parts. The first part is the preprocessor of the input data from market time series. The second part is consists of the modules of neural networks. Then the outputs of these modules that constitute the second part are post processed by the third part. The modules used here are MLPs. Each MLP

has the input layer, the hidden layer and the output layer. This research work has indicated that MNNs can serve other application areas in addition to pattern recognition and control. It has used new learning algorithm known as supplementary learning algorithm which is a modified BPA with learning constants being changed in each epoch. [43]

Chapter Three

3. Theoretical Background

3.1. Feed-ward MLP and Back Propagation Learning Algorithm

Multilayer feed-forward network is one of the milestones of research in ANN. In this research work such networks are used as basic modules in a more complex structure. These multilayer feed-ward networks are in turn consists of artificial neurons as building blocks.

3.1.1 An Artificial Neuron

The artificial neuron shown in Figure 3.1 is a very simple processing unit, which is an abstract model of the biological neuron. The neuron has a fixed number of inputs n; each input is connected to the neuron by a weighted link wi. The neuron sums up the net input according to the equation 3.1:

$$net = \sum_{i=1}^{n} wi * xi \qquad \text{.......} 3.1$$

To calculate the output, an activation function f is applied to the net input of the neuron. This function is either a simple threshold function or a continuous non linear function. Two often used activation functions are unipolar sigmoid activation function as shown in equation 3.2 and the threshold activation function as shown in equation 3.3.

$$fc(net) = \frac{1}{1 - e^{-net}} \qquad \text{.......} 3.2$$

$$fc(net) = \begin{cases} 1, if\ net > \theta \\ 0, if\ net \le \theta \end{cases} \qquad \text{.......} 3.3$$

The strength of a connection is coded in the weight. The intensity of the input signal is modeled by using a real number instead of a temporal summation of spikes. The artificial neuron works in discrete time steps; the inputs are read and processed at one moment in time.

There are many different learning methods possible for a single neuron. Most of the supervised methods are based on the idea of changing the weight in a direction that the difference between

the calculated output and the desired output is decreased. Examples of such rules are the Perceptron Learning Rule, the Widrow-Hoff Learning Rule, and the Gradient descent Learning Rule.

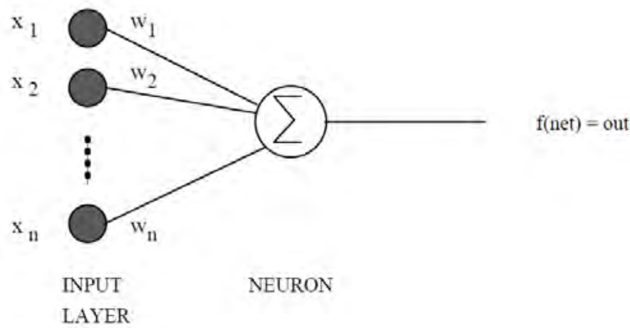

Figure 3.1: An Artificial Neuron

The Gradient descent Learning Rule operates on a differentiable activation function. The weight updates are a function of the input vector, x, the calculated output f(net), the derivative of the calculated output f'(net), the desired output d, and the learning constant ŋ.

$$net = X^T * W \quad \text{.......... 3.4}$$

$$\Delta W = n * f'(net) * (d - f(net)) * X \quad \text{.......... 3.5}$$

The delta rule changes the weights to minimize the error. The error is defined by the difference between the calculated output and the desired output. The weights are adjusted for one pattern in one learning step. This process is repeated with the aim to find a weight vector that minimizes the error for the entire training set.

3.1.2 A Single Layer Network

A single layer network is a simple structure consisting of m neurons each having n inputs. The system performs a mapping from the n-dimensional input space to the m-dimensional output space. To train the network the same learning algorithms as for a single neuron can be used.

This type of network is widely used for linear separable problems, but like a neuron, single layer network are not capable of classifying non linear separable data sets. One way to tackle this problem is to use multilayer network architecture.

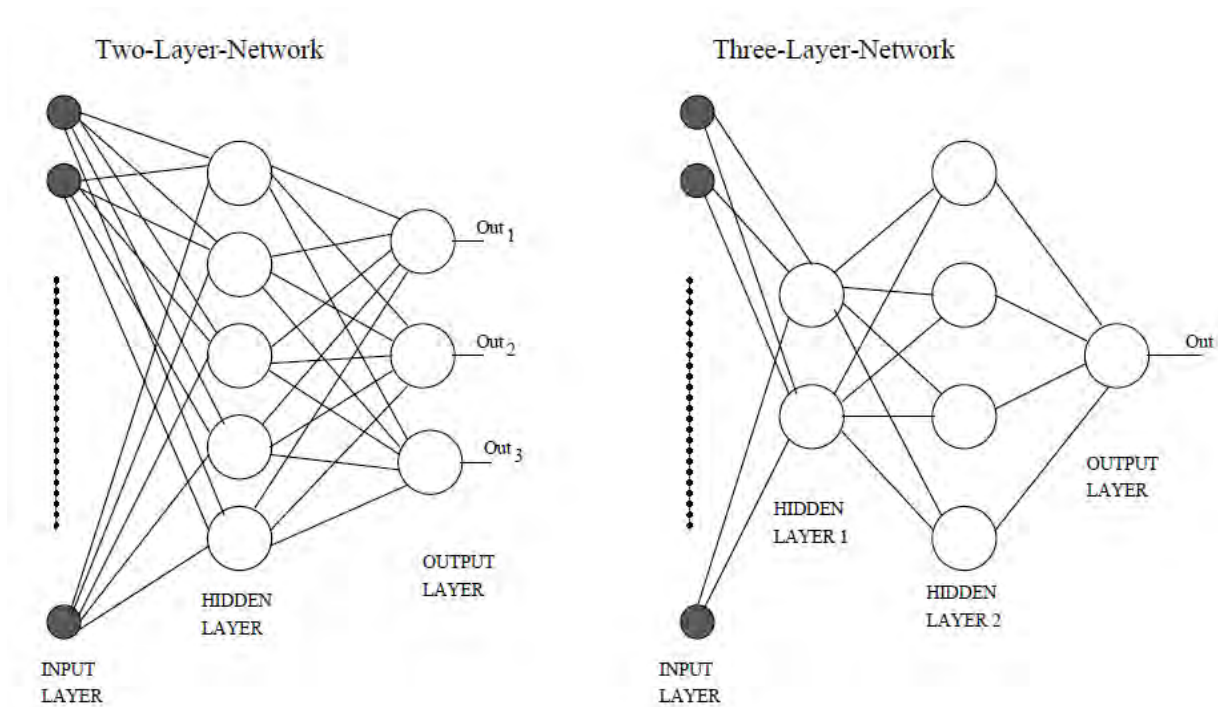

Figure 3.2: Example of Multilayer Neural Network Architecture

3.1.3. Multilayer Neural Network

Multilayer networks solve the classification problem for non linear sets by employing hidden layers, whose input neurons are not directly connected to the output. The additional hidden layers can be interpreted geometrically as additional hyper-planes, which enhance the separation capacity of the network. Figure 3.2 shows typical multilayer network architectures.

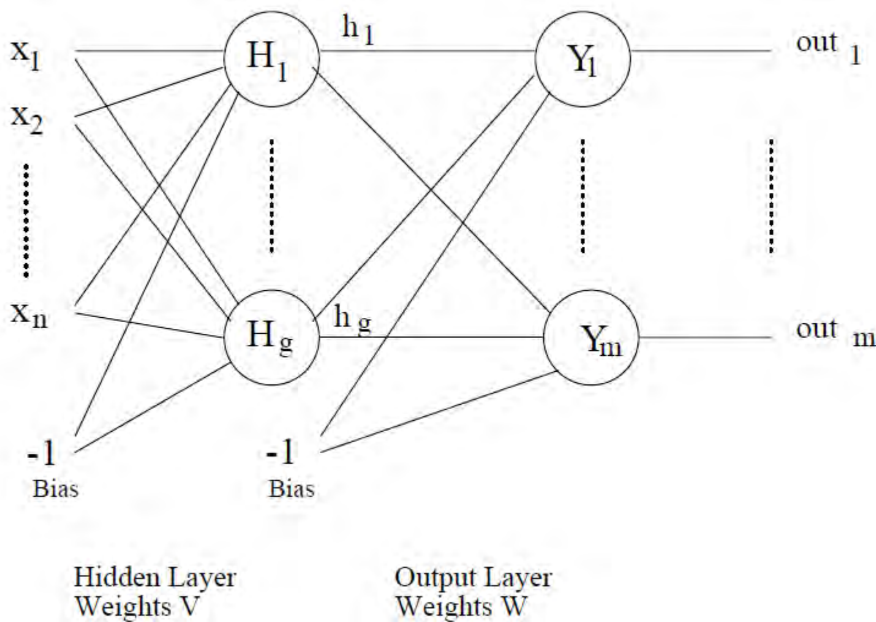

Figure 3.3: The Back Propagation Network

This new architecture introduces a new question: how to train the hidden units for which the desired output is not known. The BPA offers a solution to this problem.

The training occurs in a supervised style. The basic idea is to present the input vector to the network; calculate in the forward direction the output of each layer and the final output of the network. For the output layer the desired values are known and therefore the weights can be adjusted as for a single layer network; in the case of the BPA, weights are adjusted according to the gradient decent rule.

To calculate the weight changes in the hidden layer, the error in the output layer is back-propagated to these layers according to the connecting weights. This process is repeated for each sample in the training set. One cycle through the training set is called an epoch. The number of epochs needed to train the network depends on various parameters, especially on the error calculated in the output layer.

As it is presented in [1] the following description of the BPA is presented.

The assumed architecture is depicted in Figure 3.3 The input vector has n dimensions, the output vector has m dimensions, the bias (the used constant input) is -1, and there is one hidden layer with g neurons. The matrix V holds the weights of the neurons in the hidden layer. The matrix W

defines the weights of the neurons in the output layer. The learning parameter is ŋ and the momentum is α. The used unipolar activation function and its derivative are given by:

$$f(net) = \frac{1}{1+e^{-\partial.net}} \quad \text{.......} 3.6$$

$$f'(net) = \frac{e^{-\partial.net}}{(1+e^{-\partial.net})^2} \quad \text{.......} 3.7$$

The training set consists of pairs where, XP is the input vector and TP is the desired output vector.

T = {(X1, T1), … , (XP,TP)}……………………………………………………3.8

1) Initialize the weights V and W randomly with numbers from a suitable interval, e.g. (-1 to +1). Select the parameters ŋ and α.
2) Randomly take one unmarked pair (X, T) of the training set for the further steps and mark it as used.
3) Do the forward calculation. The notation x' is the input vector x enlarged by the bias, h' is the enlarged hidden layer output vector.

$$net_H = V^T * X^l \quad \text{.......} \mathbf{3.9}$$

$$h_i = f(net_{Hi}) \quad \text{.......} \mathbf{3.10}$$

$$net_y = W^T * h^l \quad \text{.......} \mathbf{3.11}$$

$$out_i = f(net_{yi}) \quad \text{.......} \mathbf{3.12}$$

4) Do the backward calculation.

$$\delta_{outi} = f'(net_{yi}) * (t_i - out_i) \quad \text{.......} \mathbf{3.13}$$

$$\delta_{hi} = f'(net_{Hi}) * \sum_{j=1}^{m} W_{ij} * \delta_{outi} \quad \text{.......} 3.14$$

5) Calculate the weight changes and update the weights.

$$\Delta W^T(t) = n * \delta_{out} * h'^T \quad \text{...3.15}$$

$$\Delta V^T(t) = n * \delta_H * X'^T \quad \text{...3.16}$$

$$W(t+1) = W(t) + \Delta W(t) + \propto * \Delta W(t-1) \quad \text{...3.17}$$

$$V(t+1) = V(t) + \Delta V(t) + \propto * \Delta V(t-1) \quad \text{...3.18}$$

6) REPEAT from 2 while there are unused pairs in the training set.
7) Set all the training set to unused.
8) REPEAT from step 2 UNTIL stop condition is satisfied.

Training continues until the overall error in one training cycle is sufficiently small; this stop condition is given by: Emax = E.

3.2. Modularity

Modularity is a very important concept in nature. Modularity can be defined as subdivision of a complex object into simpler objects. The subdivision is determined either by the structure or function of the object and its subparts.

Modularity can be found everywhere; in living creatures as well as in inanimate objects. The subdivision in less complex objects is often not obvious.

At very basic level electrons, positrons, and neutrons make the building blocks for any matter. At a higher level, atoms of elements are another form of simple modules of which everything is constructed. In living creatures' proteins and on a higher level, cells could be seen as basic components. This idea of modules can be continued to more and more complex structures. Even looking at the universe planets can be seen as modules within a solar system.

Replication and decomposition are the two main concepts for modularity. These concepts are found in concrete objects as well as in thinking. It is often difficult to discriminate sharply between them; replication and decomposition often occur in combination.

Replication is a way of reusing knowledge. Once one module is developed and has proved to be useful it is replicated in a larger number. This principle is often found in living organisms. Observing a human this can be seen in a various scale: two similar legs, fingers, vertebra of

similar structure, thousands of hair modules, and billions of cells. In electronics, the development of integrated circuits is based on replication of simple units to build a complex structure.

3.1.1. Modularity in the Nervous System

The idea of a modular mind is very old. Researchers in brain theory have often tried to locate certain functions in the brain. An early attempt to develop a map of the mind was made by J.G. Spuzheim in 1908 as stated in [1].

The brain is generally viewed as a black box that receives an input in the form of a certain stimulus from the environment and produces a corresponding appropriate output or response. The environmental information is coded as neural signals in the brain. The brain uses this stored neural information in the form of patterns of neural firing to generate the desired actions that are most likely to accomplish the goal at hand. The evidence of modularity in brain function can be attributed to two sources, neuropsychology and neurobiology. [2]

The research indicates that animal and human brains are divided into major parts at the coarse-grain level. The human and animal brains are not comprised of monolithic homogeneous biological neural networks within these major parts, but instead, are comprised of specialized modules performing individual specialized tasks. [2]

The human central nervous system (CNS) can be subdivided into spinal cord, medulla oblongata, pons, midbrain, diencephalon, cerebellum and the two cerebral hemispheres. All these parts have their own functions. Each region is interconnected with other parts of the brain. All these parts have their own functions. Each region is interconnected with other parts of the brain. [1]

The observed modularity in brains is of two types. Structural modularity which is evident from sparse connections between strongly connected neuronal groups (with the trivial example of the two hemispheres of the brain) and/or functional modularity, which is indicated by the fact that neural modules have different neural response patterns, are grouped together. [2]

Along with the brain having a modular structure, it also exhibits a functional and structural hierarchy. Information in the brain is processed in a hierarchical fashion. First, the information is processed by a set of transducers which transform the information into the formats that each specialist modules can process. Specialist modules after processing the data they produce the required information which is suitable for central or domain general processing. [2]

3.2.2 Modular Artificial Neural Networks

As indicated in [2], the obvious advantages of modularity in learning systems, particularly as seen in the existence of the functional and architectural modularity in the brain, has made it a main stream theme in cognitive neuroscience research areas. Specifically, in the field of ANN research, which derives its inspiration from the functioning and structure of the brain, modular design techniques are gaining popularity. The use of MNNs for the purpose of regression and classification can be considered as a competitor to conventional monolithic ANNs, but with more advantages. Two of the most important advantages are a close neurobiological basis and greater flexibility in design and implementation. Another motivation for MNNs is to extend and exploit the capabilities and basic architectures of the more commonly used ANNs that are inherently modular in nature. Monolithic ANNs exhibit a special sort of modularity and can be considered as hierarchically organized systems in which synapses interconnecting the neurons can be considered to be the fundamental level. This level is followed by neurons which subsequently form the layers of neurons of a multi layered neural network. The next natural step to extend the existing level of hierarchical organization of an ANN is to construct an ensemble of neural networks arranged in some modular fashion in which an ANN comprised of multiple layers is considered as a fundamental component. This rationale along with the advances in neurobiological sciences has provided researchers a justification to explore the paradigm of modularity in design and training of NN architectures [2].

As given by [2] with the source indicated, a formal prevalent definition of a MNN is as follows:

Definition 3.1. *A neural network is said to be modular if the computation performed by the network can be decomposed into two or more modules (subsystems) that operate on distinct inputs without communicating with each other. The outputs of the modules are mediated by an integrating unit that is not permitted to feed information back to the modules. In particular, the integrating unit decided both (1) how the modules are combined to form the final output of the system, and (2) which modules should learn which training patterns.*

Having another source, [1] also discusses that the most used ANNs have a monolithic structure. A lot of the models work on fully connected networks or layers (Hopfield or MLP). These networks perform well on a very small input space. However the complexity increases and the

performance decreases rapidly with a growing input dimension [1]. An example of the MNN is given below on Figure 3.4.

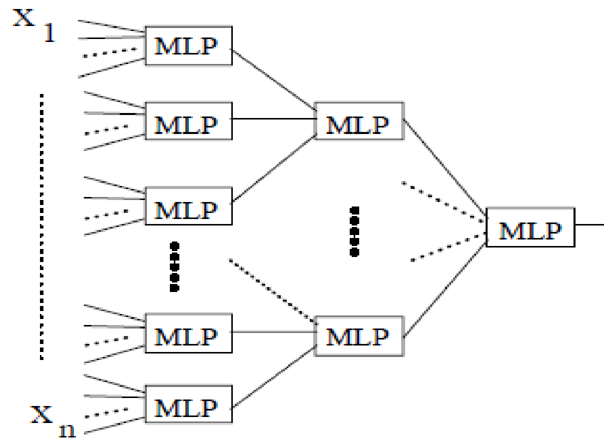

Figure 3.4: An Example of MNN Network

3.3. Particle Swarm Optimization

3.3.1. Introduction

PSO is a population based stochastic optimization technique developed by Dr. Eberhart and Dr. Kennedy in 1995, inspired by social behavior of bird flocking or fish schooling. [38]

PSO shares many similarities with evolutionary computation techniques such as GA. The system is initialized with a population of random solutions and searches for optima by updating generations. However, unlike GA, PSO has no evolution operators such as crossover and mutation. In PSO, the potential solutions, called particles, fly through the problem space by following the current optimum particles. The detailed information is given in following sections.

Compared to GA, the advantages of PSO are that PSO is easy to implement and there are few parameters to adjust. PSO has been successfully applied in many areas: function optimization, ANN training, fuzzy system control, and other areas where GA can be applied. [38]

3.3.2. Background

There are two popular swarm inspired methods in computational intelligence areas: Ant colony optimization (ACO) and PSO. ACO was inspired by the behaviors of ants and has many successful applications in discrete optimization problems. [39]

The particle swarm concept has originated as a simulation of simplified social system. The original intent was to graphically simulate the choreography of bird of a bird block or fish school. However, it was found that particle swarm model can be used as an optimizer. [40]

3.3.3. The Algorithm

As stated before, PSO simulates the behaviors of bird flocking. Suppose consider the following scenario: a group of birds are randomly searching food in an area. There is only one piece of food in the area being searched. All the birds do not know where the food is. But they know how far the food is in each iteration. So what's the best strategy to find the food? The effective one is to follow the bird which is nearest to the food.

PSO learned from the scenario and used it to solve the optimization problems. In PSO, each single solution is a "bird" in the search space. We call it "particle". All of particles have fitness values which are evaluated by the fitness function to be optimized, and have velocities which direct the flying of the particles. The particles fly through the problem space by following the current optimum particles. [38]

PSO is initialized with a group of random particles (solutions) and then searches for optima by updating generations. In every iteration, each particle is updated by following two "best" values. The first one is the best solution (fitness) it has achieved so far. (The fitness value is also stored.) This value is called pbest (particle best). Another "best" value that is tracked by the PSO is the best value obtained so far by any particle in the population. This best value is a global best and called gbest. When a particle takes part of the population as its topological neighbors, the best value is a local best and is called lbest. [38]

After finding the two best values, the particle updates its velocity and positions with following equation 3.7 and 3.8

v[] = v[] + c1 * rand() * (pbest[] - present[]) + c2 * rand() * (gbest[] - present[]) ……..………3.7

present[] = persent[] + v[] …………………….……………………………………….…………..3.8

where v[] is the particle velocity, persent[] is the current particle (solution). pbest[] and gbest[] are defined as stated before. rand () is a random number between (0,1). c1, c2 are learning factors. Usually c1 = c2 = 2. [40]

The pseudo code of the procedure is as follows:

```
For each particle
   Initialize particle
END
Do
   For each particle
      Calculate fitness value
      If the fitness value is better than the best fitness value (pBest) in history
         set current value as the new pBest
   End

   Choose the particle with the best fitness value of all the particles as the gBest
   For each particle
      Calculate particle velocity according to equation 3.7
      Update particle position according to equation 3.8
   End
While maximum iterations or minimum error criteria is not attained
```

Particles' velocities on each dimension are clamped to a maximum velocity Vmax. If the sum of accelerations would cause the velocity on that dimension to exceed Vmax, which is a parameter specified by the user. Then the velocity on that dimension is limited to Vmax. [38]

Evolutionary computation methodologies have been applied to three main attributes of neural networks: network connection weights, network architecture (network topology, transfer function), and network learning algorithms parameters. Therefore, PSO can be applied for these purposes. But when it comes to weight value optimization for this thesis work, PSO cannot be an effective optimization method as it may not converge for high dimensional particle. . Here in this thesis, PSO is used to evolve the network topology of both the overall network of the PMNN and the modules that constitute this PMNN Architecture.

Chapter Four

4. The Design of PMNN

The general objective of this work is to propose and examine a new general PMNN for high dimensional input vectors.

In this chapter the evolution of the architecture is shown. Then the proposed network together with a training algorithm and its optimization technique is described. In the last part of this chapter some aspects of the proposed architecture are analyzed and discussed.

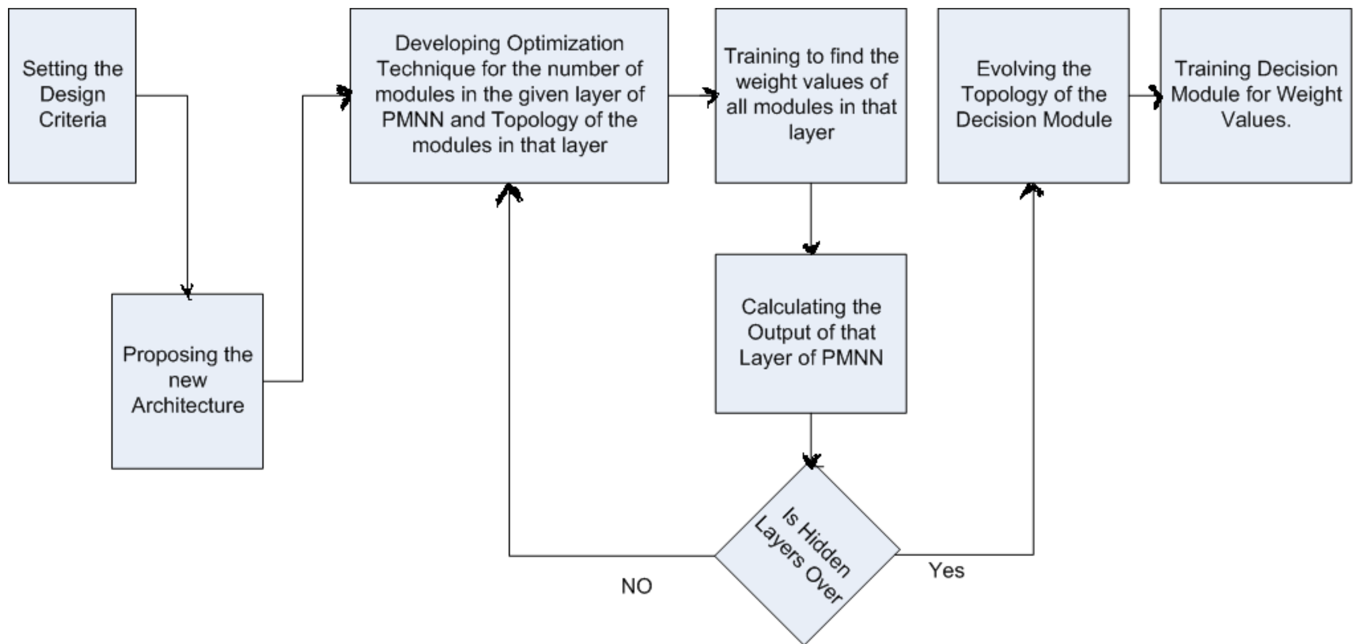

Figure 4.1: The Block Diagram of General Steps followed to Design the PMNN Architecture

4.1. The Evolution of the New Architecture

The following list shows the main design assumptions that had to be decided before setting up the Architecture. Number of architectures can be derived. But, the following questions are considered to scope down the architecture.

- Is the network homogenous or heterogeneous?
    - Are all modules of one type or are there different types of modules used?

        - Only one type (homogeneous)
- What type of architecture is used for the modules?
    - MLP? SOM? ART1? ART2?
        - MLP is used for this research work, as the applications that are going to be evaluated in this work are of classification type.
- How are the modules interconnected?
    - Are only feed-forward connections used? Are recurrent connections allowed? Are connections only made from one layer to next?
        - Only feed-ward connections as the modules are MLPs
- What is the general network structure?
    - Has the network a single layer? N-layers? Or pyramidal architecture?
        - Pyramidal Architecture
- How are the inputs connected to the network?
    - Are the inputs connected to all modules (overlapping)? Or only to one module (non-overlapping)? Are these connections made randomly or is information about the input space used? E.g. locally connected.
        - Inputs connected to only one module (non-overlapping) similar to the research work in [1] for the first and the second layers. But if there is one module with less number of inputs than other modules in the same layer of the PMNN, then to make the structure of that module uniform with other modules, some selected inputs may be subjected to double connections by connecting to two modules. The number of such inputs in each layer is based on the number of inputs required to make the additional module to have equal number of inputs with others. Additionally, although in the first and in the second layer modules of the PMNN the inputs are not connected in an overlapping manner, the outputs of the some of the last hidden layers are connected in a full overlapped manner. This forms replicated outputs of modules from the third layer of the PMNN on ward as all the layer modules are subject to the same input. As stated in chapter three, replication is one of the characteristics of modularity and it has added greater improvement over the result of the test. The first two layers of the PMNN are differed from

being connected in an overlapping manner in order to reduce the burden of increased weight connections and to reduce the dimension of the data first before being connected in such manner.

- What training algorithm is used for the modules?
  - Is a supervised or unsupervised learning method used?
    - ❖ Supervised Learning method. Each element of the network is trained using BPA with continuously changing learning factor over each epoch to avoid the algorithm from being trapped by local minima.
- How is learning organized for the whole network?
  - Does the network learn in stages? Is noise used during the training?
    - ❖ The network learns in multiple stages for each layer of the PMNN. The modules in each layer can learn parallelly as the modules are not dependant on each other.
    - ❖ Here in this thesis, noise is not applied for the test.
- How the topology (structure) of the modules in each layer be evolved?
  - Does the structure be evolved automatically by the system depending on the type of data? Or is there any assumption to make the structure of the member modules static structured? Or with mixed approach of some parameters assumed and some getting evolved automatically?
    - ❖ With Mixed approach i.e. the module structures in a layer are assumed to have of the same number of inputs and outputs. The structure of the modules for each layer of the PMNN is automatically evolved based on the number of inputs provided per each module taking only one representative module from that given layer. The weight values of the modules are searched independently for each module depending on the input subjected to each module.
    - ❖ In the implementation of this thesis, number of modules in the input layer is determined based on PSO by evaluating the performance of one of the representative module's output. The number of modules in the rest layers is decided by the user based on his/her interest usually much less than the number of modules in the preceding layer and positive integral multiple of

the number of outputs of the network. But this assumption is taken to simplify the burden of additional evolutional computing resources and time and due to the absence of more capable and resourceful computer to simulate the entire idea of the research work. But it is clearly evident that if the program can evolve the number of modules in all layers, the output and the efficiency of the network may increase apart from its huge resource requirement during evolving the network.

- How the global structure of the network being evolved?
  - Does any preset conditions assumed to make the structure static? Does it automatically evolved in the system depending on the type of input dataset? Or any mixed approach used by assuming some conditions and others to evolve from the system?
    - ❖ With Mixed approach i.e. the number of modules in the input layer is chosen randomly from some assumed range of numbers. The range of these numbers is chosen for the datasets by assuming some search space. Then using PSO, the best number of input modules is found. But the number of modules on the next subsequent layers after the input layer is set by the user statically.

- Are there any weights links from the output of the modules to the next inputs of the next layer modules?
    - ❖ No. The possible links has assumed to have the value unity. This type of methods has been used in [1]. The output of one layer of modules is another set of data and it is fed as input to the next layer modules as new reduced data. It needs not to be scaled. But in the action of determining the number of modules in a layer, if the division of the total number of inputs divided by the number of modules has remainder, some selected inputs have double connections with two modules with weight connection of half unlike others to normalize the effect of the double connection of the input trait.

- How the modules of each layer in the PMNN are evolved?
    - Is there any automatic evolving mechanism to set the best network topology or any preset assumption of the topology?
        - ❖ PSO is used to evolve the modules in each layer of the PMNN.

From the list above it can be seen that the number of possible network structures is huge. In Figure 4.1 (a-c) example architectures are given.

The proposed PMNN network Architecture is shown in figure 4.2 below. The detail discussion of this network topology is given on the following section 4.2.

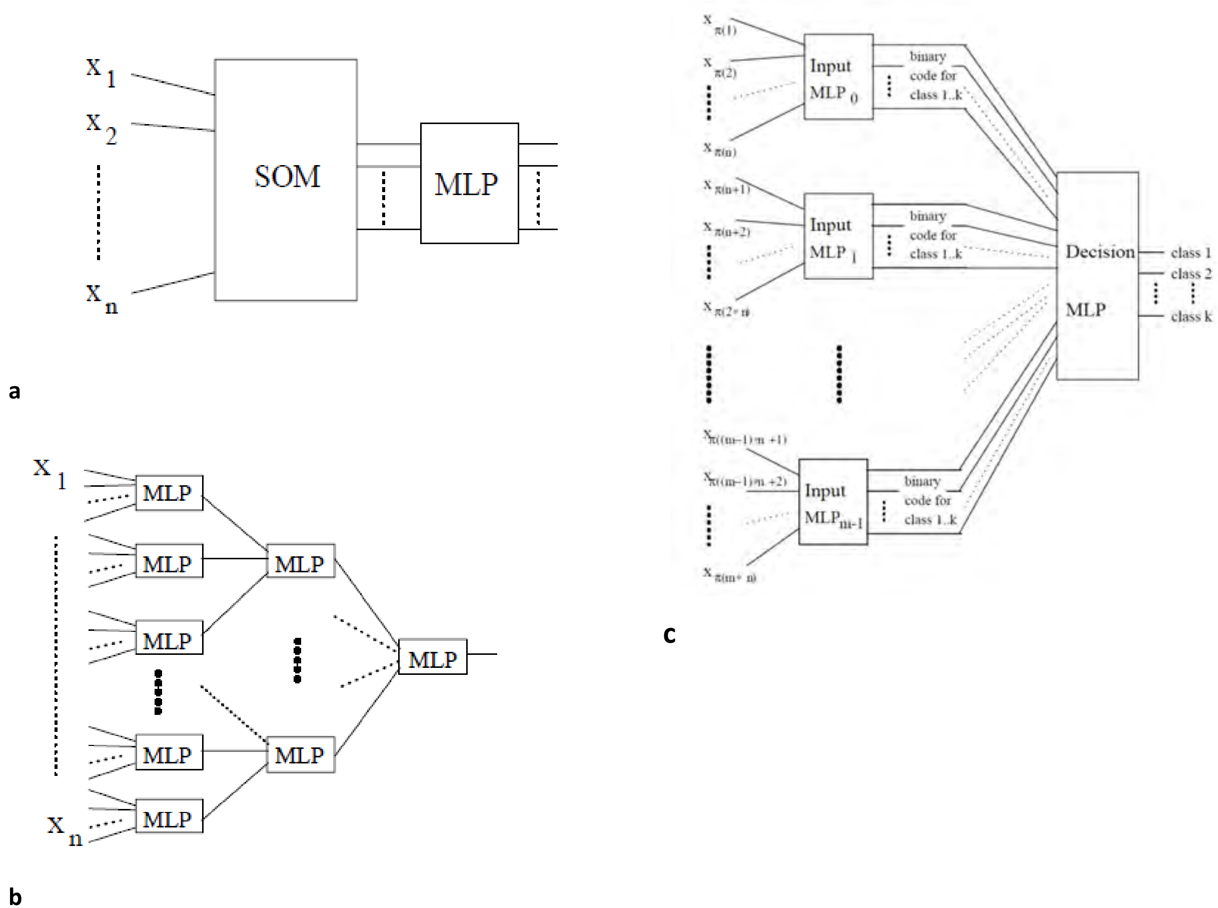

Figure 4.2: Examples of Modular Neural Network Architecture

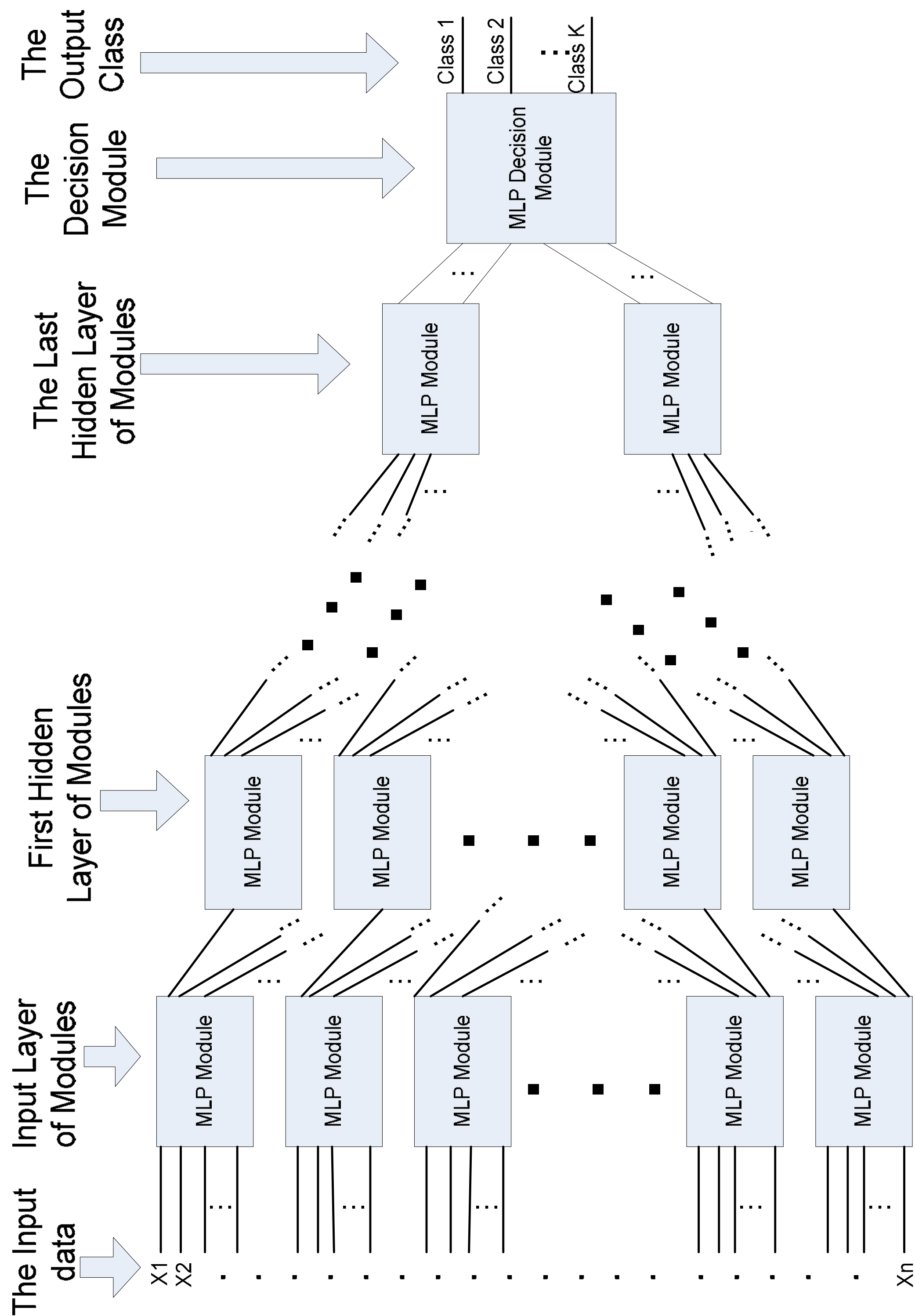

Figure 4.3: The proposed Pyramidal Architecture

4.2. The Proposed Architecture

The proposed network architecture as shown in figure 4.2, consists of a layer of input modules, layers of hidden layer modules and an additional decision module. All modules are MLPs. Only the number of inputs and the number of outputs of the module and its structure are determined by the system using PSO depending on the applied input dataset. The internal structure, such as the number of hidden layers and the number of neurons in each hidden layer can be chosen independent of the overall architecture.

Each input variable is connected to only one of the input modules in the first layer of PMNN. In a similar manner, the outputs of the input layer of the PMNN serves as the input to the first hidden layer of the PMNN. These inputs also are connected to only one of the first hidden layer of PMNN modules. These connections are chosen sequentially. But after the input data dimension is reduced enough, the outputs of some of the last hidden layers are connected fully in an overlapping manner making replications of modules in those layers. Then, the outputs of the last hidden layer modules are connected to the decision module.

4.2.1. Design of Input Layer of PMNN

The following additional assumptions are taken to simplify the implementation of the proposed network for the input layer of PMNN:

1. In the discussions that follow, the dimension of the input vector is denoted by l and the number of classes by k i.e. the output from the network is using binary representation of the output and hence the number of outputs of the network (c) is $\log_2 k$ if k is the power of 2 or the integral part of $\log_2 k$ plus one as shown in equation 4.1. Moreover, the modules constituting the input layer and the hidden layers are known as ***component module(s)*** whereas the module that makes the final output decision is referred to as ***decision module*** in this work.

$$c = \begin{cases} log_2 k, if\ k\ is\ the\ power\ of\ 2 \\ int(log_2 k) + 1\ , if\ k\ is\ not\ the\ power\ of\ 2 \end{cases} \quad \text{...4.1}$$

2. To determine a modular network it is necessary to specify either the number of inputs per input module or the number of input modules. These parameters are dependent on each other. It is assumed here that the number of inputs per module in the first layer is chosen as n, the

number of input modules (m) in the input layer can therefore be calculated as $m = \left[\frac{l}{n}\right]$. The remaining inputs left unconnected denoted as r is calculated as $r = [l\%m]$. It is further assumed that $l = m * n$. If this is not the case the remaining unconnected inputs can be connected to the next additional module so that the number of modules is increased to m+1. Alternatively, it is possible to alter the size of one module or of a group of modules. It is also possible to make overlapping connections to make one additional module to take the remaining inputs and have some additional inputs to be structurally the same with other component modules by making double connections for some selected inputs with two component modules. In the implementation of the architecture, one additional module with the unconnected inputs and additional inputs from double input connections is used.

3. The number of input modules, m, is represented by a single dimensional integer which is taken as one dimensional particle to be used in PSO to optimize the overall topology of the network. To simplify the implementation in this research work, it is assumed that the search space of the network to be between 50 and 200. But one can widen the search space beyond this for the best optimal solution. The lower bound is chosen with an appropriate test result. But the upper bound is arbitrarily chosen for the sake of reducing the search and training time.
4. Each module have number of outputs which is equal to $\log_2 k$. So $l_{h+1} = m_h * \log_2 k$. This makes the number of inputs of the next layer being known at the very beginning.
5. The division of inputs into different modules is random. Each module in a given layer have equal number of inputs. If statistically neutral inputs are fed to a module, the module cannot learn [1]. But as stated in [1], the phenomena of statistically neutral inputs to a module can be ignored for high dimensional inputs since this occurrence is not seen in its simulation.
6. The number of inputs of a module is assumed to be always greater than the number of outputs of that module. So for component modules, number of inputs of the module, n, is always greater than $\log_2 k$. In the same way, for the decision module at least outputs from two component modules are used as an input to the decision module. This criterion marks the end of hidden layers of the network. In other words, if the number of outputs of a given layer of component modules is not greater than or equal to twice of the number of inputs of one component module, then this dataset is directly considered to be the input of the decision

module. Else there will be another hidden layer of component modules and we will proceed to find the next layer structure.

4.2.2. Design of Hidden Layers of PMNN

1. Assumptions and procedures mentioned in section 4.2.1 to determine the number of modules in the given layer and the number of inputs per each module can also be used for other hidden layers of PMNN. But here in this research work for simplification of the implementation, the number of modules in the next subsequent layers is assigned by the user from the factors of the number of outputs of the immediate previous layer.

4.2.2. Design of Decision Module of PMNN

1. The outputs of all the modules in the last hidden layer of PMNN are calculated. Then all the outputs of those modules are fed to the decision module. Using these outputs, the decision module topology is evolved using PSO. After the best topology is found, then it is trained to set the weight values of the connections within the module.

4.2.3. Design Assumptions for both module types

1. The topology of both the component modules and the decision module are also evolved using PSO to determine the hidden layer structures of the modules. Once the topology is known, the training takes place to find the weight values and then fitness value is measured.
2. Both component modules and the decision module are trained using supervised learning using BPA with randomly selected learning factor in each epoch to avoid the trapping of the training in local minima.

4.3. Topology Selection

A practical issue that arises when applying NN models (topologies) to solve any problem is the model (topology) selection. Specially, when one is intending to solve problems with self-scaling topology of neural nets, model selection becomes a more complex task. The self-scaling neural networks scale their topology in an incremental fashion so as to optimally match the complexity of the task. Model selection is a methodology for choosing the adequate size of NN architecture to learn the task, yet not compromising the NN performance. The self-scaling NN architectures

are categorized into two broader types, namely hierarchical and vertical, depending upon the way the growth of the NN models proceed [2].

The problem of model selection for the ANNs has been tackled in a variety of ways. The simplest form of model selection for the neural networks is to train a number of NN models with different configurations and topologies simultaneously. Then by evaluating the performance of each of the individually trained neural networks on an independent data set, the best performing NN that generalizes well is selected and is assumed to be the optimal neural network, as far as the desired configuration and topology is concerned for a given learning task [2].

The model selection problem can be handled in a better approach when the process of NN parameter estimation is carried out simultaneously along with a NN topology altering methodology. Recently, many researchers have investigated different approaches that alter the network architecture or topology as the learning of NN progresses [2].

As discussed in [2], one such approach involves using a larger than needed NN and training it. The oversized architecture is evaluated to detect obsolete and ineffective weight connections or neurons in the NN model which then are removed before further evolution of architecture is continued. This process is repeated iteratively until a predetermined stopping criterion is achieved. The algorithms following this approach are called *pruning* algorithms [25].

Another approach uses an opposite strategy to reach an optimal topology for a NN model. This approach attempts to search for an appropriate NN topology by starting with a small NN model and successively adding hidden nodes or layers until reaching a satisfactory topology that can optimally perform the given learning task. The basic idea of all of the growing or constructive methods is to use some criterion to dynamically extend the existing NN model to improve its overall performance. The training algorithms falling into this category are called *constructive* algorithms [26].

Constructive approaches to the NN design and learning have some advantages over the NN pruning approach. Firstly, it is very easy to specify the initial NN configuration and topology, whereas while using a pruning algorithm based learning approach, it is very hard to guess how large initial NN is needed to solve a given problem. Secondly, the constructive learning starts with a small NN model and is thus more computationally plausible than pruning based learning in which majority of the training time is spent training an overly large neural network. Thirdly, constructive algorithms for NN design generally tend to find smaller neural networks than that of

the pruning algorithms [26]. Based on the aforementioned observations, a constructive approach is followed in [2] in order to design and implement the proposed self-scaling NN models.

There are also some factors that need to be taken into consideration when following the constructive approach to the NN design and training. For example, one of the most important factors is to determine when to stop further addition of the hidden nodes and layers or how many hidden nodes or layers to add after the performance of the network does not improve significantly. In addition, a decision has to be made about the selection of a training mechanism as well. For example, whether to train the whole NN after the addition of hidden nodes and layers or only to train the newly added hidden nodes and layers has to be decided [2].

4.3.1 Topology Selection for Overall Architecture

This topology (model) selection concept is discussed here to select the correct topology of the PMNN.

Here in this thesis, both pruning and constructive algorithms are mixed to be used and it is named as the new mixture of this algorithm as *PSO based pruning–constructive algorithm* as shown in figure 4.3. This PSO based algorithm codes both global and modular network topology of the Pyramidal Neural Net with general PSO codes separately and searches moving freely up (constructive algorithm) and down (pruning algorithm) of the search space provided by using particle population. Two nested algorithms are used; the development of modular topology being nested inside the global algorithm for development of overall topology of the PMNN network.

For global topology development algorithm, the number of input modules is coded with single number and initializing some random particle population and starting the search. The maximum number of the input modules is assumed to determine the position and velocity dimensions of the particle. Once the number of the input module is known, the subsequent number of modules of all layers will be known (assumed here in this implementation). Moreover, the number of inputs to each component module of layer is the same and hence all the modules in the same layer of the PMNN have the same topology, but weight connections are developed by training the network.

4.3.2. Topology Selection for Modules

As discussed above, the number of nodes in the input and output layers are determined by the problem and how it is modeled. The number of hidden layer units, m, can be specified randomly (e.g. initially set to the same as the number of input nodes, n, or some combination of n and the number of output nodes, p). This number, m, can then be systematically altered until results of the best 'quality' have been obtained. Quality is usually determined by some performance measure or multiple measures applied initially to the training data, and then examined for other datasets (see further, below). Quality is also measured in terms of model simplicity (number of parameters implied in the model structure). Some software packages attempt to compute suitable structures automatically. For example, a standard rule may be applied for selecting the number of hidden nodes, such as:

$$m = \text{int}\left(\sqrt{np}\right) \quad \text{.......... 4.2}$$

Where int() is the integer part function. Hence with 3 inputs and 2 outputs, this rule would give m=2. Other rules of thumb suggested by various authors include $m= (n+p)/2$, $m<2n$ and $m=2(n+p)/3$ [27].

In one program written in [28], it sets the maximum number of inputs as 100, the maximum hidden nodes as 100, the maximum output number as 40 and the number of hidden layers as one.

Here in this research work, the maximum number of hidden layers of a module is assumed to be 1 and the maximum number of neurons in the hidden layer having 20 minimum and 100 maximum. The minimum 20 is selected based on the result of sample module training and the maximum is selected for the purpose of simplification. So using PSO, the optimal topology of

the modules is searched between these set maximum and minimum values.

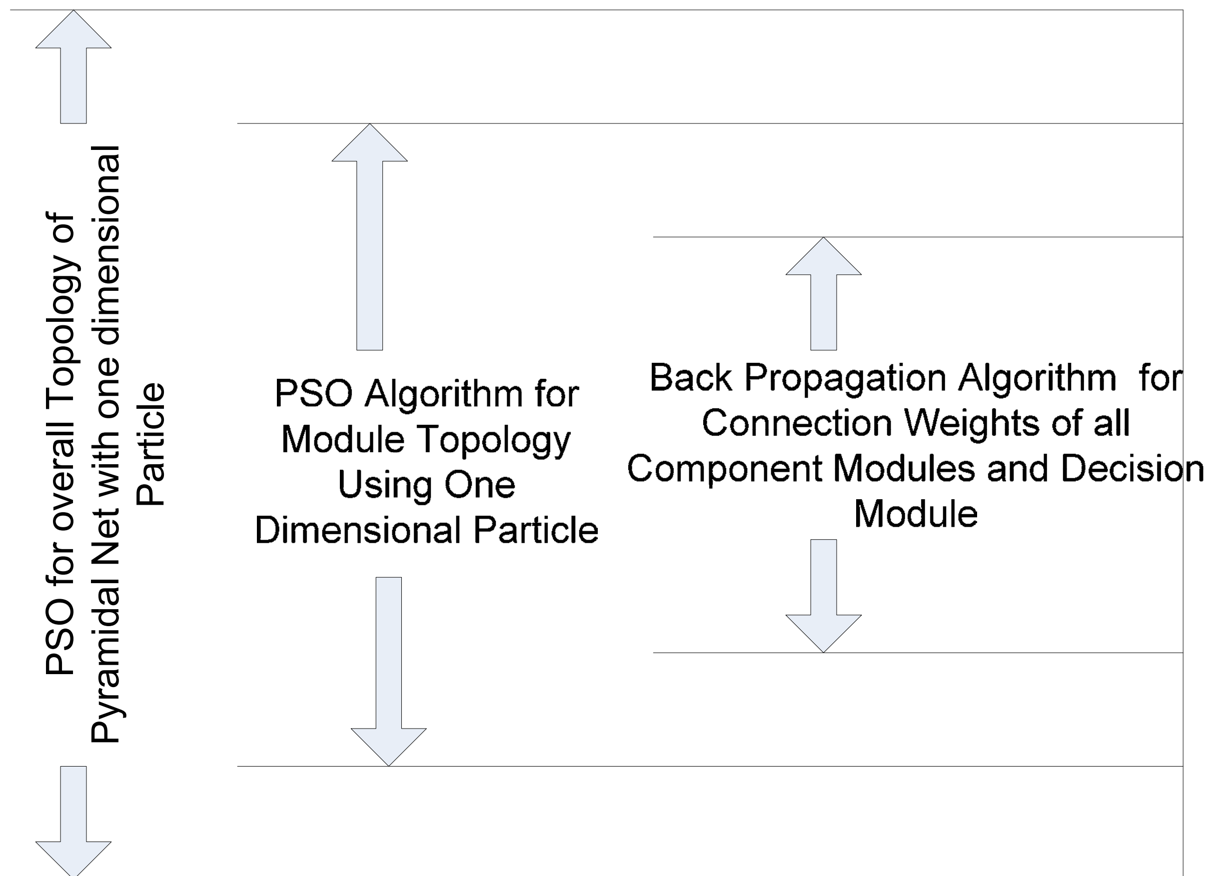

Figure 4.4: PSO based Self-Scaling Pruning-Constructive Algorithm for Pyramidal MNN Development

4.4. Training the System

The training is divided into two parts. The first part is to evolve the overall network topology of the PMNN and the module topologies as discussed in 4.3. The second part is to evolve the weight-link values of each module. The training occurs in multiple stages, using the PSO for topology search and BPA for the weight search.

Each input is prepared as float array and permutated before being fed to the network. The individual training set for each module is selected from the original training set and consists of the components of the original vectors which are connected to this particular module as an input vector together with the desired output class represented in binary or 1-out of k-coding.

Too small a network cannot learn a problem well and on the other hand an oversized NN will lead to an over fitting of the training data set and subsequent poor generalization performance [2]. Hence the use of PSO to find the optimal topology in this research work reduces this problem although it is not implemented for all modules due to the constraint of computational resources. All the modules in the same layer have the same number of inputs and hence one sample module is considered for the topology search that is taken to be for all the modules in the same layer. But it is obvious that the topology of an MLP depends not only on the number of inputs and outputs, but also on the nature of the input dataset. So if enough resources are available for computing the network (for evolving and training), the topology of each module composing the network should be evolved based on its own data provided to that module.

In the first phase, the number of input modules and the best topology of the modules in the first layer are evolved using PSO. Here only one sample module is taken for the topology search to reduce computational time. Then all modules in the input layer are trained using the same module topology. To calculate the training set each original input pattern is applied to the input layer; the resulting vector together with the desired output class (represented in a 1-out of k coding) form the training pair for the next layer of PMNN. In the second stage the number of modules of the second layer of the PMNN and the topology of one sample module are evolved using PSO. The modules in first hidden layer network is then trained using BPA. Similarly, the subsequent layer topologies are evolved and the modules of those layers of the PMNN are trained having the training set from the predecessor layer of PMNN. Finally, the decision module topology is evolved and the module is trained with the training set from the output of the last hidden layer of PMNN.

To simplify the description of the training a small intermediate representation is used, it is also assumed that the permutation function is the identity function, $\Pi(x) = x$ for all layers inputs except the first original input to the first layer. The input to the first layer of the PMNN is permutated using a sequence generated by sequence generator function:

$$x(n) = (5\,x(n-1) + 1)\%\,N$$

where n is the index of the sequence number, x(0) is the starting seed of the generator whose integral value to be assigned by the user between 0 and N, x is an array variable holding the

sequence numbers, and N is the number of elements of the sequence which is equal to the length of array x.

The original training data set after being permutated is: TDS = $\{X^j_1, X^j_2, X^j_3 \ldots X^j_L: C^j\}$, where $x^j_{i\varepsilon R}$ is the ith component of the jth input vector, C^j is the class number, j=1, 2… t, where t is the number of training instances.

Therefore, the module MLP_i is connected to: $X_{i.n+1}, X_{i.n+2} \ldots X_{(i+1).n}$.

The training set TDS_i for the module MLP_i: $\{X^j_{i.n+1}, X^j_{i.n+2} \ldots X^j_{(i+1).n}: C^j_{BIN}\}$ for all, j=1, 2… t, where C^j_{BIN} is the output C^j represented by binary code.

Therefore, the mapping performed by the input layer of the PMNN is:

$\Phi: R^{nxm} \rightarrow R^{mx[\log_2 k]}$

The mapping performed by the first hidden layer of the PMNN is:

$\Phi: R^{n(1)xm(1)} \rightarrow R^{m(1)x[\log_2 k]}$, where m(1) is the number of modules in the first hidden layer and n(1) is the number of inputs per module in the first hidden layer.

The subsequent mapping is done in a similar fashion.

4.4.1. The Training Algorithm:

The training Algorithm of this proposed network is as follows:

- Stage 1- Training Input Layer of the PMNN:
  1. Search the best number of input modules using PSO.
  2. Evolve the best topology of one sample module based on the search result in 1.
  3. Assign the topology of all modules with the topology found in 2.
  4. Select the training sets TS_i from the original training sets TS, for all i = 0, 1, … , m-1
  5. Train all MLP_i for all TS_i using BPA.
- Stage 2- Training the first Hidden Layer:
  1. Calculate the response TS_1 of the first layer for each input vector j

     $TS_1 = \emptyset(x^j_1, x^j_2, x^j_3, \ldots, x^j_i)$
  2. Search the best number of input modules using PSO.

3. Evolve the best topology of one sample module based on the search result in 2.
4. Assign the topology of all modules with the topology found in 3.
5. Select the training sets TS_{1i} from the output of first layer modules training sets TS_1, for all i = 0, 1, … , m_1-1
6. Train all MLP_{1i} for all TS_{1i} using BPA.

❖ Stage 3- Training the second Hidden Layer:

1. Calculate the response TS_2 of the first layer for each input vector j

   $TS_2 = \emptyset(x_1^j, x_2^j, x_3^j, \dots, x_{l1}^j)$
2. Search the best number of input modules using PSO.
2. Evolve the best topology of one sample module based on the search result in 2.
3. Assign the topology of all modules with the topology found in 3.
4. Select the training sets TS_{2i} from the output of first layer modules training sets TS_2, for all i = 0, 1, … , m_2-1
5. Train all MLP_{2i} for all TS_{2i} using BPA.

-
-
-

❖ Stage N- Training decision Layer:

1. Calculate the response TS_N of the first layer for each input vector j

$$TS_N = \emptyset(x_1^j, x_2^j, x_3^j, \dots, x_{lN}^j)$$

2. Evolve the best topology of decision module.
3. Assign the topology of the decision module with the topology found in 2.
4. Set the training sets TS_N from the output of first layer modules training sets TS_N.
5. Train the module using BPA on the dataset TS_N.

4.4. Calculation of Output

The calculation of output also occurs in multiple stages depending on the number of layers. First the input sub-vectors for each module are selected from the applied input vector according to the permutation function and the intermediate output is calculated by all modules. In a second stage all the outputs from the input layer are used as input for the second layer of the PMNN. This process continues until the desired output is given out by the decision module.

The mapping of the whole network is denoted by:

$$\emptyset \circ \emptyset_1 \circ \emptyset_2 \circ \ldots \circ \emptyset_N : \boldsymbol{R}^l \rightarrow \boldsymbol{R}^k$$

The k-dimensional output of the decision module is used to determine the class number for the given input.

4.5. Analysis of the New Architecture

In this section, some aspects of the architecture are considered. This analysis presents some back-ground to the network behavior on aspects of speed, learning and generalization.

4.5.1. Training Time and Speed

When we consider the learning time, since the inputs are divided into different modules in each layer with distinct and mutually independent training dataset, all the modules in a given layer can be trained fully parallelly as far as the training algorithm is implemented in such a manner. So the total training time of the network is the training time of one of the modules in the first layer plus the training time of one of the modules in the second module plus the training time of the third module plus … plus the training time of the decision module. Practically when we consider the dimension reduction in each layer, the resulting network usually is not having more than four layers including the decision module. So the average training time is four times the training time of one module as long as full parallel training is implemented in each layer.

If we consider the monolithic MLP, the weight connections of the high dimensional inputs to a single MLP and the structure of the MLP to handle this huge input dimension is larger and even additional hidden layer may be required to handle that complex input data. Here the weight connection in this monolithic MLP is therefore very huge when compared to the modules that constitute Pyramidal Topology considered in this research work. Due to high number of weight

connections, the training time to achieve the same specified stopping criterion is very much greater than a single module that is part of this architecture. The time required here is even very much greater than four times the time required to train one module that constitute the network considered here in this thesis. That is it is more than the average time required to train the network by implementing the training in each layer parallelly.

4.5.2. Learning Problems

Splitting-up the training set into subsets can also bring problems. The number of equal input vectors with different possible output values may increase, especially for modules with a small number of input variables. [1]

To discuss this problem more general the following definition is necessary. P(y=a) is the probability of the output y having the value a, and P(y=a|x=b) is the conditional probability of y=a if x=b.

Definition: A Set of Statistically Neutral Variables [1]

Consider the function $f(x_{1,} x_2, \ldots, x_n, \ldots, x_m) = y$. A sub-set of the input $\{ x_{1,} x_2, \ldots, x_n\}$ with $n<m$ is called *statistically neutral* if the knowledge of the values of these variables does not increase the knowledge of the result y.

Formally: The set of input variables $\{x1_, x_2, \ldots, x_n\}$is *statistically neutral* if

$P(y=a) = P(y=a|x_1=b_1 \& x_2=b_2 \& \ldots \& x_n=b_n)$

If the whole set of input variables and all possible subsets for all modules are statistically neutral, the network cannot learn the task. If only some of the modules are supplied with a statistically neutral set of input variables the network may perform satisfactory.

In [32], it is shown that monolithic multilayer feed-forward networks can learn statistically neutral problems but they cannot generalize on them. This situation sounds very unlikely in real world data, and the expectation was observed during experimentation. Especially in tasks with a large input space, such as picture recognition, this problem has not happened.

In order to avoid one module with statistically neutral training sets while some more important traits for learning concentrated in some few modules, the permutation of the input data set before being given to the network is applied. This method has greatly reduced this problem and even the individual modules could learn much better than the single monolithic MLP taking all input traits.

4.5.2. Generalization Ability

The ability to generalize is the main property of neural networks. This is how neural networks can handle inputs which have not been learned but which are *similar* to inputs seen during the training phase. [1]

The proposed architecture combines two methods of generalization. One method is built-in to the MLP. Each of the networks has the ability to generalize on its input space. This type of generalization is common to connectionist systems. The second method of generalization is due to the architecture of the proposed network. It is a way of generalizing according to the similarity of input patterns. This method is found in logical neural networks. [1]

Chapter 5

5. Implementation of the Proposed Architecture

The main motivation for developing the software was to evaluate practically the proposed architecture. For larger neural networks it is very difficult to analyze the behavior of the system theoretically, in particular the generalization performance. One way to demonstrate that the architecture is working is to run a simulation. Nevertheless this methodology is widely used in the field of neural networks [1]. From a more practical point of view a successful simulation does "prove" that the architecture is capable of solving a certain type of problem.

In this section aspects of the implementation are discussed, including the structure of the software for the MNN, the usage of the programs. The implementation details for preprocessing of the image data are also included.

5.1. Considerations

The following issues have to be considered for the implementation of the proposed topology:

- The implementation is seen as a prototype for the suggested architecture. It is inevitable that alterations would need to be made during the project and this required the development of a flexible implementation.
- One area of interest is to investigate the generalization performance on large input spaces.
- In order to compare the monolithic network with the proposed PMNN network, it is required to have the same implementation with component modules. The topology of the monolithic network and the training of the network for the weight values have to follow the same procedures.

The ease of its programming, flexibility, and its resourcefulness to implement different types of data structures as well as the experience of the researcher led to the decision to use C# programming language.

5.2. The Software Platform

The use of C# in Visual Studio.NET 2010 requires .NET Framework 4.0 to be used in other systems. As long as the framework is available, it can be operational on any machine and systems. The implementation is developed on Windows 7 operating system with core i3 processor laptop.

5.3. Data Preprocessing

The preprocessing of the raw input data may appear to be a trivial task; but can have a major impact on the performance [32]. A separate input processing program was developed to convert the data into float array that is fed to the network. After the data is converted to float array, the array is permutated using the permutation function. Then the values of the array are from 0 to 255 to represent the given color value of each pixel. To normalize the data, each element of the array is divided by 255 so that all elements of the array have values between 0 and 1. This subsystem serves both network topology evolving task and also weight training task of the program. Additionally, some programs that serve both tasks similar to data preprocessing are also part of this subsystem. For example, the program to divide input arrays into different modules, modular output calculations, and other similar tasks are used by two tasks and part of this subsystem.

5.4. The Program Structure

To develop the MNN first a library is written. All the important functions and objects are included in this library. The following components of the program are developed.

5.4.1. For Evolving the Topology of the Network

This subsection has two purposes: to search the optimal number of the modules in each layer of the PMNN and to evolve the topology of modules in each layer. Only one sample module topology is evolved in each layer and applied for all modules of that given layer. For both purposes PSO Algorithm is used. The block diagram of structure of the program is shown on figure 5.1. The PSO Algorithm is discussed in section 3.3. The following two subsystems are developed to address these two tasks.

5.4.1.1. For Number of Modules in the Input Layer of the PMNN

This part of the program uses PSO Algorithm to evolve the number of modules in the input layer of the PMNN. The classes in this subsystem are pyramidal topology particle, pyramidal topology particle swarm, pyramidal topology particle function minimizing, and pyramidal topology particle function minimizing swarm. During the topology search, this subsystem uses the subsystem discussed below in section 5.4.1.2 to evaluate the cost of each pyramidal particle during evolution. The program developed for this section is given in Appendix B.

5.4.1.2. For Evolving the Topology of the Modules in Each Layer

This part of the program uses PSO Algorithm to evolve the topology of the modules taking one sample module from each layer. The classes in this subsystem are modular topology particle, modular topology particle swarm, modular topology particle function minimizing, modular topology particle function minimizing swarm. During the topology search, this subsystem uses the updating of the weight values of the modules as discussed below in section 5.4.2 to evaluate the cost of each modular topology particle. Under the given number of modules in the input layer, the topology of the sample module is evolved using PSO and each evolved topology is trained using BPA. The root mean square errors of the test evaluated after one epoch training against the test set of data is used as the cost of the corresponding particle of the topology of the sample module.

In the first stage of the training, both optimal number of input modules and the best modular topology of one sample input module are evolved using this PSO algorithm. For the rest of the layers, only modular topology of one sample module of the given layer is evolved. But the number of the modules in those layers is assumed to be factors of the number of outputs of the immediate preceding layer.

Although permutation is applied on the data before being fed to the network, it clearly known that the nature of the data given to each module in each of the first two layers is slightly different. For this reason it would have been good if the topology of each module in each layer are evolved separately independent of the other. But the computational time and other resources to do so make constraints on the implementation in such manner. Assuming that each module in a given layer has the same number of inputs and outputs, the topology of the module is assumed

to be a function of these two parameters only in this implementation. The PSO program is given in Appendix A for each class.

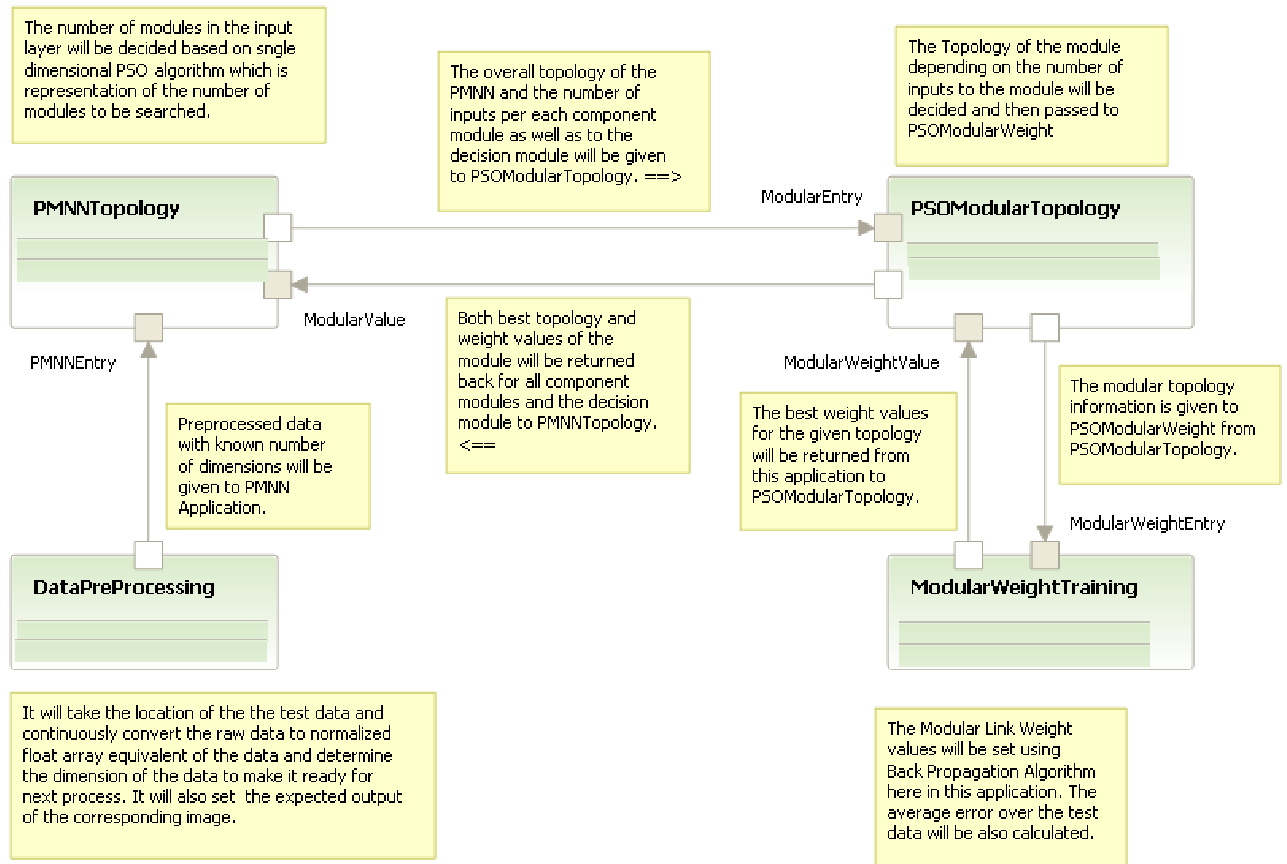

Figure 5.1: The PSO Algorithm Structure used to search the overall topology and the topology of the modules

5.4.2. Modular Weight Training Using Back Propagation Algorithm

The modules are trained using BPA. The implementation of this BPA is developed using some classes that commonly serve all the modules as libraries and some specific functions to address evaluations and other important factors. The block diagram of the structure of the program is shown below on figure 5.2.

As it is mentioned above although permutation is applied on the data before being fed to the network, it clearly known that the nature of the data given to each module in each of the first two layers is slightly different. Due to this fact, there is a slight difference on the nature of convergence of the modules even among the same layer. So to avoid this problem, a loose

stopping criterion is set with the same value in a layer but different values among the layers as error is getting reduced step by step from layer to layer.

Additionally, in order to avoid the trapping of the training in local minima, a randomly generated learning factor is used in each epoch. No matter what causes local minima, generating random learning factors and scanning through neighborhood are normally used to escape local minima. However, random learning factor is not efficient, scanning is time consuming, and both do not guarantee to escape local minima within a limited time frame [33]. The paper in [33] presents an innovative search method, "retreat and turn", to help Back-propagation escape from most of local minima. But "retreat and turn" appears complex with huge number of modules to be trained. Additionally, the topology having pyramidal structure, it reduces the network errors hierarchically from layer to the next layer. So for simplification, randomly generated learning factor is used here. It is also experimented to be better than to use a constant learning factor.

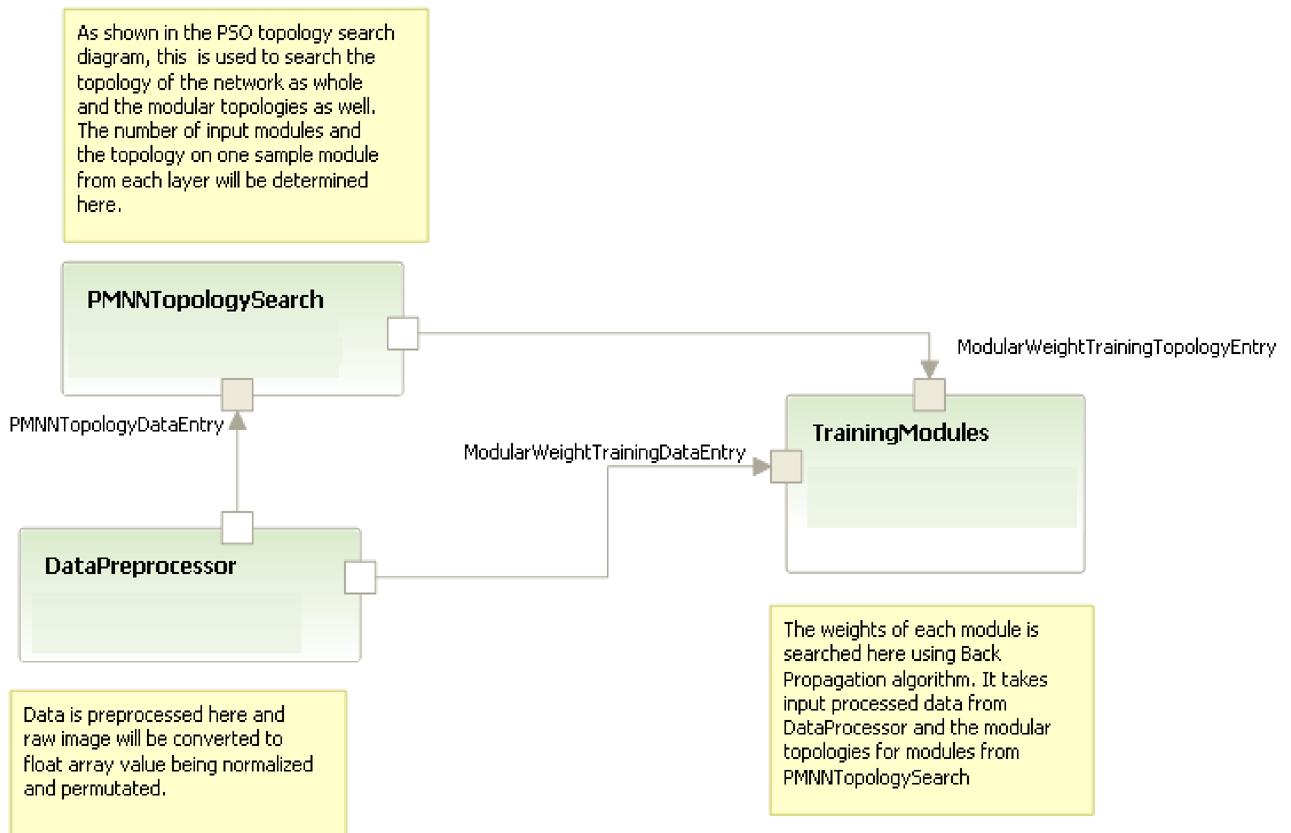

Figure 5.2: The Back Propagation Training Algorithm structure of the proposed architecture

Chapter 6

6. Experimental Evaluation

6.1. Introduction

To evaluate the proposed PMNN, a number of tests were carried out to examine its performance of classification and its ability of generalization on any given classification problem. A variety of different data sets were used to find a framework for the possible application domain for the proposed architecture.

It is applied for 3 types of pattern recognition problems which have high dimensional inputs: for *Palm-print Recognition*, for *Iris Recognition*, for *Face Recognition*. The dimensions of these three datasets vary from about 20,000 to 60,000.

A minimum criterion for experimental evaluation of NN learning algorithms is: *"An Algorithm evaluation is called acceptable if it uses a minimum of two real or realistic problems and compares the results to those of at least one alternative algorithm" [34].* The experimental result for this research work is well beyond this principle.

The result of the proposed PMNN network is compared with two types of other well-known and well-researched existing networks and the proposed network has performed better. These two networks are the monolithic MLP and the network architecture previously done in [1] having input layer of modules and one decision modules. The comparison was made with only one dataset, i.e. the *palm-print recognition* problem. Since the performance of PMNN is found to be better than other architectural results, it is decided that doing the same comparison with other high dimensional datasets is a trivial work for we can deduce that the newly proposed network performs well in comparison to the above two network types. As far as high dimensional input vectors are concerned here in this thesis, the comparison is considered for only such types of datasets.

A simulation is always based on a finite number of test cases. It is therefore difficult to draw a general conclusion from the tests. This problem particularly appears in comparing different networks. A reference network with a different learning parameter or number of hidden layers might have given a better result; or a different number of inputs to each module in the modular

network might have increased the performance [1]. To solve this problem that would result in a different comparison result, the modular and overall topological optimization technique using PSO Algorithm is applied for both two works that are used to compare the result of this proposed network. Therefore, modular topology optimization is applied to search the best topology of the monolithic MLP and both searching optimal number of input layer modules and modular topology search for both the input layer modules and the decision modules is applied before the comparison is made.

6.2. Test Methodology

The following methodology was used for the tests:

Two data sets were prepared: a training set and a test set. Some data sets were split randomly in two sets of equal size (plus/minus one if the number of instances was odd). For other tests selected instances were used for training and the entire test and training sets for measuring the performance. The networks first learnt the training set and then the recognition or prediction performance was calculated on both sets.

The result of the training set is denoted by P_M and it is a measure for the memorization capabilities. The performance on the test set is given as P_G and indicates the generalization abilities of the trained network. [1]

To compare the training time, due to the unavailability of capable hardware and computer system, it was not possible give the comparison results of the training time. Only the theoretical claim with respect to the time betterment of the network is given in chapter 4.

6.3. Pattern Recognition Problems

In this experiment both grayscale and color picture recognitions are considered. Two grayscale pattern recognitions: palm-print and iris recognition are used. The two color system with 16 bit color scale for face-recognition system. In both the three pattern recognition problems a promising result has been achieved.

The decision module output neurons give to 0 or 1 bits as an output. The used sigmoid function gives rise to the continuous values between 0 and 1. These final outputs are rounded to 0 and 1 based on the following equation 6.1.

$$output\ bit = \begin{cases} 1, if\ output\ value \geq 0.5 \\ 0, if\ output\ value < 0.5 \end{cases} \quad \text{......}6.1.$$

The output bits of the decision module represent the binary representation of number of the class being represented. All the modules that compose the PMNN Network have the same expected outputs and number of output neurons for a given application.

As modularity is based on task decomposition and replication [1], some of the hidden layers (third to fifth) in all the three tested applications are consist of replicated modules.

The performance is given in percentage of correct classification of each image in the given application as shown in equation 6.2:

$$\text{Percentage of Correct Classification} = \frac{NumberofCorrectClassification}{TotalNumberof\ \mathrm{Im}\, agesInTheTest} * 100 \quad \text{......}6.2.$$

6.3.1. PSO Algorithm in Topology of the PMNN and the modules

The PSO algorithm discussed in section 3.3 is used to evolve the number of modules of input layer of the PMNN and the optimal topology of sample modules in each layer of the PMNN. The PSO algorithm used to evolve the topology of the sample module of the input layer of the PMNN is nested inside the PSO algorithm for determining the optimal number of modules in the input layer.

6.3.1.1. For Topology of the PMNN

The topology of the PMNN refers to the number of modules in the input layer of the PMNN. The number of modules is manually set for other layers of the PMNN. The number of particles initialized for pyramidal topology is 20 since initial number of swarm is usually between 20 and 50. The particle positions are initialized with values between 50 and 200 in the search space. So the swarm (population of particles) is having 20 particles. Each particle has position and velocity. Each particle's position corresponds to the number of input modules of PMNN which are set randomly. Then the position of each particle is evaluated for the cost and then the particle that has the minimum cost from the swarm is designated as global best (global best position). The current position and cost of the particle and the position of the particle with minimum cost

from its entire history is stored and is designated as local best position. The next swarm is then updated depending on the equations 3.7 and 3.8 for its velocity and position respectively. On the next iteration of evolution after updating, each particle is evaluated again for its corresponding cost of its position. The global best of the swarm and the local best of each particle positions are updated after the evaluations.

The cost of each particle is evaluated after the optimal topology of the module is found using the procedure stated in section 6.3.1.2. The cost value of the best topology is taken as the cost of the pyramidal particle.

The stopping criterion is when the swarm converges. All the particles are moving to the best position from the search space provided and finally each particle attains stable positions with the same cost value and no cost improvement is observed for each particle. Then the iteration stops here and then takes the best position of the particle with minimum cost as the return value of the evolution. The results of the evolution of the number of modules in the input layer of the PMNN for the tested pattern recognition problems discussed below is given under each corresponding section.

6.3.1.2. For topology of Sample Modules in Each Layer of the PMNN

After the numbers of modules in each layer of the PMNN are set, the optimal topology of one sample module is evolved using PSO algorithm. Then the same topology is used for all other modules in the same layer. Modular topology particle swarm has 20 particles initialized. The particle positions are initialized between 20 and 100 in the search space. The particles are characterized by two values: the particle position and particle velocity. Each particle's position corresponds to the number of hidden layer nodes of the module which are set randomly. These particles are evaluated for the cost of the positions. Then the global best of the swarm and the local best of each particle are set.

In the iterations of the PSO algorithm, the positions and the velocities are updated based on equations 3.7 and 3.8 respectively. Then the next swarm particles are evaluated for corresponding costs. Then global best of the swarm and local bests of particles are updated based on the swarm evaluation results.

The cost of each particle is evaluated after the training of the module for weight values using BPA. The root mean square errors of the module over the test data is returned as the cost of the corresponding particle.

The stopping criterion discussed in section 6.3.1.1 is also used in this section. The values of the evolution for each layer under the following pattern recognition problems are given in the corresponding sections of the application result discussions below.

6.3.2. Palm Print Recognition

In this Palm Print Recognition problem, the data input contains 80 palm-print images captured from 10 subjects from the right hand. All palm-print images are 8 bit gray-level JPEG files by Institute of Automation, Chinese Academy of Sciences using their own self-developed palm-print recognition device. About 5,502 images from 312 subjects with both left and right hand images are prepared by this institute. From this image database, 80 left hand images for ten subjects are selected for the purpose. Here in this thesis, 40 even numbered images are used for training and all 80 images for evaluation (testing). All the images are 640x480 pixels and gray scale as stated above. The objective of this application is to train the palm print of 10 different persons and to correctly recognize the person when subjected with test palm prints.

The topology evolving of the network resulted in 118 modules each with 500 number of inputs, 37 hidden layer neurons and four output neurons in the input layer of the PMNN, 8 modules with 59 number of inputs, 23 hidden layer neurons and four output neurons in the second layer of the PMNN, then three layers of PMNN (third, fourth and fifth) has 8 replicated modules with each 32 number of inputs, 23 hidden layer neurons and four outputs and finally one decision module with 32 number of inputs, 23 hidden layer neurons, and four output neurons.

The result of each replicated modules starting from the third layer as shown in the following figure 6.1 is 41.25% for the third layer, 88.75% for the forth Layer, 95% for the fifth Layer and 96.25% for the decision layer as shown below on figure 6.1.

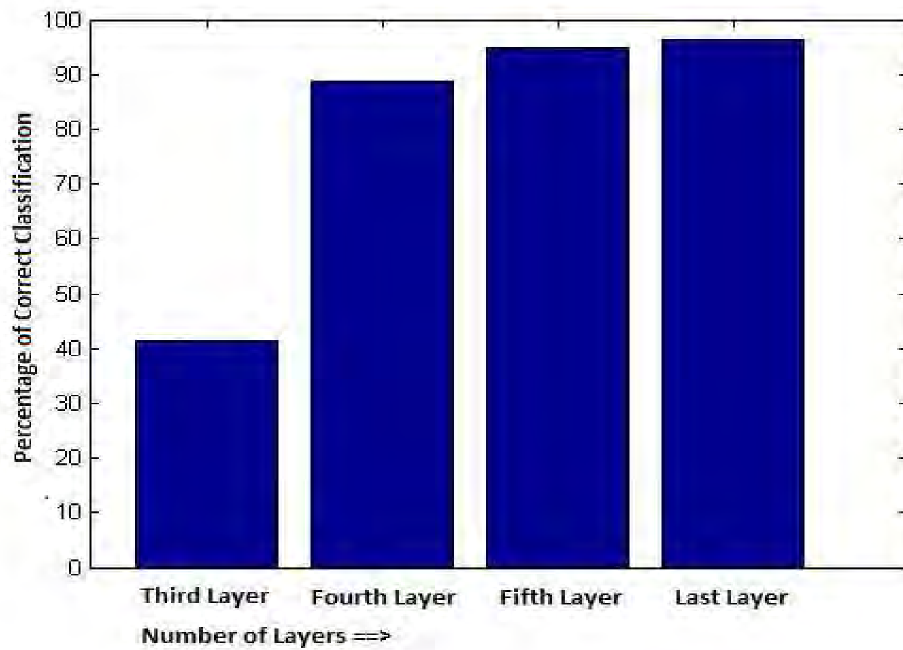

Figure 6.1: Progressive result of the Pyramidal Network for Palm-print as layer increases

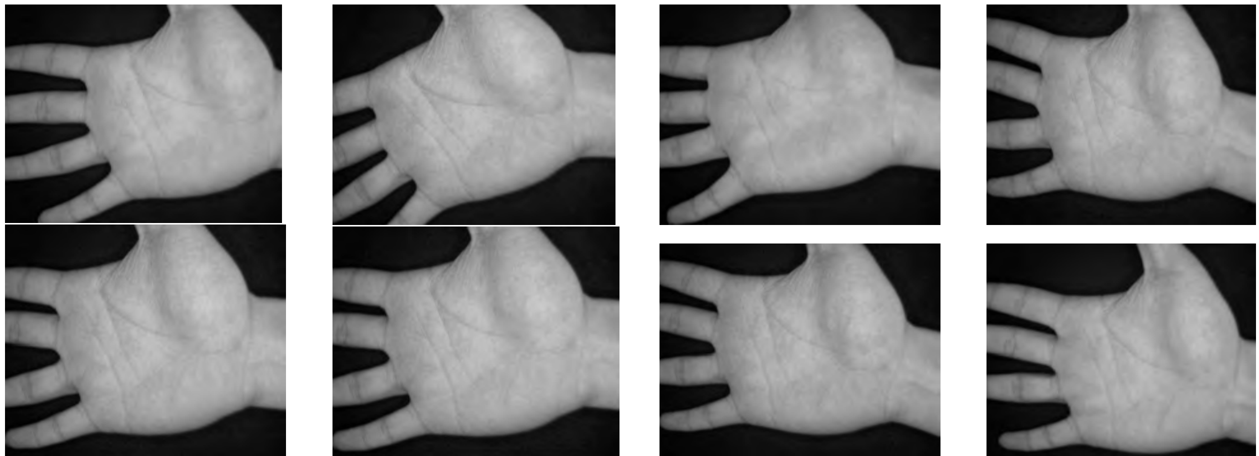

Figure 6.2: Eight Typical Palm Print Images in the database for one subject

The stopping criteria are different for different layers of PMNN. For this palm print recognition problem, the stopping criterion for the first layer of PMNN Modules is 0.55 root mean square errors. The second layer of the PMNN modules has 0.47 root mean square errors. Then the third layer has used 0.40, the fourth with 0.22 the fifth with 0.15 and the final decision module with 0.10 root mean squares.

The performance of PMNN is compared with 2 layers MNN and monolithic NN and the results are shown in fig 6.3.

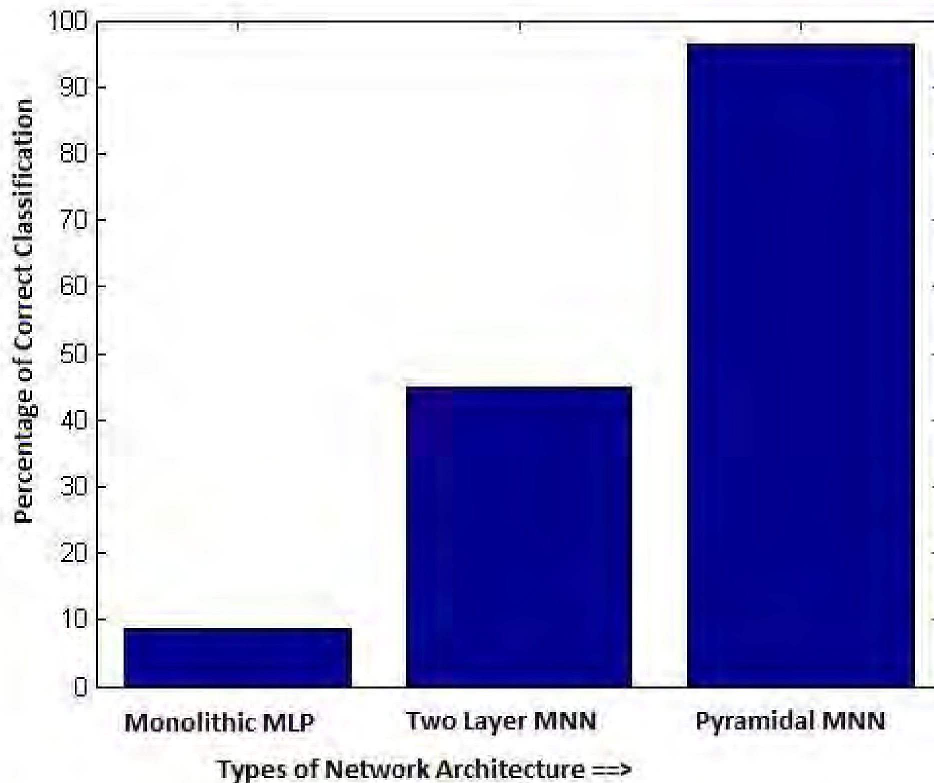

Figure 6.3: The results of monolithic MLP, the two layer architecture with input layer modules and the proposed Pyramidal Architecture with palm print recognition

The same optimization techniques have been applied to find the topology of the modules of the two layers MNN and the monolithic NN.

In addition to modular topology evolution, the number of modules in the input layer of the two layers MNN is being the same as the PMNN, i.e. 118. The input layer modules of the two layers MNN has the same topology as used in the PMNN, i.e. 500 inputs, 37 hidden layer neurons, and for output neurons. The decision module of this two layer MNN is with 472 inputs, 37 hidden layers and four outputs.

The stopping criteria of the two layers MNN is the same as with the PMNN in the input layer only i.e. 0.50 root mean square errors but about 0.42 root mean square errors for its second layer (decision module).

The module topology of the monolithic NN is with 59000 inputs, 78 hidden layer neurons and four outputs. The stopping criterion for the training of the monolithic NN is 0.54 root mean square errors.

As it is clearly shown in the figure above, the result of the proposed architecture is far better than the monolithic as well as the two layer architecture. The result of the palm print recognition is 96.25% for this proposed architecture. The two layer architecture similar to the one proposed and evaluated in [1] has yielded only 45% correct classification. The monolithic MLP has resulted in only 8.75%.

6.3.3. Iris Pattern Recognition

The images are taken from image database for biometric identification by *Institute of Automation, Chinese Academy of Sciences* for about 250 subjects in iris interval only. The training and the testing images are selected from on category of the iris image database. From these 250 subjects only right eye iris images of about 13 subjects each having 7 right eye iris images are taken. All the images are gray scale with 320x280 pixels. From these 91 selected images, 52 are used for training and all the 91 for evaluation of the error and testing.

As it is shown in figure 6.4, the result of the iris recognition is 96.75%. In all the three problems, the network does function two purposes: to *reduce the dimension* of the input vector and to *reduce errors* computing hierarchically. For example, the output of a module in the after the first hidden layer is showing 59.34% correct result for iris recognition. After the addition of one replicated layer with 8 modules taking the same output of the hidden layer as input, and feeding this replicated output to a single decision module and training, the correct classification has dramatically increased to 86.81%. Doing the same for the next layer makes about 94.51%. Replicating the third additional layer makes about 96.75%. Although the above figures differ among the three problems, the same pattern of improvement is shown from layer to the next layer with all above 95% correct classification ability for the final decision module output.

The topology evolving of the network resulted in 64 modules each with 390 number of inputs, 34 hidden layer neurons and four output neurons in the input layer of the PMNN, 8 modules with 32 number of inputs, 23 hidden layer neurons and four output neurons in the second layer of the

PMNN, then three layers of PMNN (third, fourth and fifth) 8 replicated modules with each 32 number of inputs, 23 hidden layer neurons and four outputs and finally one decision module with 32 number of inputs, 23 hidden layer neurons, and four output neurons.

The stopping criteria are different for different layers of the PMNN. For this iris recognition problem, the stopping criterion for the first layer of the PMNN Modules is 0.45 root mean square errors. The second layer of the PMNN modules has 0.42 root mean square errors. Then the third layer has used 0.38, the fourth with 0.20 the fifth with 0.15 and the final decision module with 0.10 root mean squares.

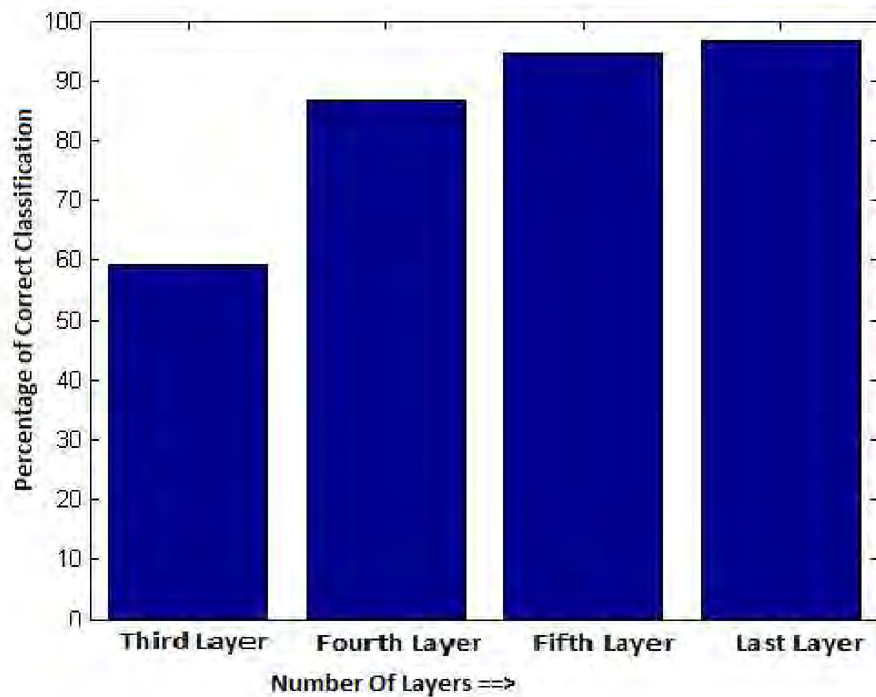

Figure 6.4: Progressive result of the Pyramidal Network for Iris Recognition as layer increases

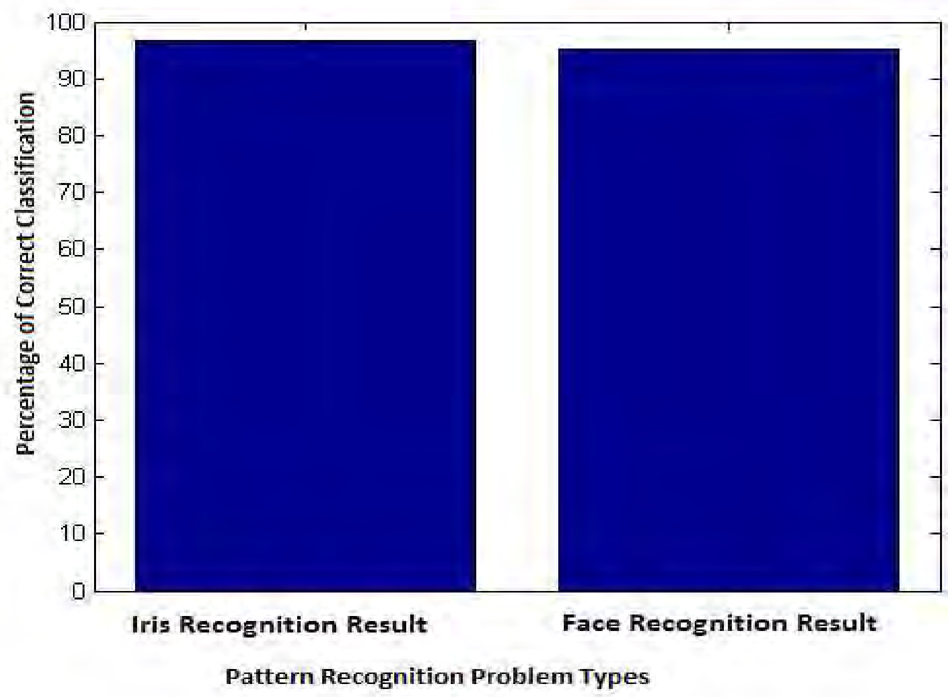

Figure 6.5: The Proposed Pyramidal Network Architecture Results on Iris and Face Recognition

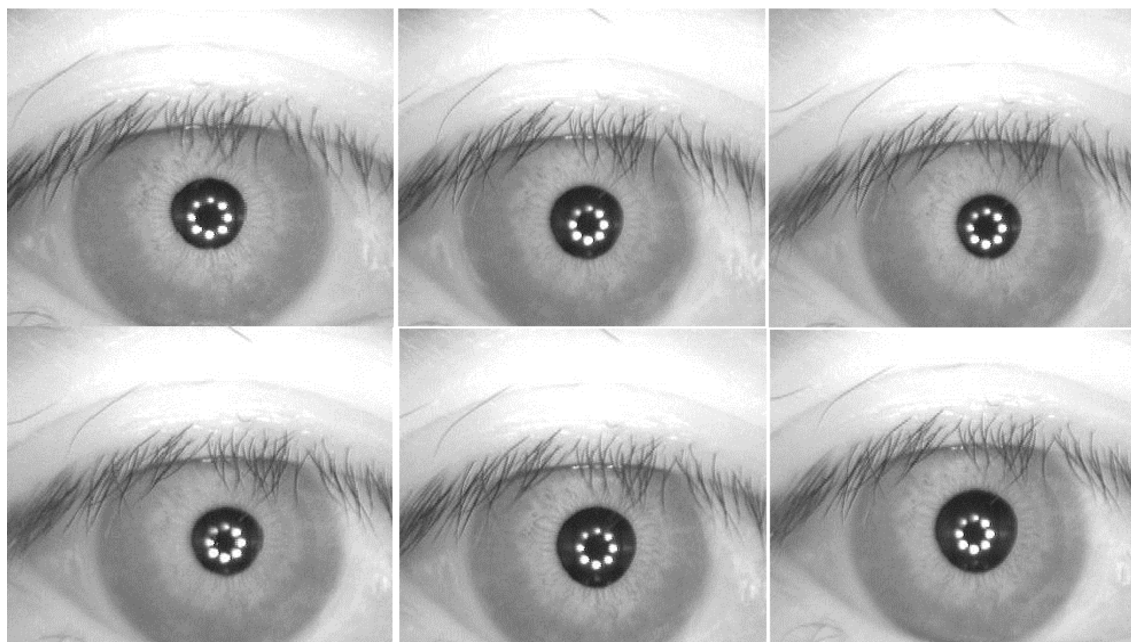

Figure 6.6: Six Typical Iris Images from the database for one subject

6.3.4. Face Recognition

The images are taken from image database for biometric identification by *Institute of Automation, Chinese Academy of Sciences* for about 500 subjects from the cropped version of the database. The training and the testing images are selected from this cropped image database. From these 500 subjects only images of about 25 subjects each having 5 images are taken. All the images are 16 bit color scale with about 130x160 pixels. From these 125 selected images, 75 are used for training and all the 125 for evaluation of the error and testing.

As it is shown in figure 6.7, the result of the face recognition is 95.25%. The output of a module in the after the first hidden layer is showing 36.8% correct result for iris recognition. After the addition of one replicated layer with 8 modules taking the same output of the hidden layer as input, and feeding this replicated output to a single decision module and training, the correct classification has dramatically increased to 71.2%. Doing the same for the next layer makes about 91.2%. Replicating the third additional layer makes about 95.25%.

The input preprocessor only considers every image as gray scale. The consideration of the images for color is not included. It is deliberately left out to check whether the network works for colored images being considered as only gray scale. Additionally, when we consider the image pixel with RGB color value, the number of the input dimension increases to 3 fold of the gray scale consideration. Here for test purpose, it was required to simplify the resulting network that would have been more complex if color value was added.

The topology evolving of the network resulted in 56 modules each with 445 number of inputs, 33 hidden layer neurons and five output neurons in the input layer of the PMNN, 8 modules with 35 number of inputs, 21 hidden layer neurons and five output neurons in the second layer of the PMNN, then two layers of PMNN (third and fourth) 8 replicated modules with each 40 number of inputs, 25 hidden layer neurons and five outputs and finally one decision module with 40 number of inputs, 25 hidden layer neurons, and five output neurons.

The stopping criteria are different for different layers of the PMNN. For this face recognition problem, the stopping criterion for the first layer of the PMNN Modules is 0.49 root mean square errors. The second layer of the PMNN modules has 0.43 root mean square errors. Then the third layer has used 0.39, the fourth with 0.20 the fifth with 0.17 and the final decision module with 0.11 root mean squares.

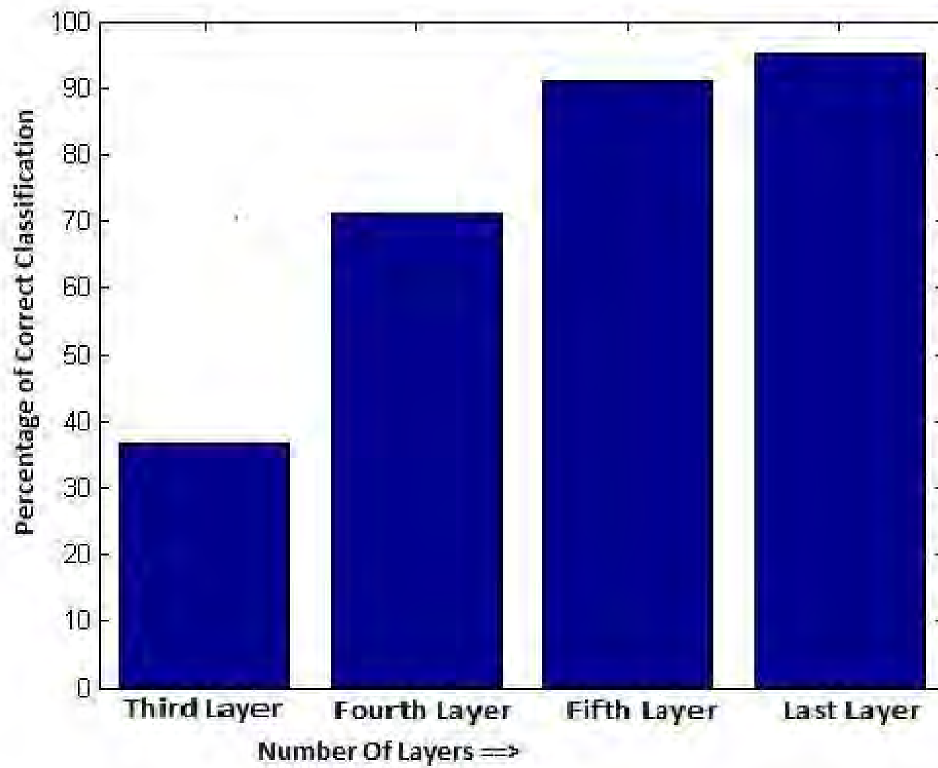

Figure 6.7: Progressive result of the Pyramidal Network for Face Recognition as layer increases

Figure 6.8: Five Typical cropped face images

6.4. Summary of the Evaluation

The proposed architecture proved to be useful for many different real world data sets. The experiments showed that the architecture performed satisfactory for all of the tested application domains.

For larger input dimensions, the PMNN architecture proved to be superior over both the monolithic MLP and the two-layer MNN in [1] in terms of generalization. The theoretical limitation for learning statistically neutral data sets did not appear in the test cases used. This problem seems to be unlikely in practice, particularly when dealing with a large number of inputs.

An important improvement is the speed of training in the proposed network as compared with the monolithic NN architecture. The training process was much faster due to the smaller number of weight connections and the splitting of the input vector into smaller multiple vectors. This effect was particularly significant in problems with high dimensional input spaces. Although the parallel training is not implemented in each layer in this work due to computational resource limitation, the training time of a module that constitutes the pyramidal network is on average about one minute. But the training time of a single monolithic MLP being fed with high dimensional input data takes more than ten minutes to be trained even with greater error set as stopping criterion. But for the same stopping criterion as the component modules, the monolithic MLP does not converge.

Considering the parallel implementation of the training for modular weight values, the total time required for the pyramidal topology is the number of layers times the time required for training one module. About six layers are used for this selected datasets to evaluate the performance of the architecture. But the former network architecture developed by [1], only has two layers and hence its training time is better than this pyramidal network architecture.
The training of the decision module proved to be very easy, often very few cycles were needed to reach a sufficiently small error value. For some experiments, the decision module converged in less than 10 seconds.

Due to the absence of parallel training, the topology search for modules and the overall network as well as the modular weight trainings of each module is done separately one after the other. By average the network evolution has taken about 12 hours. But the modular weight training of each network depends on the number of the modules in the network and nature of the data used. But on average the training of one module takes about one minute. The maximum number of modules used among the three applications is for palm print recognition. It has used 118 modules for the first layer, 8 modules for the second layer, and 3 layers with eight modules and 1 decision module that makes up to 130 modules (Since a single module's output is copied to other modules of the same replicated layer, only one module is considered for calculating the training time. Since there are 3 replicated layers, 3 modules are added instead of 24 modules). The time required for training the modules is about 130 minutes. Then the total time required for evolving the topology as well as training the modules cost about 14 hours and 10 minutes. But it has to be remembered that parallel training algorithm is not implemented here.

If computational resources are available to implement the network architecture fully automated for evolving and training the network, more search space for more accurate and optimal solution can be implemented. The result of the implementation of the network is given that is done with core i3, 4 GB RAM and 2.13 GHz clock frequency laptop. Due to the limitations in the computational resource, the training was made with crude and some limiting assumptions. For example, the optimal number of the modules in the first layer is only searched based on only one input module's accuracy that would result in some limitation. Furthermore, the searching of the number of modules is done only for the first layer. Other subsequent layers are assumed to have the number of modules by the developer. The search space is also limited to evolve the modular topology and the number of input modules. These limitations are due to the computational resource unavailability.

6.5. The Proposed Network for Other High Dimensional Input Vectors

The network simulating program is demonstrated to handle three dimensionally varying pattern recognition problems with dimensions from 20,000 to 60,000. It can be applied for classifying medical images, radar images, automatic seal and signature recognition systems and other biometric and pattern recognition problems in a wide variety of fields such as medical sciences, security system, criminal identification systems, attendance systems, defense systems and other technology and science applications.

Chapter 7

7. Conclusions and Recommendations

7.1. Conclusions

In this thesis work, a new MNN architecture known as PMNN is proposed and evaluated. The building blocks of the architecture are MLPs trained using BPA. There are two types of modules in this network: the *component modules* that constitute the input and the hidden layers of the PMNN and the *decision module* which does the final classification accepting all outputs from the modules of final hidden layer of the PMNN.

The PMNN architecture is a general architecture which can serve wide range of applications with high dimensional input vectors. In order to suit the network for new applications and train the network to address that application, an optimization technique known as PSO is applied. The PSO algorithm is used for two purposes: to search the optimal number of modules in the layers of the PMNN and to find the optimal topology of the modules from the given search space. The second purpose PSO algorithm is nested inside first purpose PSO algorithm.

The suggested training algorithm works in multiple stages depending on the number of hidden modular layers of the network. After the number of modules in each layer of the PMNN as well as the topology of the modules are evolved using PSO, each of the modules are trained for the weight-link values using BPA. The class outputs of the network are binary representations of the class numbers. All the modules of the PMNN have the same expected output as the last decision module. So during training, the expected outputs of all the modules are set with the same value for a given input vector and the modules are trained to give that output.

An object-oriented program using C# is written to simulate the behavior of the PMNN network. Program for both PSO algorithm and the training of the modules using BPA is developed. Using the simulating program, the evaluation of the PMNN architecture and its optimization for different real-world data has been carried out. The evaluation is done on three pattern recognition problems: palm print recognition, iris recognition, and face recognition. In all the three evaluations, the results found on the test data of respective applications are more than 95% accurate.

The comparison is also made with two well-known and well- researched similar architecture for palm print recognition problem. The same PSO optimization is applied to evolve topology of the monolithic MLP and the modules of the two-layer MNN in [1] as well as the number of modules in the input layer of the two-layer MNN. The generalization as well as the recognition ability of this PMNN network is found to be superior over both the monolithic MLP and the two-layer MNN architecture done by [1].

During training due to computational resource limitation, the parallel training of the modules in the same layer has not been implemented in this thesis work. If the parallel training mechanism is applied, the training time of the PMNN will be superior over the monolithic MLP network. But the training time of the two-layer MNN in [1] is better than this proposed PMNN as the number of the layers here in this PMNN network is more than the two-layer MNN architecture in [1].

The proposed architecture proved to be useful for many different real world data sets. The results which are far from comprehensive are very promising. The findings during the experiments are encouraging to search for new artificial neural networks based on parallelism and modularity.

7.2. Future Works and Recommendations

This thesis work has proposed new MNN general architecture and also showed an optimization technique on how to evolve the network depending on the nature of the data input. It has solved three high dimensional recognition problems while it has automatically evolves the network that suits most to that problem. The test has been done on 20,000 to 60,000 dimensional inputs. This work has used PSO for network topology evolution. The aim of this work is to answer the general question: Is the proposed PMNN architecture useful? The result of the experimentation has showed that the PMNN architecture is really useful.

The next step should be to investigate further issues:

- ⇨ How do the following parameters influence the performance of the network:
    - o The number of training steps in each module.
    - o The updating of learning parameter.

- o The output representation of the input and the hidden layers of the PMNN.

- ⇨ Different training algorithms may be explored that may be more effective for high dimensional inputs and larger number of weight connections.

- ⇨ Modular and overall topology evolution mechanisms can be improved. For example, the number of modules in each layer should be evaluated based on the final output of the network. But here for simplification, the output of only one input module has been used to evolve the number of modules in the first layer. For other layers, it has been assigned manually and not evolved Furthermore, the modular topology is evolved based on only one sample module in each layer in this work. But it is clearly known that the topology of an MLP depends not only on the number of inputs and number of outputs, but also on the nature of data fed to each module. Therefore, it is good to evolve the topology of each module in each layer separately independent of one another.

- ⇨ Other network topology evolving mechanisms can be applied especially those with multi-objective searching mechanisms like hierarchical genetic algorithm.

- ⇨ The behavior of the modular architecture for data sets with a larger number of classes may be analyzed.
- ⇨ The manner of connections can be best optimized. The number of modules to which each input data has to be connected can be optimized.

Appendix A: Code Sample for PSO Algorithm of Module Topology

A.1. Modular Topology Particle

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;

namespace ThesisPyramidalMNN.Particles
{
    /// <summary>
    /// Provides a basic implementation of a paricle.
    /// </summary>
    public abstract class TopologyParticle : IComparable<TopologyParticle>
    {
        #region Fields
        protected static Random _rnd = new Random();
        protected double _bestCost = double.MaxValue;
        #endregion
        //--------------------------------------------------------------------
        #region implementation
        /// <summary>
        /// The cost for this particle. The lower the better.
        /// </summary>
        public double Cost { get; set; }
        //--------------------------------------------------------------------
        /// <summary>
        /// The current position of the particle.
        /// </summary>
        public virtual int[] Position { get; set; }
        //--------------------------------------------------------------------
        /// <summary>
        /// The best position of this particle so far.
        /// </summary>
        /// <remarks>pBest.</remarks>
        public int[] BestPosition { get; protected set; }
        //--------------------------------------------------------------------
        /// <summary>
        /// Indexer.
        /// </summary>
        /// <param name="index">Index.</param>
        /// <exception cref="IndexOutOfRangeException">
        /// Der Index ist außerhalb der gültigen Grenzen.
        /// </exception>
        public int this[int index]
        {
            get { return this.Position[index]; }
            set { this.Position[index] = value; }
        }
        //--------------------------------------------------------------------
        /// <summary>
        /// The velocity of the particle.
        /// </summary>
        public int[] Velocity { get; protected set; }
        //--------------------------------------------------------------------
        /// <summary>
```

```
/// The particle swarm to whom this particle belongs.
/// </summary>
public TopologyParticleSwarming Swarm { get; protected set; }
#endregion
//----------------------------------------------------------------------
#region Methods
/// <summary>
/// Calculates the cost for this particle.
/// </summary>
public abstract void CalculateCost();
//----------------------------------------------------------------------
/// <summary>
/// Updates the history for this particle.
/// </summary>
public void UpdateHistory()
{
    if (this.Cost < _bestCost)
    {
        _bestCost = this.Cost;
        this.BestPosition = this.GetArrayCopy();
    }
}
//----------------------------------------------------------------------
/// <summary>
/// Updates the position (and velocity) of the particle.
/// </summary>
/// <param name="bestPositionOfSwarm">
/// The current best position of the particle swarm.
/// </param>
public void UpdateVelocityAndPosition(int[] bestPositionOfSwarm)
{
    if (this.BestPosition == null)
        this.UpdateHistory();

    // Determine maximum allowed velocity:
    double xmax = Math.Max(
        Math.Abs(this.Position.Min()),
        Math.Abs(this.Position.Max()));
    double vmax = this.Swarm.PercentMaximumVelocityOfSearchSpace * xmax;

    // Range-Check-Elimination: Therefore get a reference to the arrays
    // on the local stack -> improvement of speed:
    int[] localVelocity = this.Velocity;
    int[] localPosition = this.Position;
    int[] localBestPosition = this.BestPosition;

    for (int i = 0; i < localVelocity.Length; i++)
    {
        // Factors for calculating the velocity:
        double c1 = this.Swarm.TendencyToOwnBest;
        double r1 = _rnd.NextDouble();
        double c2 = this.Swarm.TendencyToGlobalBest;
        double r2 = _rnd.NextDouble();
        double m = this.Swarm.Momentum;

        // New velocity of the particle:
        double newVelocity =
```

```
                    m * localVelocity[i] +
                    c1 * r1 * (localBestPosition[i] - localPosition[i]) +
                    c2 * r2 * (bestPositionOfSwarm[i] - localPosition[i]);

                // Limit the velocity to the maximum value:
                if (newVelocity > vmax)
                    newVelocity = vmax;
                if (newVelocity < -vmax)
                    newVelocity = -vmax;

                // Assign new velocity and calculate the new position:
                localVelocity[i] = (int)newVelocity;
                localPosition[i] += (int)localVelocity[i];
            }
        }
        #endregion
        //----------------------------------------------------------------------
        #region Private Methods
        /// <summary>
        /// Gets a copy of the current solution vector.
        /// </summary>
        private int[] GetArrayCopy()
        {
            int[] tmp = new int[this.Position.Length];
            this.Position.CopyTo(tmp, 0);

            return tmp;
        }
        #endregion
        //----------------------------------------------------------------------
        #region IComparable<Particle> Member
        /// <summary>
        /// Compares to particles. Used for sorting.
        /// </summary>
        public int CompareTo(TopologyParticle other)
        {
            if (other == null)
                throw new ArgumentNullException("other");
            //------------------------------------------------------------------
            if (this == other || this.Cost == other.Cost)
                return 0;

            if (this.Cost > other.Cost)
                return 1;
            else
                return -1;
        }
        #endregion
    }
}
```

A.2. Modular Topology Particle Swarm

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
```

```
using ThesisPyramidalMNN.Particles;
using ThesisPyramidalMNN.MNNMinimizingFunction;

namespace ThesisPyramidalMNN.Particles
{
    public abstract class TopologyParticleSwarming
    {
        #region Fields
              protected static Random _rnd = new Random();
              #endregion
              //-----------------------------------------------------------------------
              #region implementation
              /// <summary>
              /// The particles of this swarm.
              /// </summary>
              public TopologyParticle[] Particles { get; set; }
              //-----------------------------------------------------------------------
              /// <summary>
              /// Indexer.
              /// </summary>
              public TopologyParticle this[int index]
              {
                     get { return this.Particles[index]; }
                     set { this.Particles[index] = value; }
              }
              //-----------------------------------------------------------------------
              /// <summary>
              /// The number of particles in the swarm.
              /// </summary>
              public int SwarmSize
              {
                     get
                     {
                            if (this.Particles == null)
                                   return 0;

                            return this.Particles.Length;
                     }
              }
              //-----------------------------------------------------------------------
              /// <summary>
              /// The cost of the current best particle of the swarm.
              /// </summary>
              public virtual double Cost { get { return this[0].Cost; } }
              //-----------------------------------------------------------------------
              /// <summary>
              /// The cost of the best particle so far.
              /// </summary>
              public double BestCost { get; protected set; }
              //-----------------------------------------------------------------------
              /// <summary>
              /// The current best of position of any particle in the swarm.
              /// </summary>
              /// <remarks>gBest.</remarks>
              public int[] CurrentBestPosition { get { return this[0].Position; } }
              //-----------------------------------------------------------------------
              /// <summary>
```

```
/// The best position of the swarm so far.
/// </summary>
public int[] BestPosition { get; protected set; }
//-------------------------------------------------------------------
private double _tendencyToOwnBest = 1;
/// <summary>
/// Accelerationcoefficient c1. Default value: 1
/// </summary>
public double TendencyToOwnBest
{
       get { return _tendencyToOwnBest; }
       set { _tendencyToOwnBest = value; }
}
//-------------------------------------------------------------------
private double _tendencyToGlobalBest = 1;
/// <summary>
/// Accelerationcoefficient c2. Default value: 1
/// </summary>
public double TendencyToGlobalBest
{
       get { return _tendencyToGlobalBest; }
       set { _tendencyToGlobalBest = value; }
}
//-------------------------------------------------------------------
private double _momentum = 1.05;
/// <summary>
/// Momentum. Default value: 1.05
public double Momentum
{
       get { return _momentum; }
       set { _momentum = value; }
}
//-------------------------------------------------------------------
private double _percentMaximumVelocityOfSearchSpace = 1;
/// <summary>
/// Factor for maximum allowed velocity.
/// vmax = k * xmax mit k in [0.1, 1].
/// Defaults to 1.
/// </summary>
public double PercentMaximumVelocityOfSearchSpace
{
       get { return _percentMaximumVelocityOfSearchSpace; }
       set { _percentMaximumVelocityOfSearchSpace = value; }
}
//-------------------------------------------------------------------
private bool _useGlobalOptimum = false;
/// <summary>
/// Determines wheter to use the global optimum or the current best
/// solution for updating the particles. Default: false
/// </summary>
/// <remarks>
/// This can be an improvement on speed (converges faster) but
/// the possibility to get stucked in a local optimum raises.
/// <para>
/// The global versiön can be used to get a rough solution and the
/// refine with the current best version.
/// </para>
```

```
/// </remarks>
public bool UseGlobalOptimum
{
        get { return _useGlobalOptimum; }
        set { _useGlobalOptimum = value; }
}
#endregion
//-----------------------------------------------------------------------
#region constructor
/// <summary>
/// Constructor.
/// </summary>
protected TopologyParticleSwarming()
{
        this.BestCost = double.MaxValue;
}
#endregion
//-----------------------------------------------------------------------
#region Methods
/// <summary>
/// Sorts the particles according their cost.
/// </summary>
public void SortParticles()
{
        Array.Sort(this.Particles);
}
//-----------------------------------------------------------------------
/// <summary>
/// Does one iteration of the algorithm.
/// </summary>
public virtual void Iteration()
{
        // Foreach particle calculate the cost and update the history:
        var localParticles = this.Particles;     // Range-Check-Elimination
        for (int i = 0; i < localParticles.Length; i++)
        {
                localParticles[i].CalculateCost();
                localParticles[i].UpdateHistory();
        }

        // Sort according to fitness:
        this.SortParticles();

        // Update history of the swarm:
        if (this.Cost < this.BestCost)
        {
                this.BestCost = this.Cost;
                this.BestPosition = this[0].BestPosition;
        }

        // Determine new velocity and position of the particles in
        // the swarm:
        for (int i = 0; i < localParticles.Length; i++)
                if (_useGlobalOptimum)
localParticles[i].UpdateVelocityAndPosition(BestPosition);
                else
```

```
        localParticles[i].UpdateVelocityAndPosition(this[0].Position);
                }
                #endregion
        }
}
```

A.3. Modular Topology Particle Function Minimizing

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using ThesisPyramidalMNN.Particles;
using ThesisPyramidalMNN.MNNMinimizingFunction;

namespace ThesisPyramidalMNN.MNNMinimizingFunction
{
    class FunctionMinimizingTopologyParticle:TopologyParticle
    {
        private ModularTopologyFunction _function;
        //---------------------------------------------------------------------
        public FunctionMinimizingTopologyParticle(
            ModularTopologyFunction function,
            TopologyParticleSwarming swarm,
            int[] position,
            int[] velocity)
        {
            _function = new ModularTopologyFunction();
            _function = function;
            this.Swarm = swarm;
            this.Position = position;
            this.Velocity = velocity;

            this.CalculateCost();
        }
        //---------------------------------------------------------------------
        public override void CalculateCost()
        {
            if (_function.DecisionMode)
                this.Cost = _function.Function(this.Position,
_function.PyranidalTopology, _function.AllModuleWeights);
            else
                this.Cost = _function.Function(this.Position);

        }
    }
}
```

A.4. Modular Topology Particle Swarm Function Minimizing

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using ThesisPyramidalMNN.Particles;
using ThesisPyramidalMNN.MNNMinimizingFunction;
```

```
namespace ThesisPyramidalMNN.MNNMinimizingFunction
{
    public sealed class FunctionMinimizingTopologyParticleSwarm:TopologyParticleSwarming
    {
        public FunctionMinimizingTopologyParticleSwarm(
            ModularTopologyFunction function, int partDimSize,
            int swarmSize)
        {
            // Create the swarm:
            this.InitSwarm(swarmSize, partDimSize, function);

            // Sort according to the cost of each particle:
            this.SortParticles();
        }
        //--------------------------------------------------------------------------------
---------
        public void InitSwarm(int swarmSize, int partDimSize, ModularTopologyFunction
function)
        {
            // Create the array of particles:
            this.Particles = new FunctionMinimizingTopologyParticle[swarmSize];
            for (int i = 0; i < swarmSize; i++)
            {
                int[] particlePosition = new int[partDimSize];
                int[] particleVelocity = new int[partDimSize];
                //Assumed ranges of values of number of neurons per each layer in the
modules to be within [50,100].
                for (int j = 0; j < partDimSize; j++)
                {
                    particlePosition[j] = _rnd.Next(20,100);
                    particleVelocity[j] = _rnd.Next(100);
                }
                this[i] = new FunctionMinimizingTopologyParticle(function, this,
particlePosition, particleVelocity);
            }
        }
    }
}
```

A.5. Modular Topology Function

```
  public class ModularTopologyFunction //: FunctionBase
    {
        private int firstLayer;
        private int lastLayer;
        private ModularNeuralNet mnn;
        private ModularWeightFunction mw;
        private ModularCompute mc;
        private Boolean decisionMode;
        private int[] pyranidalTopology;
        private float[][] allModuleWeights;

        //Constructors
        public ModularTopologyFunction()
        { }
```

```
public ModularTopologyFunction(ModularNeuralNet mnet)
{
    mnn = new ModularNeuralNet();
    mnn = mnet;
}
public int FirstLayer
{
    get { return firstLayer; }
    set { firstLayer = value; }
}
public ModularNeuralNet Mnn
{
    get { return mnn; }
    set { mnn = value; }
}
public int LastLayer
{
    get { return lastLayer; }
    set { lastLayer = value; }
}
public Boolean DecisionMode
{
    get { return decisionMode; }
    set { decisionMode = value; }
}
public int[] PyranidalTopology
{
    get { return pyranidalTopology; }
    set { pyranidalTopology = value; }
}
public float[][] AllModuleWeights
{
    get { return allModuleWeights; }
    set { allModuleWeights = value; }
}
public double Function(int[] position, int[] calcTop, float[][] weights)
{
    int[] newPosition = new int[2];
    newPosition[0] = position[0];
    newPosition[1] = lastLayer;
    mnn.LayerNumbers = newPosition;
    mc = new ModularCompute(mnn);
    mc.CalcTopology = new int[calcTop.Length];
    mc.CalcTopology = calcTop;
    mc.Weights = new float[weights.Length][];
    mc.Weights[0] = new float[weights[0].Length];
    mc.Weights = weights;
    //mc.RunDecisionModuleWeight();
    float[] bestWeightArray = new float[mc.BestSolution.Length];
    bestWeightArray = mc.BestSolution;
    mw = new ModularWeightFunction(mnn);
    double z = mw.Function(bestWeightArray);
    return z;
}
public double Function(int[] position)
{
    int[] newPosition = new int[2];
```

```
            newPosition[0] = position[0];
            //newPosition[1] = position[1];
            //newPosition[2] = position[2];
            newPosition[1] = lastLayer;
            mnn.LayerNumbers = newPosition;
            mc = new ModularCompute(mnn);
            mc.RunModularWeight();
            float[] bestWeightArray = new float[mc.BestSolution.Length];
            bestWeightArray = mc.BestSolution;
            double z = mc.Cost;
            return z;
        }

    }
```

Appendix B: Code Sample for PSO Algorithm of Pyramidal Topology

B.1. Pyramidal Topology Particle

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;

namespace ThesisPyramidalMNN.Particles
{
    /// <summary>
    /// Provides a basic implementation of a paricle.
    /// </summary>
    public abstract class PyramidalTopologyParticle :
IComparable<PyramidalTopologyParticle>
    {
        #region Fields
        protected static Random _rnd = new Random();
        protected double _bestCost = double.MaxValue;
        #endregion
        //---------------------------------------------------------------------
        #region implementation
        /// <summary>
        /// The cost for this particle. The lower the better.
        /// </summary>
        public double Cost { get; set; }
        //---------------------------------------------------------------------
        /// <summary>
        /// The current position of the particle.
        /// </summary>
        public virtual int Position { get; set; }
        //---------------------------------------------------------------------
        /// <summary>
        /// The best position of this particle so far.
        /// </summary>
        /// <remarks>pBest.</remarks>
        public int BestPosition { get; protected set; }
        //---------------------------------------------------------------------
```

```
/// <summary>
/// Indexer.
/// </summary>
/// <param name="index">Index.</param>
/// <exception cref="IndexOutOfRangeException">
/// Der Index ist außerhalb der gültigen Grenzen.
/// </exception>
public int this[int index]
{
    get { return this.Position; }
    set { this.Position = value; }
}
//-----------------------------------------------------------------------
/// <summary>
/// The velocity of the particle.
/// </summary>
public int Velocity { get; protected set; }
//-----------------------------------------------------------------------
/// <summary>
/// The particle swarm to whom this particle belongs.
/// </summary>
public PyramidalParticleSwarming Swarm { get; protected set; }
#endregion
//-----------------------------------------------------------------------
#region Methods
/// <summary>
/// Calculates the cost for this particle.
/// </summary>
public abstract void CalculateCost();
//-----------------------------------------------------------------------
/// <summary>
/// Updates the history for this particle.
/// </summary>
public void UpdateHistory()
{
    if (this.Cost < _bestCost)
    {
        _bestCost = this.Cost;
        this.BestPosition = this.GetArrayCopy();
    }
}
//-----------------------------------------------------------------------
/// <summary>
/// Updates the position (and velocity) of the particle.
/// </summary>
/// <param name="bestPositionOfSwarm">
/// The current best position of the particle swarm.
/// </param>
public void UpdateVelocityAndPosition(int bestPositionOfSwarm)
{
    if (this.BestPosition == 0)
        this.UpdateHistory();

    // Determine maximum allowed velocity:
    double xmax = Math.Max(
        Math.Abs(this.Position),
        Math.Abs(this.Position));
```

```
    double vmax = this.Swarm.PercentMaximumVelocityOfSearchSpace * xmax;

    // Range-Check-Elimination: Therefore get a reference to the arrays
    // on the local stack -> improvement of speed:
    double localVelocity = this.Velocity;
    int localPosition = this.Position;
    int localBestPosition = this.BestPosition;

        // Factors for calculating the velocity:
        double c1 = this.Swarm.TendencyToOwnBest;
        double r1 = _rnd.NextDouble();
        double c2 = this.Swarm.TendencyToGlobalBest;
        double r2 = _rnd.NextDouble();
        double m = this.Swarm.Momentum;

        // New velocity of the particle:
        double newVelocity =
            m * localVelocity +
            c1 * r1 * (localBestPosition - localPosition) +
            c2 * r2 * (bestPositionOfSwarm - localPosition);

        // Limit the velocity to the maximum value:
        if (newVelocity > vmax)
            newVelocity = vmax;
        if (newVelocity < -vmax)
            newVelocity = -vmax;

        // Assign new velocity and calculate the new position:
        localVelocity = newVelocity;
        localPosition += (int) localVelocity;

}
#endregion
//----------------------------------------------------------------------
#region Private Methods
/// <summary>
/// Gets a copy of the current solution vector.
/// </summary>
private int GetArrayCopy()
{
    int tmp = Position;

    return tmp;
}
#endregion
//----------------------------------------------------------------------
#region IComparable<Particle> Member
/// <summary>
/// Compares to particles. Used for sorting.
/// </summary>
public int CompareTo(PyramidalTopologyParticle other)
{
    if (other == null)
        throw new ArgumentNullException("other");
    //------------------------------------------------------------------
    if (this == other || this.Cost == other.Cost)
        return 0;
```

```
            if (this.Cost > other.Cost)
                return 1;
            else
                return -1;
        }
        #endregion
    }
}
```

B.2. Pyramidal Topology Particle Swarm

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;

namespace ThesisPyramidalMNN.Particles
{
    public abstract class PyramidalParticleSwarming
    {
        #region Fields
              protected static Random _rnd = new Random();
              #endregion
              //----------------------------------------------------------------------
              #region implementation
              /// <summary>
              /// The particles of this swarm.
              /// </summary>
              public PyramidalTopologyParticle[] Particles { get; set; }
              //----------------------------------------------------------------------
              /// <summary>
              /// Indexer.
              /// </summary>
              public PyramidalTopologyParticle this[int index]
              {
                     get { return this.Particles[index]; }
                     set { this.Particles[index] = value; }
              }
              //----------------------------------------------------------------------
              /// <summary>
              /// The number of particles in the swarm.
              /// </summary>
              public int SwarmSize
              {
                     get
                     {
                            if (this.Particles == null)
                                   return 0;

                            return this.Particles.Length;
                     }
              }
              //----------------------------------------------------------------------
              /// <summary>
              /// The cost of the current best particle of the swarm.
              /// </summary>
```

```
public virtual double Cost { get { return this[0].Cost; } }
//--------------------------------------------------------------------
/// <summary>
/// The cost of the best particle so far.
/// </summary>
public double BestCost { get; protected set; }
//--------------------------------------------------------------------
/// <summary>
/// The current best of position of any particle in the swarm.
/// </summary>
/// <remarks>gBest.</remarks>
public int CurrentBestPosition { get { return this[0].Position; } }
//--------------------------------------------------------------------
/// <summary>
/// The best position of the swarm so far.
/// </summary>
public int BestPosition { get; protected set; }
//--------------------------------------------------------------------
private double _tendencyToOwnBest = 1;
/// <summary>
/// Accelerationcoefficient c1. Default value: 1
/// </summary>
public double TendencyToOwnBest
{
       get { return _tendencyToOwnBest; }
       set { _tendencyToOwnBest = value; }
}
//--------------------------------------------------------------------
private double _tendencyToGlobalBest = 1;
/// <summary>
/// Accelerationcoefficient c2. Default value: 1
/// </summary>
public double TendencyToGlobalBest
{
       get { return _tendencyToGlobalBest; }
       set { _tendencyToGlobalBest = value; }
}
//--------------------------------------------------------------------
private double _momentum = 1.05;
/// <summary>
/// Momentum. Default value: 1.05
public double Momentum
{
       get { return _momentum; }
       set { _momentum = value; }
}
//--------------------------------------------------------------------
private double _percentMaximumVelocityOfSearchSpace = 1;
/// <summary>
/// Factor for maximum allowed velocity.
/// vmax = k * xmax mit k in [0.1, 1].
/// Defaults to 1.
/// </summary>
public double PercentMaximumVelocityOfSearchSpace
{
       get { return _percentMaximumVelocityOfSearchSpace; }
       set { _percentMaximumVelocityOfSearchSpace = value; }
```

```
}
//-----------------------------------------------------------------------
private bool _useGlobalOptimum = false;
/// <summary>
/// Determines wheter to use the global optimum or the current best
/// solution for updating the particles. Default: false
/// </summary>
/// <remarks>
/// This can be an improvement on speed (converges faster) but
/// the possibility to get stucked in a local optimum raises.
/// <para>
/// The global versiön can be used to get a rough solution and the
/// refine with the current best version.
/// </para>
/// </remarks>
public bool UseGlobalOptimum
{
	get { return _useGlobalOptimum; }
	set { _useGlobalOptimum = value; }
}
#endregion
//-----------------------------------------------------------------------
#region constructor
/// <summary>
/// Constructor.
/// </summary>
protected PyramidalParticleSwarming()
{
	this.BestCost = double.MaxValue;
}
#endregion
//-----------------------------------------------------------------------
#region Methods
/// <summary>
/// Sorts the particles according their cost.
/// </summary>
public void SortParticles()
{
	Array.Sort(this.Particles);
}
//-----------------------------------------------------------------------
/// <summary>
/// Does one iteration of the algorithm.
/// </summary>
public virtual void Iteration()
{
	// Foreach particle calculate the cost and update the history:
	var localParticles = this.Particles;	// Range-Check-Elimination
	for (int i = 0; i < localParticles.Length; i++)
	{
		localParticles[i].CalculateCost();
		localParticles[i].UpdateHistory();
	}

	// Sort according to fitness:
	this.SortParticles();
```

```
                    // Update history of the swarm:
                    if (this.Cost < this.BestCost)
                    {
                           this.BestCost = this.Cost;
                           this.BestPosition = this[0].BestPosition;
                    }

                    // Determine new velocity and position of the particles in
                    // the swarm:
                    for (int i = 0; i < localParticles.Length; i++)
                           if (_useGlobalOptimum)

       localParticles[i].UpdateVelocityAndPosition(BestPosition);
                           else

       localParticles[i].UpdateVelocityAndPosition(this[0].Position);
             }
             #endregion
    }
}
```

B.3. Pyramidal Topology Particle Function Minimizing

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using ThesisPyramidalMNN.Particles;
using ThesisPyramidalMNN.MNNMinimizingFunction;

namespace ThesisPyramidalMNN.MNNMinimizingFunction
{
    public sealed class FunctionMinimizingPyramidalParticle:PyramidalTopologyParticle
    {
        private PyramidalTopology _function;
        //----------------------------------------------------------------------
        public FunctionMinimizingPyramidalParticle(
            PyramidalTopology function,
            PyramidalParticleSwarming swarm,
            int position,
            int velocity)
        {
            _function = new PyramidalTopology();
            _function = function;
            this.Swarm = swarm;
            this.Position = position;
            this.Velocity = velocity;

            this.CalculateCost();
        }
        //----------------------------------------------------------------------
        public override void CalculateCost()
        {
            int[] newPosition = new int[1];
            newPosition[0] = this.Position;
            this.Cost = _function.Function(newPosition);
```

```
        }
    }
}
```

B.4. Pyramidal Topology Particle Swarm Function Minimizing

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using ThesisPyramidalMNN.Particles;

namespace ThesisPyramidalMNN.MNNMinimizingFunction
{
    public sealed class PyramidalFunctionMinimizingSwarm:PyramidalParticleSwarming
    {
         public PyramidalFunctionMinimizingSwarm(
            PyramidalTopology function, int swarmSize)
        {
            // Create the swarm:
            this.InitSwarm(swarmSize, function);

            // Sort according to the cost of each particle:
            this.SortParticles();
        }
        //-----------------------------------------------------------------------------
---------
         public void InitSwarm(int swarmSize, PyramidalTopology function)
        {
            // Create the array of particles:
            this.Particles = new FunctionMinimizingPyramidalParticle[swarmSize];
            for (int i = 0; i < swarmSize; i++)
            {
                //Assumed ranges of values of the number of input modules.
                int particlePosition = _rnd.Next(120, 200);
                int particleVelocity = _rnd.Next(100);

                this[i] = new FunctionMinimizingPyramidalParticle(function, this,
particlePosition, particleVelocity);
            }
        }
    }
}
```

B.4. Pyramidal Topology Function

```
//For Overall Pyramidal Topology Search
    //--------------------------------------------------------------------------
    public class PyramidalTopology //: FunctionBase
    {
        private List<float[]> _solutions;
        private float[] _bestSolution;
        private float[][] solusions;
        private float[][] tempOutput;
        private float[] nextInput;
        private ReaderWriterLockSlim _slimLock = new ReaderWriterLockSlim();
        private FunctionMinimizingParticleSwarm _swarm;
```

```
        private FuncMinimizingParticleSwarm swarm;
        private const int SWARM_SIZE = 20;
        private int partDimension;
        private int[] calculatedTopology;
        private PyramidalMnnNet pmnn;
        private PyramidalCompute pc;
        private ModularCompute mc;
        private ModularNeuralNet mnn;
        private ModularWeightFunction mw;
        private ModularTopologyFunction mtf;
        private FinalPyramidalTopology fpt;
        private ModuleWeights mdw;
        private PyramidalMNN formPmnn;

        public double Function(int[] pyramidalTopology)
        {
            pmnn = new PyramidalMnnNet(pyramidalTopology[0]);
            pmnn.inputValueSetting(0,0);
            calculatedTopology = pmnn.determineTopology();
            //calculatedTopology[0] = 118;
            calculatedTopology[1] = 8;
            fpt = new FinalPyramidalTopology();
            fpt.PyramidalFinalTopology = new int[calculatedTopology.Length];
            fpt.PyramidalFinalTopology = calculatedTopology;
            pmnn.inputDivideToModules();
            //calculatedTopology[0] = 118;
            //calculatedTopology[1] = 2;
            int inputLenght = pmnn.InputsDivided.Length;
            int inputLengthSec = pmnn.InputsDivided[0].Length;
            float[][] devidInput = new float[inputLenght][];
            devidInput[0] = new float[inputLengthSec];
            devidInput = pmnn.InputsDivided;
            float[] expectedOutput = pmnn.ExpectedOutput;
            mnn = new ModularNeuralNet();
            mnn.InputArray = devidInput[0];
            mnn.NoOfInputs = devidInput[0].Length;
            fpt.NoOfInputsPerComModule = inputLenght;
            int ouptputLegnth = expectedOutput.Length;
            mnn.ExpectedOutputArray = pmnn.ExpectedOutput;
            mtf = new ModularTopologyFunction();
            mtf.FirstLayer = pmnn.PerModuleInputs;
            mtf.LastLayer = expectedOutput.Length;
            mnn.NoOfInputModules = calculatedTopology[0];
            mnn.InputMod = true;
            mnn.CalculatedTopology = new int[calculatedTopology.Length];
            mnn.CalculatedTopology = calculatedTopology;
            pc = new PyramidalCompute(mnn);
            pc.RunModularTopology(mtf);
            int[] bestTopology = new int[pc.BestSolution.Length];
            bestTopology = pc.BestSolution;
            double cost = pc.Cost;
            return cost;
        }
}
```

8. References

1. Schdmit Albrecht, “A MNN Architecture with Additional Generalization Abilities for High Dimensional Input Vectors”: Thesis Work, 1996.
2. Farooq Azam, “Biologically Inspired MNNs”: PhD Dissertation, Virginia Tech. 2000
3. Patricia Melin, Claudia Gonzalez, Diana Bravo hryrthg, “MNNs with Fuzzy Sugeno Integral for Pattern Recognition in Annual Meeting of the North American Fuzzy Information Processing Society”, NAFIPS 2005 – 2005.
4. Patricia Melin, Felma Gonzalez, Gabriela Martinez and Oscar Castillo, “Optimization of MNNs with Fuzzy Integration using Genetic Algorithms Applied to Pattern Recognition in Annual Meeting of the North American Fuzzy Information Processing Society”, NAFIPS 2005 – 2005.
5. Bart L.M. Happel and Jacob M.J. Murre, “The Design and Evolution of MNN Architectures”, IEEE Transactions on Neural Networks, 1994, 7, 985-1004.
6. Rajeev Kumar and Peter Rockett, “Multiobjective Genetic Algorithm Partitioning for Hierarchical Learning of High-Dimensional Pattern Spaces: A Learning-Follows-Decomposition Strategy”, IEEE Transactions on Neural Networks, Vol. 9, No. 5, September 1998
7. Omar Blanchet, Martha Ramirez, Maribel Gutierrez, Jassiny Quintero, Alejandra Mancilla and Patricia Melin, “A Hybrid Approach with MNNs and Fuzzy Logic for Time Series Prediction”.
8. Peter Dürr, Claudio Mattiussi, and Dario Floreano, “Genetic Representation and Evolvability of Modular Neural Controllers”, IEEE Computational Intelligence Magazine, August 2010.
9. Qiangfu Zhao, “Modeling and Evolutionary Learning of MNNs”, The University of Aizu
10. Andrea Di Ferdinando, Raffaele Calabretta and Domenico Parisi, “Evolving Modular Architectures for Neural Networks”, 2000.
11. Viswanath Ramamurti and Joydeep Ghosh, “Flexible Modular Architecture for Changing Environments”, University of Texas.

12. Seiichi Ozawa, Kazuyoshi Tsutsumi and Norio Baba, "Design of MNN Architectures Using Genetic Algorithms".
13. Shailesh Kumar and Joydeep Ghosh, "GAMLS: A Generalized Framework for Associative Modular Learning Systems".
14. Jürgen Fritsch, "MNNs for Speech Recognition", 1996.
15. Ricardo Téllez and Cecilio Angulo, "Modularity in Artificial Neural Networks", Technical University of Catalonia, Spain.
16. G. Barna and K. Kaski, "Choosing optimal network structure", Proceedings of the International Neural Network Conference (INNC90), pp. 890-893, 1990.
17. O. Castillo and P. Melin, "Hybrid Intelligent Systems for Time Series Prediction using Neural Networks, Fuzzy Logic and Fractal Theory", IEEE Transactions on Neural Networks, Vol.13, no. 6, pp. 1395-1408, 2002.
18. Fhivio de Almeida e Silva, Guilhenne Bittencourt, Mauro Roisenberg, Jorge M. Barreto, Renato C.Vieira, Dennis K. Coelho, "Behavior Implementation in Autonomous Agents using Modular and Hierarchical Neural Networks", Proceedings of the 2004 IEEE Conference on Robotics, Automation and Mechatronics Singapore, 1-3 December, 2004.
19. Changlin Cai, Zhongzhi Shi, " A Modular Neural Network Architecture with Approximation Capability and Its Applications", Proceedings of the Second IEEE International Conference on Cognitive Informatics (ICCI'03), 0-7695-1986-5/03, 2003
20. Bhupesh Gour, T. K. Bandopadhyaya, Ravindra Patel, "ART and Modular Neural Network Architecture for multilevel Categorization and Recognition of Fingerprints", Third International Conference on Knowledge Discovery and Data Mining, 2010
21. Sankar K. Pal, Pabitra Mitra, "Rough-Fuzzy MLP: Modular Evolution, Rule Generation, and Evaluation", IEEE Transactions on knowledge and Data, Vol. 15, No. 1, January/February 2003
22. Yanlai Li, Kuanquan Wang, Tao Li, "Modular Neural Network Structure with Fast Training/Recognition Algorithm for Pattern Recognition", *School of Computer Science and Technology, Harbin Institute of Technology,* China
23. MNNs, Wikipedia, the free encyclopedia.
24. Particle Swarm Optimization, http://www.swarmintelligence.org/tutorials.php. (Accessed on 02/14/2011)

25. R. Reed, “Pruning algorithms”, *IEEE Transactions on Neural Networks*: a survey, 4(5):740–747, 1993.
26. Tin-Yau Kwok and Dit-Yan Yeung, “Constructive algorithms for structure learning in feedforwardneural networks for regression problems”, *IEEE Transactions on Neural Networks*, 8(3):630–645, 1997.
27. de Smith, Goodchild, Longley ,“Learning and back-propagation for MLPs”, Geospatial Analysis - a comprehensive guide, 3rd edition © 2006-2011, http://www.spatialanalysisonline.com/output/html/Learningandback-propagationforMLPs.html
28. Hema Chandrasekaran, “Training of MLP using Levenberg-Marquardt Algorithm”, Image Processing and Neural Networks laboratory , University of Texas at Arlington
29. http://www-ee.uta.edu/eeweb/ip/Software/OldSW/Lm_mlp.txt Date of Retrieval 02/18/2011
30. Data Mining Statisticall software.http://www.statsoft.com (Accessed on 20/07/2011)
31. J. V. Stone and C. J, “Thorton. Can Artificial Neural Networks Discover Useful Regularities?”, In: Artificial Neural Networks, Conference Publication No. 409 IEE. Pages 201-205. 26-27 June 1995.
32. Sanjeev S. Malalur and Michael T. Manry, “Multiple optimal learning factors for feed-forward networks”, Department of Electrical Engineering, University of Texas at Arlington, Arlington TX 76013: Research work, March, 2010
33. Hung-Han Chen, “The Turning Points on MLP’s Error Surface”, F. Sun et al. (Eds.): ISNN 2008, Part I, LNCS 5263, pp. 512–520, 2008.
34. L. Prechelt, “A Quantitaive Study of Experimental Neural Network Learning Algorithm Evaluation Practicies”, In: Artificial Neural Networks, Conference Publication No. 409 IEE. Pages 223-227. 26-28 June 1995.
35. CASIA Palmprint Database collected by the Chinese Academy of Sciences’ Institute of Automation (CASIA). http://biometrics.idealtest.org/ (Accessed on 19/06/11)
36. CASIA Face Image Database collected by the Chinese Academy of Sciences’ Institute of Automation (CASIA). http://biometrics.idealtest.org/ (Accessed on 20/06/11)
37. CASIA Iris Image Database collected by the Chinese Academy of Sciences’ Institute of Automation (CASIA). http://biometrics.idealtest.org/ (Accessed on 20/06/11)

38. PSO Tutorial. http://www.swarmintelligence.org/tutorials.php. (Accessed on 15/08/2011)
39. Ant Colony Optimization. http://iridia.ulb.ac.be/~mdorigo/ACO/ACO.html. (Accessed on 22/08/11)
40. James Kennedy and Russell Eberhart, "Particle Swarm Optimization", IEEE Int'l. Conf. on Neural Networks (Perth, Australia),IEEE Service Center, Piscataway, NJ, IV:1942-1948. Bureau of Labor Statistics, Purdue School of Engineering and Technology. From http://www.engr.iupui.edu/~shi/Coference/psopap4.html. (Accessed on 20/08/11).
41. Sarosh Islam Khan, "Modular Neural Network Architecture for Detection of Operational Problems on Urban Arterials", University of California: PHD Disertation, 1996
42. Hyunsoo Choi and Chulhee Lee, "Motion Adaptive Deinterlacing with Modular Neural Networks", IEEE transactions on circuits and systems for video technology, vol. 21, no. 6, June 2011.
43. Takashi Kimoto and Kazuo Asakawa, "Stock Market Prediction System with Modular Neural Networks", Computer-based Systems Laboratory Fujitsu Laboratories Ltd: A Research work.