View metadata, citation and similar papers at core.ac.uk
brought to you by CORE
provided by VCU Scholars Compass

Virginia Commonwealth University
VCU Scholars Compass

Theses and Dissertations | Graduate School

2012

FAST NEURAL NETWORK ALGORITHM FOR SOLVING CLASSIFICATION TASKS

Noor Albarakati
Virginia Commonwealth University

Follow this and additional works at: http://scholarscompass.vcu.edu/etd

© The Author

Downloaded from
http://scholarscompass.vcu.edu/etd/2740

This Thesis is brought to you for free and open access by the Graduate School at VCU Scholars Compass. It has been accepted for inclusion in Theses and Dissertations by an authorized administrator of VCU Scholars Compass. For more information, please contact libcompass@vcu.edu.

© Noor M. Albarakati 2012
All Rights Reserved

FAST NEURAL NETWORK ALGORITHM FOR SOLVING CLASSIFICATION TASKS

A thesis submitted in partial fulfillment of the requirements for the degree of Master of Science at Virginia Commonwealth University.

By

Noor Mubarak Albarakati
Bachelor of Science, King Abdul-Aziz University, Saudi Arabia, 2005

Director: Dr. Vojislav Kecman
Associate Professor, Department of Computer Science

Committee Members
Dr. Vojislav Kecman
Dr. Kayvan Najarian
Dr. Rosalyn Hobson

Virginia Commonwealth University
Richmond, Virginia
April, 2012

Dedication

This thesis is dedicated first to my parents: my dear father “May you rest in peace dear father” and my dear mother. Thank you my mother for your endless love, unconditional support and encouragement.

I owe my deepest gratitude to my dear siblings for their love, affection and moral support, and especially my sister Nahla and my brother Noseir, who have been kind, taking care and very patient with me in those tough times I went through, encouraging and creatively boosting my energy to its maximum.

I would like also to thank my best friend Mahsa Zahary, who was always willing to raise my morale and give me best suggestions. I never felt lonely working in my lab when she was around. “You will be fine” her favorite sentence to calm my stress down. Best thankful is going to Reyhaneh Mogharabnia, who was always making splendid short visits to my lab.

Acknowledgment

This thesis would not have been possible to exist without having a full supervising, encouragement, guidance and supporting from the initial step of how to do a research until the final step of documenting it into a thesis format unless I have all of these from Dr. Vojislav Kecman, associate professor in Computer Science department. He really helped me to fully understand the thesis subject, directed me during all my experiment, and taught me the research methodologies and how to write them down academically. I am really glad to have the opportunity to work under Dr. Vojislav Kecman supervision.

I am so thankful for all those people who helped me while I was working in this thesis. At first, I would like to show my gratitude to Robert Strack, who was always readily agreed to help me whenever I faced a problem. I am also very gratitude to the entirely patient Michael Paul Pfaffenberger for grammatically correcting my thesis.

I would like also to acknowledge my job back home, Yanbu University College, and Saudi Arabian Cultural Mission, for their academic and financial support.

Table of Contents

List of Tables ………………………………………………………………………...………… vii

List of Figures ……………………………………………………………………..…..….. viii

Abbreviations ……………………………………………………………………………… ix

Abstract ……………………………………………………………………………..…..... x

1 AN INTRODUCTION TO NEURAL NETWORKS 1

1.1 Introduction 1

1.2 Artificial Neural Network 2

1.3 Architectures of Neural Network 2

1.4 Learning Methods 3

1.4.1 Supervised Learning 3

1.4.2 Unsupervised Learning 4

1.4.3 Reinforcement Learning 4

1.5 Applications of Supervised Learning in NN 4

1.6 Perceptron 5

1.6.1 Perceptron Learning Algorithm 6

1.7 Multilayer Perceptron (MLP) 8

1.8 Activation Functions 9

1.8.1 Threshold Activation Functions 9

1.8.2 Linear Activation Functions 10

1.8.3 Nonlinear Activation Functions 10

1.8.3.1 Unipolar Logistic (Sigmoidal) Function 10

1.8.3.2 Bipolar Sigmoidal Function (Hyperbolic Tangent Function) 10

1.9 Learning and Generalization 10

1.9.1 Over-Fitting and Under-Fitting Phenomena 11

1.9.2 Bias and Variance Dilemma 11

1.9.3 Controlling Generalization Errors 12

1.9.3.1 Cross Validation 12

1.10 Problem Statement and Previous Work 14
2 OVERVIEW OF THE EXPERIMENT 15
2.1 Introduction 15
2.2 Experimental Overview 15
2.3 Contents of Experimental Chapters 17
3 EXPERIMENTAL PROCEDURES OF DEVELOPING FAST NN ALGORITHM 18
3.1 Least Mean Squares Algorithm (LMS) 18
3.2 Adapting Learning Rate and the Momentum Term 22
3.3 Error Back-Propagation Algorithm (EBP) 23
3.4 Fast Neural Network Algorithm 28
3.4.1 Batch Learning Technique 28
3.4.2 Batch EBP Algorithm 29
3.4.3 Summary of the Fast Neural Network Algorithm 32
3.4.4 Issues to be Considered 34
3.4.4.1 Labeling Desired Output 34
3.4.4.2 Initializing Weights 35
3.4.4.3 Using a Single Neuron in the OL of One Model/*K* OL Neurons Structure for Two-Class Data Sets 35
3.5 Experimental Data Sets 36
3.5.1 General Information 36
3.5.2 Preprocessing 37
3.5.2.1 Scaling Raw Data 37
3.5.2.2 Shuffling the Scaled Data Set 37
4 EXPERIMENTAL NEURAL NETWORK STRUCTURES 38
4.1 The Differences between Neural Network Structures 38
4.1.1 One Model/*K* Output Layer Neurons Structure 39
4.1.2 *K* Separate Models/One Output Layer Neuron Structure 39
4.1.3 *K* Joint Models/One Output Layer Neuron 40
4.2 Simulated Example 40
4.2.1 One Model/*K* OL Neurons 40

4.2.2 *K* Separate Models/One OL Neuron ... 41
4.2.3 *K* Joint Models/One OL Neuron ... 42
5 EXPERIMENTAL RESULTS AND DISCUSSION ... 45
5.1 Controlling the Experimental Environment ... 45
5.2 Comparison of Three Different MLP Structures ... 47
5.2.1 Comparison of Three Different MLP Structures in Term of Accuracy ... 47
5.2.2 Comparison of Three Different MLP Structures in Terms of Structure Size ... 51
5.2.3 Comparison of Three Different MLP Structures in Term of Time Consumption ... 53
5.3 Using a Neuron in the OL of One Model/*K* OL Neurons Structure for Two-Class Data Sets ... 55
6 CONCLUSIONS ... 56
6.1 The Conclusion ... 56
6.2 Future Works ... 57

List of Tables

3.1 Experimental Data Set Information ………………………………………………. 36

5.1 Experimental Fixed Parameters …………………………………………………... 45

5.2 Experimental Variable Parameters ……………………………………………….. 46

5.3 Accuracy of Three MLP Structures ………………………………………………. 48

5.4 Number of HL Neurons of Three MLP Structures ……………………………….. 52

5.5 The Accuracy of Using One or Two OL Neurons in Vote Data Set ………………... 55

List of Figures

1.1 A Single Perceptron 5

1.2 Multilayer Perceptron of One Hidden Layer and One Output Layer 9

1.3 The Trade-off between Bias and Variance 12

1.4 Cross Validation Procedure 13

3.1 A HL Neuron *J* has a Connection with an OL Neuron *K* in Details 25

4.1 *One Model/K OL Neurons* Structure 41

4.2 *K Separate Models/One OL Neuron* Structure 43

4.3 *K Joint Models/One OL Neuron* Structure 44

5.1 The Scores of Ranking Three Different MLP Structures 49

5.2 The Accuracy of Different MLP Structures of Eleven Data Sets 49

5.3 The Accuracy of Different MLP Structures of Eleven Data Sets 50

5.4 The Structure Size of Three MLP Structures for Eleven Data Sets 53

5.5 Training time of Different MLP Structures of Eleven Data Sets........ 54

Abbreviations

ANN	Artificial Neural Network
ART	Adaptive Resonance Theory
EBP	Error Back Propagation
FNN	Feedforward Neural Network
HL	Hidden Layer
IL	Input Layer
LMS	Least Mean Square
MLP	Multilayer Perceptrons
MSE	Mean Square Error
NN	Neural Network
OL	Output Layer
OvA	One-versus-All
RBFN	Radial Basis Function Network
RNN	Recurrent Neural Network
SLP	Single Layer Perceptron
SOM	Self-Organizing Map
SVM	Support Vector Machine

Abstract

FAST NEURAL NETWORK ALGORITHM FOR SOLVING CLASSIFICATION TASKS

By Noor M. Albarakati, BS.

A thesis submitted in partial fulfillment of the requirements for the degree of Master of Science at Virginia Commonwealth University.

Virginia Commonwealth University, 2012.

Major Director: Dr. Vojislav Kecman,
Associate Professor, Department of Computer Science

Classification is one-out-of several applications in the neural network (NN) world. Multilayer perceptron (MLP) is the common neural network architecture which is used for classification tasks. It is famous for its error back propagation (EBP) algorithm, which opened the new way for solving classification problems given a set of empirical data. In the thesis, we performed experiments by using three different NN structures in order to find the best MLP neural network structure for performing the nonlinear classification of multiclass data sets. A developed learning algorithm used here is the batch EBP algorithm which uses all the data as a single batch while updating the NN weights. The batch EBP speeds up training significantly and this is also why the title of the thesis is dubbed 'fast NN …'. In the batch EBP, and when in the output layer a linear neurons are used, one implements the pseudo-inverse algorithm to calculate the output layer weights. In this way one always finds the local minimum of a cost function for a given hidden layer weights. Three different MLP neural network structures have been investigated while solving classification problems having *K* classes: o*ne model/K output layer neurons*, *K separate models/One output layer neuron*, and *K joint models/One output layer neuron.* The extensive series of experiments performed within the thesis proved that the best structure for solving multiclass classification problems is a *K joint models/One output layer neuron* structure.

1 An Introduction to Neural Networks

1.1 Introduction

Machine learning is a significant part of almost all research and developments today. Gaining knowledge from empirical data is the core of machine learning. The knowledge is achieved by changing either a structure or parameters of a model or both in order to improve its expected performance on future data [3]. These changes have been performed to accomplish one of artificial intelligence tasks which can be learning, decision making, prediction, recognition, diagnosis, planning, control, …, etc. Recently, different approaches are used to learn from data such as support vector machine (SVM), decision tree, clustering, Bayesian networks, genetic programming, and artificial neural network. This thesis will discuss learning from experimental data by using artificial neural network. In particular, it will develop a fast neural network algorithm and it will test several neural network structures in order to find what the best approach for multiclass classification problems is.

1.2 Artificial Neural Network

Artificial neural network (ANN), or often it called *neural network*, is a parallel computational model that takes its structure and function from biological neural networks. A neuron is the main artificial node in the NN. It processes the summation of inputs by using activation function to generate an output. An activation function could be linear or nonlinear. All neurons are connected peer-to-peer to each other by weights w_i. The output of a nonlinear neuron is given by

$$o = f(u) = f(\sum_{i=1}^{n} w_i \, x_i + \, b) \, = f(\boldsymbol{w}^T \boldsymbol{x} \, + b) \qquad (1.1)$$

where, u is an input to the neuron and o is its output, $f(u)$ is an known dependency, mapping or function, between input and output, x_i is the ith input, w_i is the *i-th* weight, n is the total number of inputs, and b is a threshold or a *bias*.

1.3 Architectures of Neural Network

Neural network can basically be divided into feedforward neural network, and recurrent neural network.

Feedforward neural network (FNN) architecture consists of a finite number of layers which contain a finite number of neurons in a feedforward manner. There is neither no feedback connection in the whole network, nor a connection between neurons in a single layer. The layers are connected by network weights. Number of neurons in a single layer has to be sufficient to solve the problem, and number of layers has to be minimal as much as possible to reduce the problem solving time. FNN are classified into *fully connected layered FNN* or *partially connected layered FNN*. When each neuron connects to every feedforward neurons in the

network, it is considered as a fully connected layered FNN. Otherwise, FNN will be considered to be a partial one. Multilayer Perceptrons (MLP) and Radial Basis Function Network (RBFN) are the most fully connected layered FNN could be used in NN.

In *recurrent neural network* (RNN), there is at least one feedback connection, and that make this type of network a dynamic NN. Hopfield model and the Boltzmann machine are the most popular RNN.

1.4 Learning Methods

Neural network has to learn its parameters, such as weights by using training data (learning process) in order to predict, or to estimate, the correct output for any new input (generalization process). Learning methods are mostly classified into supervised, unsupervised and reinforcement learning.

1.4.1 Supervised Learning

Supervised learning is basically about having the data set as pairs of input and desired output ($\mathbf{x}$, d). *Error-correction rule* is a learning technique which is used in supervised learning algorithms to do a direct comparison between desired output d and actual network output o for a given input $\mathbf{x}$ in order to minimize the errors values between them ($e = d - o$). During training phase, network weights have been adjusted by feeding the errors back to the network. Usually, mean square error approach (MSE) is used as a cost function [3]. Two neural network applications that apply supervised learning algorithms are the classification and regression. Solving multiclass classification problems by using MLP neural network which is one of supervised learning algorithms is the central part of this thesis.

1.4.2 Unsupervised Learning

In an unsupervised learning, there is no desired output in training data in which consequently there are no errors counted to direct learning process. Unsupervised learning method relies on finding the relations and correlations among the input features to figure out the hidden structure of unlabeled data. The self-organizing map (SOM) and adaptive resonance theory (ART) are two instances of neural network models that use unsupervised learning algorithms.

1.4.3 Reinforcement Learning

Reinforcement is a type of supervised learning; however, it has less detailed information of the output available. It depends upon evaluative signals from learning environment to direct the learning.

Both the unsupervised learning and the reinforcement one are beyond the scope of this study.

1.5 Applications of Supervised Learning in NN

Neural network has been borne by the end of 1940s, and it has been started to solve complex problems in science and engineering fields by 1980s decade [5]. Thus, different applications appeared in the neural network world such as modeling and identification of systems, pattern recognition, signal processing, optimization, controlling and classification. Most useful applications of neural network that implemented supervised learning methods are classification and function approximation (regression).

Classification is a statistical application which is solely based on assigning discrete input data to a number of discrete classes or categories by approximating the underlying function that classified the data set. Both MLP and RBFN architectures are mostly used in classification tasks. Most algorithms that solve classification problems currently are MLP neural network, support vector machines, k-nearest neighbors, Gaussian mixture model, Gaussian, naive Bayes, decision tree and RBF classifiers.

Function approximation is another statistical application which is based on finding numerical mapping between input and desired output. Regression is one example of function approximation which generates the continuous approximation of the underlying function between input and output.

1.6 Perceptron

A neuron which has a *linear combiner* followed by a *hard limiter* is called a *perceptron*. The perceptron is used to classify the two classes of autonomously input patterns were linearly separable. Figure 1.1 illustrates the perceptron graphically [4].

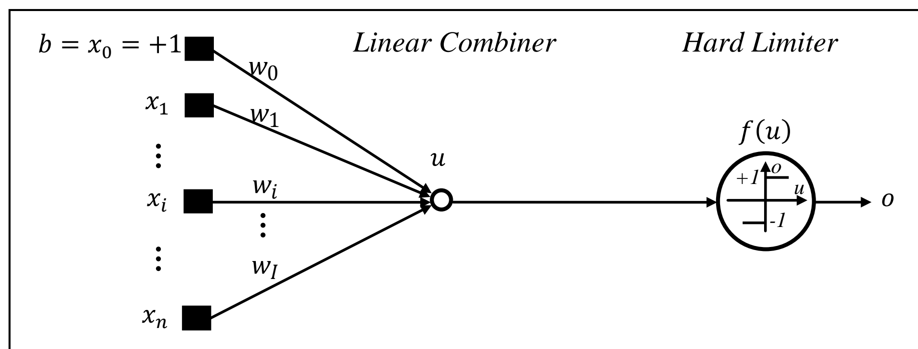

Figure 1.1: A Single Perceptron

The *linear combiner* is an operation of summing n+1 weighted inputs to produces u, which is mathematically represented by equation (1.2)

$$u = \sum_{i=0}^{n} w_i x_i = w_0 x_0 + w_1 x_1 + \dots + w_n x_n = \boldsymbol{w}^T \boldsymbol{x} \tag{1.2}$$

where x_0 is a bias which is fixed to one, and w_0 is its corresponding weight that are used in order to shift the *decision boundary* (or *separation line*) of a classifier away from the origin.

The *Hard limiter* produces an output o which is either +1 if the hard limiter's input u is positive, or -1 if u is negative. Equation 1.3 describes the operation of hard limiter mathematically

$$o = f(u) = sign(u) = sign(\sum_{i=0}^{n} w_i x_i) = \begin{cases} +1 & for\ u \geq 0 \\ -1 & for\ u < 0 \end{cases} \tag{1.3}$$

where *sign*(.) stands for the signum function (known also as the Heaviside function) [5].

1.6.1 Perceptron Learning Algorithm

Perceptron learning algorithm is an iterative algorithm which depends upon the *error-correction rule* to adjust the network weights **w** proportional to the error $e = d - o$ between the desired output d and the actual perceptron output o of a given random chosen data pair (**x**, d), in such a way that the errors will be reduced to zero.

To describe the perceptron model mathematically, let us define the following column vectors

$$\mathbf{x} = [+1,\ x_1,\ x_2,\ \dots,\ x_n]^T \tag{1.4}$$

$$\mathbf{w} = [\,w_0,\ w_1,\ w_2,\ \dots,\ w_n]^T, \tag{1.5}$$

where the input vector **x** has n features or dimensions, and the vector **w** is its corresponding weights. Notice that throughout this entire thesis, both **x** and **w** will be augmented vectors by +1 and w_0 respectively. Thus, by given a set of P training data pairs, assumes that a randomly chosen labeled input pair ($\mathbf{x}_p$, d_p) at time p is applied to the perceptron to classify it into two distinct classes: class-1 or class-2, and the vector of weights $\mathbf{w}_p$ is randomly initiated, and consequently, the linear combiner of the perceptron results the weighted sum of inputs u_p which is defined by equation 1.2. According to figure 1.1, a resultant value u_p of the linear combiner is applied to the hard limiter $f(u_p)$ to classify the input $\mathbf{x}_p$ to either class-1 if the o_p value is equal or greater than zero, or to class-2 if it is less than zero by using a signum function (equation 1.3). Moreover, the *decision boundary* or *separation line*, which is estimated by the classifier to separate two linear separable classes, is defined by a resultant value u_p when it is equal to zero as follows

$$u = \sum_{i=0}^{n} w_i \, x_i = \mathbf{w}^T\mathbf{x} = 0 \tag{1.6}$$

According to *error-correction rule*, the *perceptron learning algorithm* iteratively changes the network weights proportionally to the error $e_p = d_p - o_p$, and a new adaptive weights $\mathbf{w}_{p+1}$, which is the sum of weights $\mathbf{w}_p$ and its correction weights $\Delta\mathbf{w}_p$, will be calculated as following

$$\boldsymbol{w}_{p+1} = \boldsymbol{w}_p + \Delta\boldsymbol{w}_p = \boldsymbol{w}_p + \eta \, e_p \, \boldsymbol{x}_p = \boldsymbol{w}_p + \eta \left(d_p - o_p\right) \boldsymbol{x}_p \tag{1.7}$$

where η is a *learning rate* that controls the learning process by specifying the magnitude of the correction weights $\Delta\mathbf{w}_p$, however, it does not determine the direction of weights changes. After that, a next randomly data pair ($\mathbf{x}_{p+1}$, d_{p+1}) at time p+1 is chosen from training data, and the whole perceptron learning algorithm strategy is repeated by using the new adaptive weights $\mathbf{w}_{p+1}$.

By performing the previous procedure on training data for p = 1, 2, 3, …, P, the adaption of network weights will be stopped when $e_p = 0$ for all data pairs.

A single perceptron is considered as one node (neuron) in NN, and it is used in a single layer perceptron (SLP) network to linearly classify multiclass data sets. However, for nonlinearly multiclass classification problems, multilayer perceptron neural network is used.

1.7 Multilayer Perceptron (MLP)

The architecture of multilayer perceptron consists of fully connected layers of neurons between input and output. Typically, it consists of one or multiple hidden layers and one output layer. Each layer empirically has to apply the same activation functions. Last node in each layer is a threshold or a bias which is fixed to one. As what has already mentioned, weights in the network are used to connect neurons between layers. Figure 1.2 illustrates a MLP model that has an *input layer* (IL), a single *hidden layer* (HL) and an *output layer* (OL). A given training data ($\mathbf{x}$, d), which has input $\mathbf{x}$ of n features, is applied to MLP. Hidden layer that has J neurons is connected to the nodes of input layer by $\mathbf{V}$ weights, however, $\mathbf{W}$ weights is used to connect K neurons of OL with J neurons of HL.

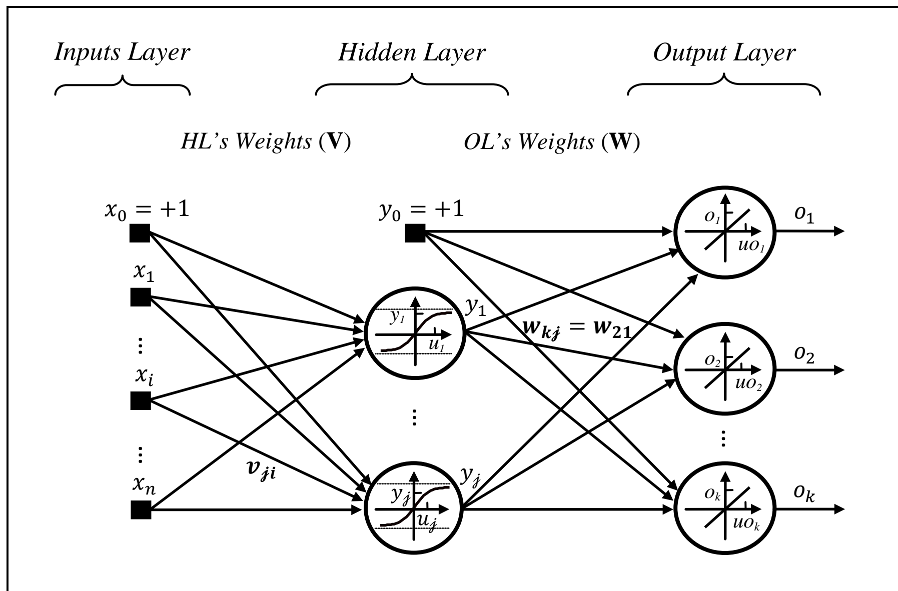

Figure 1.2: Multilayer Perceptron of One Hidden Layer and One Output Layer

1.8 Activation Functions

The most important ability of neural network is transforming the activation level of summing the weighted inputs of a neuron into an output by using an activation function. Usually, it maps the real numbers of inputs into either an interval (-1, +1) or (0, +1). The activation functions used in NN are classified into threshold, linear and nonlinear activation functions. In MLP, hidden layer has nonlinear activation functions. However, output layer has both linear and nonlinear activation functions.

1.8.1 Threshold Activation Functions

Threshold activation function is a hard limited activation function. A signum function is an example of threshold function that always gives -1 or $+1$ output value. Threshold functions are useful for binary classification that classifies the inputs into two groups by using a winner-takes-all approach.

1.8.2 Linear Activation Functions

Linear activation function of a neuron gives an output which is equal to its linear combiner u. By applying the pseudo-inverse algorithm in the OL of MLP, as an instance, the linear activation function is used to give the local minimum of a cost function E for a given HL weights **V**.

1.8.3 Nonlinear Activation Functions

Nonlinear activation functions are used in both HL and OL to iteratively update network weights, and thus solve complex nonlinear problems. The most useful nonlinear activation functions in MLP that has S-shaped are unipolar logistic function and bipolar sigmoidal function.

1.8.3.1 Unipolar Logistic (Sigmoidal) Function

Logistic sigmoidal function is a unipolar function that is applied in a neuron gives an output value y within a range of [0, +1] for any input u. It mathematically produces y as follows

$$y = f(u) = \frac{1}{1+e^{-u}} \tag{1.8}$$

1.8.3.2 Bipolar Sigmoidal Function (Hyperbolic Tangent Function)

Hyperbolic tangent function gives an output value y within the range [-1, +1] for a given input u that is applied to a neuron. Equation (1.9) formulates the hyperbolic tangent function

$$y = f(u) = \frac{2}{1+e^{-u}} - 1 \tag{1.9}$$

1.9 Learning and Generalization

Repeatedly, learning from a given data set is used to identify either the model's parameters of an approximated underlying function, or the model's structure. After a learning process is

completed and a model is obtained, the model has a generalization ability to predict or estimate the accurate output of a new input. Neural networks mainly have two phenomena that affect respectively on learning and/or generalization which are over-fitting and under-fitting.

1.9.1 Over-Fitting and Under-Fitting Phenomena

The most significant two problems affect on learning and generalization of neural networks which are under-fitting, and over-fitting (or over-trained), respectively [5]. Over-fitting phenomenon occurs when the neural network has trained the noisy or imprecise data during the learning phase. Thus, the model could achieve 100% accuracy for classify a given data set; however, it will not have a strong generalization ability of new input data. Empirically, data sets usually have a certain level of noise. Therefore, when the neural network has to learn from training data, it should stop learning in such criteria that the generalization ability is superior. On the other hand, under-fitting problem is about being far away from the actual underlying function of a given data set. Generally, those two problems identify the bias and variance dilemma.

1.9.2 Bias and Variance Dilemma

Overall generalization errors come from two terms: bias and variance. Bias occurs when the network tries to fit all data points including the noises. In contrary, variance problem addresses the smoothness of an approximated model in comparison with the actual underlying function that generated the training data. The over-fitting problem in training data has occurred when the model has small bias and large variance. Under-fitting phenomenon has been caused by a model which has a large bias and a small variance. Always there is a trade-off between bias and variance. Figure 1.3 shows the optimal area of the trade-off between bias and variance [5].

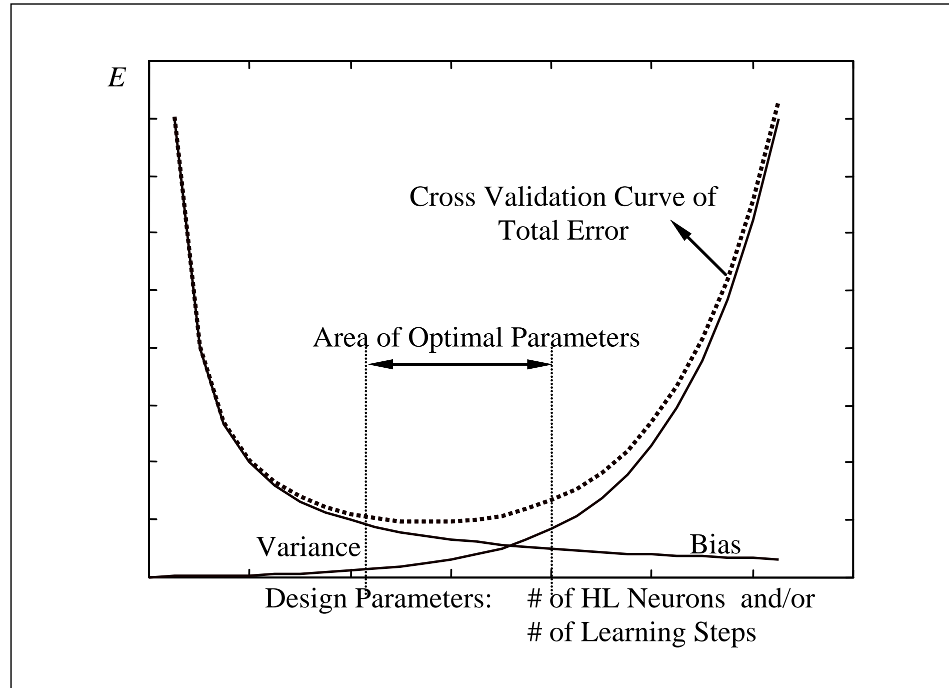

Figure 1.3: The Trade-off between Bias and Variance

1.9.3 Controlling Generalization Errors

The optimal trade-off area between bias and variance on a model (figure 1.3) that reduces the generalization errors will give an effective and an efficient model for learning and generalization processes on a given data set. The most useful method that controls the generalization errors is a cross validation approach which is used to estimate how accurately the model performs in unseen data.

1.9.3.1 Cross Validation

Cross validation is a stopping criterion that controls learning process to implement a model that has good generalization process. It is a statistical method that divides the overall data set randomly into two sets: training set and testing (or validation) set. The majority of the data goes to the training set. *K*-fold cross validation technique is the most popular technique has been used

which splits the data into *k*-folds in such a way that (*k*-1) folds are used in training to build a model, and the last fold left is held-out for testing or validation. For a particular iteration in *k*-fold cross validation method, each data point should be existed once in either training or testing sets. To apply *k*-fold cross validation, all data set must initially be shuffled and all classes must be presented in the training set. Moreover, after training phase is completed, the obtained model uses test set to see how the approximated model behaves on unseen data (validation). As a result, cross validation technique helps the approximated model to give good generalization ability on future data. Recently, 10-fold cross validation is the popular form used in data mining and machine learning fields [4]. Figure (1.4) graphically demonstrates an example of 3-fold cross validation technique.

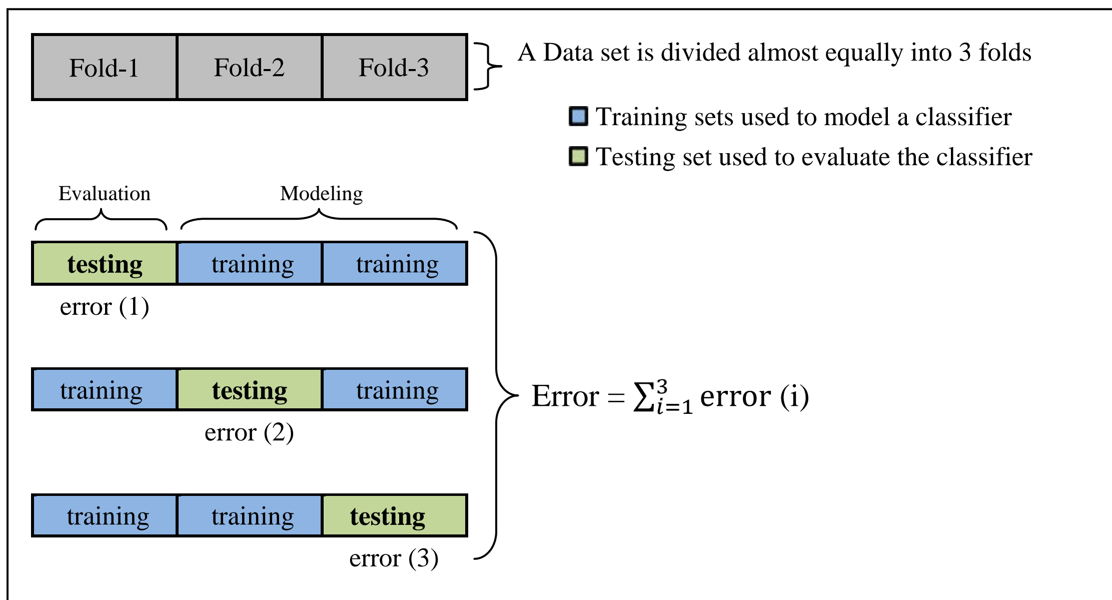

Figure 1.4: Cross Validation Procedure

Cross validation technique has been used during implementing the model to achieve one of the following two reasons:

- Enhancing the generalization ability of a model to accurately predict the future data.
- Comparing the performance of two or more learning algorithms by using a double cross validation approach to find out the best one for a given data set.

1.10 Problem Statement and Previous Work

The goal of this research is to identify the best MLP model for multiclass classification tasks by constructing different structures of MLP, applying a fast error back propagation (EBP) algorithm on all structures, and then comparing their performances in term of accuracy and time consumption.

Previous work was about designing and implementing an EBP learning algorithm for solving multiclass classification problems by using MLP neural network. Here, the fast EBP based on a batch version of the EBP algorithm is designed and used to learn iteratively the weights of hidden and output layers. Three activation functions are implemented in both hidden and output layers. A significant speed up can be achieved when OL neurons are linear by using pseudo-inverse algorithm for calculation of the output layers weights w_{kj} [5]. The EBP implemented can also use the momentum term in updating the weights which usually speeds up the learning.

2 Overview of the Experiment

2.1 Introduction

The ultimate goal of this study is to find which structure of MLP NN is the best for performing the nonlinear classification of multiclass datasets. In order to perform this task, a fast EBP algorithm is developed and tested on eleven data sets in terms of accuracy and CPU time needed during the training i.e., learning phase. The basic description of the EBP algorithm is given in chapter 3, while the three different structures for solving multiclass problems are presented in chapter 4.

2.2 Experimental Overview

In the first phase of the experiment, the fast EBP algorithm is developed and implemented for training nonlinearly separable data sets in MLP models. During the learning, 10-fold cross validation was applied and the scaled and shuffled data have been used in order to enhance the ability of generalization on future, previously unseen, data. Activation function of hidden layer

can be either logistic sigmoidal function or a hyperbolic tangent. Here, the latter was used. In the output layer linear activation function was used, which enables the use of the pseudo-inverse for calculation of the OL weights.

Two types of parameters were used in the algorithm: fixed parameters and variable parameters. *Fixed parameters* are constant during all the experiment. They contain values of momentum term, the range of randomly initiating HL weights, and number of cross validation folds used in training phase. They also determine the learning approach of OL weights to be either direct estimation by using pseudo-inverse method, or iterative adjustment by using EBP algorithm. On the other hand, *variable parameters*, which are the core of the first phase of this study, are the number of neurons in a hidden layer (J), learning rate (η) and number of training iterations (*iterations*). The 10-fold cross validation over the variable parameters gives us their best values for a given data set.

The fast neural network algorithm, which is developed in the first phase, was used in the second phase of experiment by applying it for three different MLP structures: *One Model/K output layer neurons*, *K separate models/One output layer neuron*, and *K joint models/One output layer neuron*. All three different structures that applied fast neural network algorithm for solving nonlinear multiclass classification problems are described in more detail in chapter 4.

The experimental training time and accuracies were computed as well as the structure size for all three MLP structures, and the results of MLP structures that applied fast neural network algorithm for multiclass classification task are deeply discussed in experimental results and discussion chapter.

2.3 Contents of Experimental Chapters

Experimental chapters are organized as follows: chapter 3 describes in more detail the first experimental phase which is a developing the fast neural network algorithm as well as the experimental data sets. Second part of the experiment will be explained in chapter 4 by describing all MLP structures that are used in the experiment, and then stating experiment simulation examples of three MLP structures. Finally, the experimental results and discussions are deeply clarified in chapter 5.

3 Experimental Procedures of Developing Fast NN Algorithm

Since MLP is the most popular neural network structure for classification tasks [5], the developed fast neural network algorithm is based on error back-propagation (EBP) which includes the least mean square (LMS) algorithm as a linear adaptive-filtering algorithm. The main goal of the first phase of this study is to find the best variable parameters of a MLP model that applies the fast neural network algorithm to classify a given multiclass data set. The following sections have full descriptions of all experimental algorithms are applied.

3.1 Least Mean Squares Algorithm (LMS)

Least mean square algorithm (known also as *delta learning rule*), is an adaptive learning algorithm which iteratively adapts the network's weights by minimizing the cost function E rather than computing the misclassified patterns[1]. Precisely, it is an adaptive linear gradient-descent algorithm that is used to successively adjust weights $\Delta\mathbf{w}$ by taking step size η proportionally to a

[1] The presentation of EBP follows [4] and [5].

direction of steepest descent of *E*, which is the opposite direction of a gradient vector $\nabla E(\mathbf{w})$ of the cost function *E*. For that reason, gradient-descent optimization method is applied to steeply converge to a local minimum of the cost function surface. By using the step size η (*learning rate*) in the LMS algorithm, it is noticeable when η is small, the smooth and accurate LMS performance is achieved. However, the rate of convergence to the local minimum is slow. Equation (3.1) defines the adaption of network weights **w** by using LMS learning rule when the pair *p* is given to the network.

$$\boldsymbol{w}_{p+1} = \boldsymbol{w}_p + \Delta\boldsymbol{w}_p = \boldsymbol{w}_p - \eta \; \nabla E(\boldsymbol{w}_p) = \boldsymbol{w}_p - \eta \left.\frac{\partial E}{\partial \boldsymbol{w}}\right|_p \tag{3.1}$$

Learning the model parameter (weights) by LMS algorithm is in an on-line mode in which the network weights are updated after each pair (pattern) of the training data that has been trained. Thus, it is pattern- based and not epoch-based, which the latter is dependent on the entire epoch (all patterns of training data) to be processed before doing any update. So for brevity in the following sections, the subscript $_p$ will be skipped.

During a one epoch, a training data pair (**x**, *d*) is taken randomly from the training data and applied to a neuron that has an activation function *f*(*u*). Fortunately, the weighted inputs *u* of the neuron which is given by equation 1.2 can be written by using the matrix form as follows

$$u = \boldsymbol{w}^T \boldsymbol{x} \tag{3.2}$$

By applying the activation function on *u*, the output of the neuron is given by

$$o = f(u) = f(\boldsymbol{w}^T \boldsymbol{x}) \tag{3.3}$$

To determine the error magnitude of weights **w** for a particular pattern (**x**, *d*), a direct comparison between the desired output *d* and the neuron output *o* will be computed

$$e = d - o = d - (\boldsymbol{w}^T\,\boldsymbol{x}) \tag{3.4}$$

This error will be used to control the adaption of weights of the neuron in such a sense of minimizing the cost function *E* of overall network weights. The sum of error squares is taken as a cost function which has to be gradually minimized during the training phase, thus the derivation of the learning procedure $\nabla E(\boldsymbol{w})$ has only to be made through deterministic argument [5]. In addition, by using the sum of error squares, the continuously nonlinear differentiable cost function *E* of weights vector **w** will geometrically be a quadratic hyper-surface [4]. Equation (3.5) defines the cost function *E*

$$E = \frac{1}{2}\,e^2 = E(\boldsymbol{w}) \tag{3.5}$$

The differentiation of cost function *E* with respect to vector **w** gives the gradient vector $\nabla E(\boldsymbol{w})$. Thus, the chain rule of the differentiation is given by

$$\nabla E(\boldsymbol{w}) = \frac{\partial E}{\partial u}\,\frac{\partial u}{\partial \boldsymbol{w}} \tag{3.6}$$

where the term $\frac{\partial E}{\partial u}$ is called the *error signal δ*, which measures how much the error is changing in response to the change of the inputs of neuron *u*, and the term $\frac{\partial u}{\partial \boldsymbol{w}}$ measures the influence of the vector weights **w** when that particular input *u* is calculated. By applying the chain rule again on equation (3.6), we get

$$\nabla E(\boldsymbol{w}) = \frac{\partial E}{\partial e}\,\frac{\partial e}{\partial o}\,\frac{\partial o}{\partial u}\,\frac{\partial u}{\partial \boldsymbol{w}} \tag{3.7}$$

We get $(\frac{\partial E}{\partial e})$ by differentiating both sides of equation (3.5) with respect to *e*

$$\frac{\partial E}{\partial e} = e \tag{3.8}$$

In addition, $(\frac{\partial e}{\partial o})$ can be found by differentiating both sides of equation (3.4) with respect to *o* as follows

$$\frac{\partial e}{\partial o} = -1 \tag{3.9}$$

The term $(\frac{\partial o}{\partial u})$ can be found by differentiating both sides of equation (3.3) with respect to *u*

$$\frac{\partial o}{\partial u} = f'(u) \tag{3.10}$$

Lastly, $(\frac{\partial u}{\partial \boldsymbol{w}})$ is gained by differentiating both sides of equation (3.2) with respect to **w**

$$\frac{\partial u}{\partial \boldsymbol{w}} = \boldsymbol{x} \tag{3.11}$$

Therefore, by replacing the equivalent terms of equation (3.7) each by equations (3.8), (3.9), (3.10) and (3.11), the first partial derivative of cost function *E* with respect to weights vector **w** is

$$\nabla E(\boldsymbol{w}) = -\, e\, f'(u)\, \boldsymbol{x} \tag{3.12}$$

As a result, LMS or the delta learning rule when the pattern *p* is presented in network can be written as

$$\boldsymbol{w}_{p+1} = \boldsymbol{w}_p + \Delta\boldsymbol{w}_p = \boldsymbol{w}_p - \eta\, \nabla E(\boldsymbol{w}_p) = \boldsymbol{w}_p + \eta\, e_p\, f'(u_p)\, \boldsymbol{x}_p \tag{3.13}$$

Since LMS is applied in EBP algorithm, which is the most popular algorithm for multiclass classification problems, it is better to specify the *error signal* term δ for the OL in the formula of LMS algorithm, where the error signal is given by

$$\delta = e\, f'(u) = (d - o)\, f'(u) \tag{3.14}$$

Therefore, equation (3.13) can be rewritten by using error signal term δ as follow

$$\boldsymbol{w}_{p+1} = \boldsymbol{w}_p + \Delta \boldsymbol{w}_p = \boldsymbol{w}_p + \eta\, \delta_p\, \boldsymbol{x}_p \tag{3.15}$$

Recall that LMS is in on-line mode, therefore for the pattern p, the equation (3.13) can be written in terms of each vector component $w_{j,p}$ of the weights vector $\mathbf{w}_p$

$$w_{j,p+1} = w_{j,p} + \eta\left(d_p - o_p\right) f'\left(u_p\right) x_{j,p} = w_{j,p} + \eta\, \delta_p\, x_{j,p} \tag{3.16}$$

For the linear activation function which can be applied in OL neurons of MLP models, the derivation of activation function is equal to one $f'(u) = 1$, consequently, the error signal δ is equal to the individual error e as follows

$$\delta = e\ f'(u) = e = d - o \tag{3.17}$$

Therefore, LMS learning algorithm for linear neuron is given by

$$\boldsymbol{w}_{p+1} = \boldsymbol{w}_p + \Delta \boldsymbol{w}_p = \boldsymbol{w}_p + \eta\, \left(d_p - o_p\right) \boldsymbol{x}_p = \boldsymbol{w}_p + \eta\, e_p\, \boldsymbol{x}_p \tag{3.18}$$

3.2 Adapting Learning Rate and the Momentum Term

Unfortunately, choosing an optimal learning rate on EBP of MLP is dependent upon trial-and-error technique. It is affected in addition by the number of learning steps. In other words, having a small learning rate could smooth the convergence but it needs large number of iteration

steps during learning process. However, the large learning rate could escape the local minimum of cost function. Table 5.2 lists the values of learning rate η that are used in the experiment.

To speed up the convergence to the minimum of the cost function in EBP algorithm, the *momentum* term is used [5]. It reaches the minimal by using small number of iterations during learning process. The following equation gives the adaptive weights for a pattern p in terms of using the gradient vector $\nabla E(\boldsymbol{w}_p)$ of the cost function E, and the momentum term η_m.

$$\boldsymbol{w}_{p+1} = \boldsymbol{w}_p + \Delta\boldsymbol{w}_p = \boldsymbol{w}_p - \eta\, \nabla E(\boldsymbol{w}_p) + \eta_m[\boldsymbol{w}_p - \boldsymbol{w}_{p-1}] \tag{3.19}$$

where $[\mathbf{w}_p - \mathbf{w}_{p-1}] = \Delta\mathbf{w}_{p-1}$.

Therefore, the adaption of weights $\boldsymbol{w}_{p+1}$ by using the momentum term is given by

$$\boldsymbol{w}_{p+1} = \boldsymbol{w}_p + \Delta\boldsymbol{w}_p = \boldsymbol{w}_p - \eta\, \nabla E(\boldsymbol{w}_p) + \eta_m[-\eta\ \nabla E(\boldsymbol{w}_{p-1})] \tag{3. 20}$$

3.3 Error Back-Propagation Algorithm (EBP)

Error back-propagation algorithm is an adaptive learning algorithm which applies the LMS algorithm to learn network weights. It is basically defined as such: after training a given data set and the network weights are gained, the calculated errors are propagated backward into the network to adapt its weights. In order to apply EBP algorithm on MLP, an activation function especially in HL should be differentiable. Therefore, the common differentiable activation functions in use with MLP are hyperbolic tangent function and logistic sigmoidal function [5]. EBP learning procedure divides into two phases: first phase is forward-pass computations through the MLP network, i.e. from the left side of the network to the right side, and secondly is the backward-pass computation phase which is in the opposite direction of the first phase.

Training process by MLP over a given data set is the main task of the first phase. For an on-line mode, a random training pattern (**x**, d) propagates through the MLP, layer-by-layer in a neuron-by-neuron basis, until it reaches the end of network with a resultant output o. Every neuron in the MLP has two computational roles: first role is about applying an activation function $f(u)$ on the weighted inputs u in order to produce the neuron's output o, which is described in detail in the LMS algorithm (section 3.1). The second computational role of a neuron, which is necessary for backward-pass phase, is to estimate the *error signal* term δ which is a gradient vector of the cost function E with respects to weights vector **w** which effectively weighted the values of the inputs u in a particular neuron. The error signal term δ is a significant term in the back propagation formula. Thus, calculating the error signal δ is dependent on the state of a neuron in HL or OL, as follows:

1. If the neuron is in OL, the equation (3.14) is used to calculate the error signal δ, which is equal to the product of the associated derivative $f'(u)$ of a particular neuron and its corresponding associated error e.

2. If the neuron is in HL, the calculation of error signal term δ is equal to the product of the associated derivative $f'(u)$ of a particular neuron, and the sum of weighted error signals δs of all successive neurons that are connected to that neuron (equation 3.31).

From the LMS algorithm we have a full explanation of how the error signal δ in the OL is calculated. In this section we will merely explain the calculation of the error signal δ in the HL.

Recall that by applying EBP algorithm in on-line mode, the second phase of the algorithm has focused exclusively on adapting the network weights $\boldsymbol{w}_{p+1} = \boldsymbol{w}_p + \Delta\boldsymbol{w}_p$ of every pattern p in the training data by finding all its correction weight component Δw_{kj} of a particular weight w_{kj} that

connects a node j with a node k in the MLP network. Delta rule is used to find the correction weight Δw_{kj} as follows

$$correction\ weight = (learning\ rate) \times (error\ signal) \times (input\ of\ the\ neuron)$$ [4]

$$\Delta w_{kj} = \eta\ \delta_k\ y_j \tag{3.21}$$

Equation (3.15) is a correction weights vector of a neuron which is located in the OL. It is noticeable in the LMS algorithm that the inputs vector of a neuron is denoted by **x**, however, in EBP algorithm which is applied in MLP neural network, for the OL neurons we will represent the inputs vector by **y**. Otherwise, all equations symbols remain the same (see equation 3.33). The HL is different than the OL in computing its errors. Since in an OL neuron, the desired output d of input **x** accurately measures the errors by equation (3.4); however, in a HL neuron, all succeeding neurons have the common responsibilities to calculate the error of the HL neuron. A HL neuron j is depicted in figure (3.1) which shows a left-side connection from node (or neuron) i to neuron j by weight v_{ji}, and a right-side connection from HL neuron j to an OL neuron k by weight w_{kj} [4].

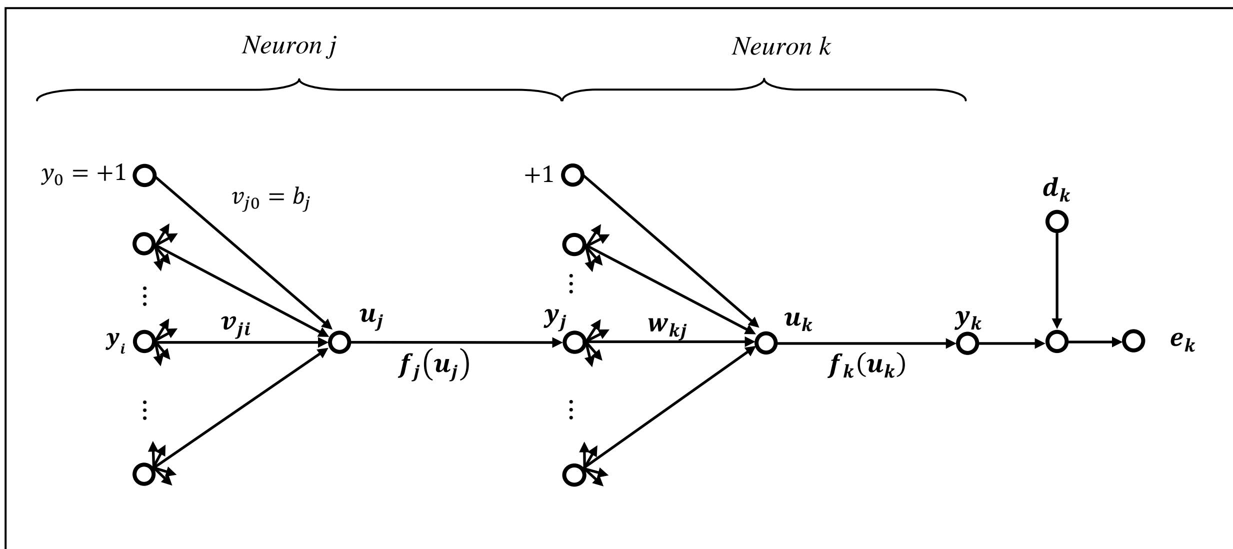

Figure 3.1: A HL Neuron j has a Connection with an OL Neuron k in Details

Therefore, to understand the calculation of error signal δ in the HL by using EBP, let us consider a given pattern ($\mathbf{x}$, d), where $\mathbf{x}$ is an input vector and d is its desired output, and the pattern is propagated through a fully connected MLP network, layer by layer in a neuron-by-neuron fashion until it reaches the HL neuron j which has a differentiable activation function $f_j(u)$. The error signal δ of the HL neuron j is given by

$$\delta_j = -\frac{\partial E}{\partial y_j}\frac{\partial y_j}{\partial u_j} = -\frac{\partial E}{\partial y_j}\ f'_j(u_j) \quad (3.22)$$

As it is depicted in figure (3.1), a neuron k, which is one of the OL neurons, affects the cost function E by

$$E = \frac{1}{2}\ \sum_k {e_k}^2 \quad (3.23)$$

The partial derivative of $\frac{\partial E}{\partial y_j}$ is defined by the differentiate equation (3.23) with respect to the input y_j of neuron k

$$\frac{\partial E}{\partial y_j} = \sum_k e_k\ \frac{\partial e_k}{\partial y_j} \quad (3.24)$$

By using the calculus chain rules in equation (3.24), we get

$$\frac{\partial E}{\partial y_j} = \sum_k e_k\ \frac{\partial e_k}{\partial u_k}\ \frac{\partial u_k}{\partial y_j} \quad (3.25)$$

Since the error of neuron k in the OL is the difference between a desired output d_k and neuron output y_k as follows

$$e_k = d_k - y_k = d_k - f_k(u_k) \quad (3.26)$$

Thus, the first partial derivative $\frac{\partial e_k}{\partial u_k}$ of the error of neuron k with respect to its weighted inputs is given by

$$\frac{\partial e_k}{\partial u_k} = -f'_k(u_k) \tag{3.27}$$

Whereas the weighted input u_k of neuron k is given by

$$u_k = \sum_{j=0}^{J} w_{kj}\, y_j \tag{3.28}$$

where J is a total number of inputs as well as the bias that applies to the neuron k. By differentiating equation (3.28) with respect to y_j, we get

$$\frac{\partial u_k}{\partial y_j} = w_{kj} \tag{3.29}$$

Thus, by replacing the equivalent terms of (3.27) and (3.29) into the equation (3.25), we get

$$\frac{\partial E}{\partial y_j} = -\sum_k e_k\, f'_k(u_k)\, w_{kj} = -\sum_k \delta_k\, w_{kj} \tag{3.30}$$

where the error signal δ_k for the OL neuron K is defined as equation (3.14) in terms of its associated error and weights. As a result, the error signal δ_j formula of a neuron j which is located in the HL is given by using equation (3.30) in (3.22) as follows

$$\delta_j = f'_j(u_j) \sum_k \delta_k\, w_{kj} \tag{3.31}$$

As a result, EBP algorithm that updates the network weights **V** and **W** are given respectively as

$$v_{ji} = v_{ji} + \Delta v_{ji} = v_{ji} + \eta\delta_j x_i = v_{ji} + \eta f'_j(u_j) x_i \sum_{k=1}^{K} \delta_k w_{kj}, \quad j=1,\ldots,J\text{-}1, \; i=1, \ldots, P. \tag{3.32}$$

$$w_{kj} = w_{kj} + \Delta w_{kj} = w_{kj} + \eta\delta_k y_j = \eta(d_k - o_k) f'_k(u_k) y_j, \quad k=1,\ldots,K, \quad j=1,\ldots,J \tag{3.33}$$

The EBP algorithm which adapts network weights typically is in an on-line mode. Our fast neural network algorithm essentially is about using EBP algorithm in an off-line (or batch) mode, which is explained in section 3.4.2 in detail.

3.4 Fast Neural Network Algorithm

Fast neural network algorithm is an EBP batch learning algorithm which is trained in a MLP network of one HL and one OL. At first, batch learning technique is defined in section 3.4.1. Then, it is followed by describing a batch EBP algorithm. In section 3.4.3, the fast neural network algorithm which implicitly has the batch EBP is given in detail.

3.4.1 Batch Learning Technique

All supervised learning algorithms depend on *error-correction rule* to improve a system's performance. In MLP models, the errors which are gained from a model define the cost function of estimated network weights, which is depicted on the space as a multidimensional error-performance surface by using the network weights as its coordinates. Error- performance surface of the average of overall training instances is the accurate one. Thus, the improvement of performance over time has to successively move down toward a minimum point of the error surface. To achieve this goal, an instantaneous gradient vector of the error surface is estimated. Therefore, estimating the gradient will improve the system in the direction of steepest descent of the error surface.

Any adjustment of the network weights represents a new point in the error surface. In a batch learning procedure, the adjustment of weights is performed after all training data are presented, which is considered as one epoch of training. Therefore, the error is defined as the

average error of total number of instances in a training data set. In other words, the experiment goes through an epoch-by-epoch learning basis to adjust the network weights. Recall that the adaption of network weights is only to achieve the goal of minimizing the cost function by estimating its gradient accurately. Thus, the accurate estimation of the gradient vector of a cost function by using batch learning procedure can rapidly converge to the minimum value of the cost function.

During the experiment of this thesis, batch learning was used by applying a 10-fold cross validation technique over a training data set, in such a way that any adjustment on the next epoch weights was performed within 9-training-folds (or chunks) of an iteration of the cross validation process. A batch EBP algorithm is described below.

3.4.2 Batch EBP Algorithm

The batch version of the EBP algorithm implemented in the thesis is divided into two phases: feedforward phase and back-propagation phase. A training data **X,** which has P patterns for K classes, is given by

$$\mathbf{X} = \{\mathbf{x}_p, \mathbf{d}_p, p=1, \dots, P\}, \tag{3.34}$$

where $\mathbf{x}_p$ is an input vector of a pattern p that has n features (or dimensions)

$$\mathbf{x} = [+1 \quad x_1 \quad x_2 \quad \dots \quad x_n]^T \tag{3.35}$$

and $\mathbf{d}_p$ is a vector of its associated labeled desired output

$$\mathbf{d} = [d_1 \quad d_2 \quad \dots \quad d_K]^T \tag{3.36}$$

Feedforward Phase

For J neurons in the HL, the (P, J) dimensional input matrix $\mathbf{u}$ is calculated

$$\boldsymbol{u} = \boldsymbol{X}\,\boldsymbol{V} \tag{3.37}$$

where **X** is the (*P*, *n*+1) training data matrix, and **V** is a (*n*+1, *J*) dimensional matrix of the HL weights. Two differentiable activation functions are used in the algorithm: hyperbolic tangent function and logistic sigmoidal function. A (*P*, *J*) dimensional matrix **y** is the output of the hyperbolic tangent activation function in HL

$$\mathbf{y} = 2/(1 + \exp(-\mathbf{u})) - 1 \tag{3.38}$$

The derivative needed in equation (3.31) for calculating the error signals of the HL is given by

$$\boldsymbol{y}' = 0.5\left(1 - \boldsymbol{y}^2\right) \tag{3.39}$$

Notice that a vector of zeros is in the last column of the matrix **y'** which is the derivative of the fixed bias term. For a sigmoidal activation function the equivalent equations are

$$\boldsymbol{y} = 1/(1 + exp(-\boldsymbol{u})) \tag{3.40}$$

$$\boldsymbol{y}' = \boldsymbol{y}\,(1 - \boldsymbol{y}) \tag{3.41}$$

The hidden layer is augmented with a bias, and so the matrix $\mathbf{y}_b$ of size (*P*, *J*+1), which its last vector is ones $\boldsymbol{y}_b = [\boldsymbol{y} \quad \mathbf{1}]$, is the inputs matrix to the OL. Thus, if the OL neurons are linear, then the OL weights **W** with (*J*+1, *K*) dimensions is directly calculated by using pseudo-inverse algorithm

$$\boldsymbol{W} = \boldsymbol{y}_b^* \, \boldsymbol{d} \tag{3.42}$$

where $\boldsymbol{y}_b^*$ is a pseudo-inverse of $\boldsymbol{y}_b$. Subsequently, a (*P*, *K*) $\mathbf{u}_o$ matrix of inputs to the OL is calculated by

$$\boldsymbol{u}_o = \boldsymbol{y}_b \boldsymbol{W} \tag{3.43}$$

Since, OL neurons are linear, the output of OL neurons $\mathbf{o} = \mathbf{u}_o$, and a (*P*, *K*) dimensional matrix of ones is its derivative as follows

$$\boldsymbol{o}' = \mathbf{1} \tag{3.44}$$

In addition, when OL neurons are nonlinear, the OL weights **W** must be learned iteratively by using a batch EBP algorithm. Now, a (P, K) dimensional matrix **o** is the output of applying the hyperbolic tangent function in the OL, and a matrix **o'** with size (P, K) is its derivatives

$$\boldsymbol{o} = 2/(1 + exp(-\boldsymbol{u_o})) - 1 \quad (3.45)$$

$$\boldsymbol{o}' = 0.5\ (1 - \boldsymbol{o}^2) \quad (3.46)$$

For sigmoidal activation function, the outputs **o** and its derivatives **o'** are given by

$$\boldsymbol{o} = 1/(1 + exp(-\boldsymbol{u_o})) \quad (3.47)$$

$$\boldsymbol{o}' = \boldsymbol{o}\ (1 - \boldsymbol{o}) \quad (3.48)$$

The network **errors** (a matrix of (P, K) dimensions) is defined as a difference between the labeled desired output **d** and the network output **o**

$$\boldsymbol{errors} = \boldsymbol{d} - \boldsymbol{o} \quad (3.49)$$

Back-propagation Phase

Error signals in equations (3.14) and (3.31) for the OL and HL are calculated as a (P, K) dimensional matrix **deltaO** and a (P, J+1) dimensional matrix **deltaY** respectively

$$\boldsymbol{deltaO} = \boldsymbol{errors}\ \ \boldsymbol{o}' \quad (3.50)$$

Similarly, the delta signal matrix **deltaY** for a hidden layer is calculated as a product of derivative output matrix, weights and **deltaO** matrix. To iteratively update the network weights **V** and **W** by using batch EBP, the delta rule is used to find the correction weights of every weight matrices. The correction weights are defined by equation (3.21), which is the product matrix of the inputs of a layer and its associated error signals multiplied by a scalar of chosen learning rate η. The product matrix of the HL and OL are given by **gradV** and **gradW**, respectively. The product matrix **gradV** which has (n+1, J) dimensions is calculated as

$$\boldsymbol{gradV} = \boldsymbol{X}^{\boldsymbol{T}}\ \boldsymbol{deltaY} \tag{3.51}$$

For the OL, **gradW** is a product matrix of size (*J*+1, *K*)

$$\boldsymbol{gradW} = \boldsymbol{y_b}^{\boldsymbol{T}}\ \boldsymbol{deltaO} \tag{3.52}$$

To speed up the learning process, momentum scalar η_m was used. Thus, to update the weights by using the momentum term (equation 3.20), the previous batch **gradV** and **gradW** are saved respectively into **gradV_old** and **gradW_old** in order to use them during the adaption of network weights. Therefore, the network weights **V** and **W** are adapted by using the delta rule and momentum term as follows

$$\boldsymbol{V} = \boldsymbol{V} + \eta\ \boldsymbol{gradV} + \eta_m\ \eta\ \boldsymbol{gradV_old} \tag{3.53}$$

$$\boldsymbol{W} = \boldsymbol{W} + \eta\ \boldsymbol{gradW} + \eta_m\ \eta\ \boldsymbol{gradW_old} \tag{3.54}$$

where the product of (η **gradV**) is the correction weights of **V**, and (η **gradW**) as well is the correction weights of **W**. Last terms, (η_m η **gradV_old**) and (η_m η **gradW_old**), respectively describe using the momentum terms for updating both **V** and **W** weights.

3.4.3 Summary of the Fast Neural Network Algorithm

The NN code developed within the thesis implements the fast NN algorithm within the *k*-fold cross validation (*k*-fold CV) loops. Here, in designing the best NN we have to find three best variable parameters of the NN - first one being a number of hidden layer neurons, second one is the best learning rate η and the last variable parameter that must be determined within the *k*-fold CV loops is the number of iterations. This is why the code developed has three major outer loops within which there is *k* loops for executing the *k*-fold CV as follows:

Step 1. Define three vectors for the three variable parameters: a vector of number of HL neurons J_0, a vector of learning rates η_0, and a vector of number of learning steps $iterations_0$.

Step 2. For first outer loop, pick J (a number of HL neurons) from the vector J_0

Step 3. Initialize **V** and **W** weights as (n+1, J) dimensional matrix and (J+1, K) dimensional matrix, respectively. Note that a fixed parameter *kw* is used to initialize **V** in the range [-*kw*, +*kw*].

Step 4. Pick the learning rate η from the vector η_0 to perform second nested loop, and then for third inner loop pick the iterations number *iterations* from the vector $iterations_0$

Step 5. Reset the *i_error* to zero, which is a scalar of the model's errors that are calculated after using a particular combination of variable parameters (J, η , and *iterations*)

Step 6. For particular variable parameters, 10-fold cross validation is applied over the scaled and shuffled training data, in such a way that within each iteration of cross validation process the **V** and **W** weights are resets to the initial, and the **gradV** and **gradW** matrices, which are respectively as same size as **V** and **W**, are rest to zeros.

Step 7. For each iteration of the 10-fold cross validation:

a. Apply the batch EBP algorithm by using learning rate η on the training folds and *iterations* (number of times) to estimate **V** and **W** weights.

b. Evaluate the estimated model, **V** and **W**, by training the model over a testing fold to calculate its errors in the *i_error* scalar.

c. Accumulate the calculated errors of 10-fold in the *i_error* scalar.

Step 8. Calculate a percentage of the accumulated errors *i_error* over all training data *P*, and then save the percent errors of a particular variable parameters in a three dimensional array **i_Errors** as follows:

i_Errors (J, η, *iterations*) = 100 * *i_error* / *P*

Step 9. Repeat the three nested loops for all values of J_0, η_0 and $iterations_0$ by going to step 2, and save their percent errors in the array **i_Errors**

Step 10. Find the minimum percent errors of entire **i_Errors** array, and consequently extract its associated indices (variable parameters) that represent the best J, best η, and best *iterations* for classification of a particular data set.

Step 11. Build a classification model by using the best variable parameters (best J, best η, and best *iterations*) on all training data *P* to estimate the weights **V** and **W**.

Step 12. Validate the classifier model on all training data *P* of a particular data set by calculating its percent errors. Accuracy can be calculated by subtracting the best percent error from one.

3.4.4 Issues to be Considered

3.4.4.1 Labeling Desired Output

A multiclass NN classifier classifies a given multiclass data set into *K* classes. Typically, a standard binary classifier using linear or hyperbolic tangent activation functions needs the desired output to be labeled either as (-1, +1). If the sigmoidal activation function is used the labeling is

(0, +1). Thus, the output vector for K = 3 classes labeled as $d_0 = [2 \quad 1 \quad 3 \quad \dots]$ should be relabeled as a matrix of three vectors. Each vector identifies one class as follows:

$$\boldsymbol{d} = \begin{bmatrix} -1 & +1 & -1 & \\ +1 & -1 & -1 & \\ -1 & -1 & +1 & \\ & \dots & & \end{bmatrix}.$$

3.4.4.2 Initializing Weights

Practically, initializing weights by using small absolute random values is sufficient for having a good convergence to the underlying classified function [5]. In the experiment, we used *kw* parameter, which is fixed to (0.1), to identify and initiate at random the small absolute values of the hidden layer weights. Bad initial weights may have an effect on learning by getting stuck at such a local minimum, or by having slow convergence to the optimal weights. Empirically, initializing weights by using small values and then increasing them speeds up the learning process in MLP more than starting with maximum values of weights and then the decreasing. It iteratively adjusts a model to the optimal one by starting with almost flat weights and then reshaping it according to the inputs data and number of iterations that have been used.

3.4.4.3 Using a Single Neuron in the OL of One Model/*K* OL Neurons Structure for Two-Class Data Sets

Our empirical evidence shows that using one neuron in the output layer of a two-class data set gives almost the same accuracy in comparison with using two neurons in the output layer. It takes, however, less time for training the model. Therefore, in our structure of a one model and *K* neurons in the output layer, we use only one OL neuron and not two. Section 5.3 discusses the results of using one OL neuron instead of two.

3.5 Experimental Data Sets

3.5.1 General Information

Eleven real data sets were involved in the experiment, which gathered from two different sources: the UCI machine learning repository and benchmarks of Reinhardt and Hubbard [1][2]. First nine data sets are taken from the UCI, and the last two data sets are the benchmarking data which were constructed by Reinhardt and Hubbard for protein sub-cellular localization. The number of features in both of the Reinhardt and Hubbard data sets typically is 20-dimensional amino acid composition for protein sub-cellular localization classification. Table 3.1 summarizes information about the eleven experimental data sets.

Table 3.1: Experimental Data Set Information

Data set	# Instances	# Features	# Classes
Iris	150	4	3
Glass	214	9	6
Vote	232	16	2*
Wine	178	13	3
Teach	151	5	3
Sonar	208	60	2*
Cancer	198	32	2*
Dermatology	366	33	6
Heart	297	13	5
Prokaryotic	997	20	3
Eukaryotic	2427	20	4

3.5.2 Preprocessing

3.5.2.1 Scaling Raw Data

Scaling the raw data sets is a fundamental task in many technical analyses. It assists the development of neural network in effective and efficient ways. Basically, it is used to remove any outliers by spreading out the distribution of data normally into a zero mean and a unit variance, in such a way that the mean and standard deviation for the inputs data are associated with each particular input.

3.5.2.2 Shuffling the Scaled Data Set

In cross validation technique, the data set has to be shuffled. Hence, the reliability in a model's performance is increased by using a large number of estimations on different (shuffled) training data. *K*-fold cross validation technique establishes only *K* numbers of estimated models, thus, shuffling the data set and then estimating *K* models to come up with overall average accuracy of final estimated model will give a model with good generalization ability for future data.

4 Experimental Neural Network Structures

4.1 The Differences between Neural Network Structures

For a given data set that has *K* classes, three different MLP structures were used in the experiment: o*ne model/K output layer neurons*, *K separate models/One output layer neuron*, and *K joint models/One output layer neuron*. A model in this experiment refers to a fully connected MLP neural network that has an input layer, a hidden layer and an output layer. The difference between three structures is the number of models used in a structure. For o*ne model/K OL neurons* structure, there is one model used that has *K* neurons in its OL. However, for *K separate models/One OL neuron* and *K joint models/One OL neuron*, there are *K* models with one neuron in its OL used in a structure. Moreover, *K* models have another difference in their training approach of 10-fold cross validation over a given data set, in which it could be either training jointly all *K* models (*joint models/one OL neuron*), or training separately the 10-fold cross validation on each model alone (*separate models/one OL neuron*). The three different structures are described below in detail.

4.1.1 One Model/*K* Output Layer Neurons Structure

This structure contains one model that has *K* neurons in its OL. Note that during the training the complete matrix **d** given in section 3.4.4.1 is given as the desired NN output in batch learning. The number of neurons in the HL is subjected to the fast neural network algorithm. Precisely, the best variable parameters, number of HL neurons, learning rate and learning iterations (J, η, *iterations*), characterize the model after 10-fold cross-validation on a given data set. For two-class data sets, empirically using one neuron in the OL ($K = 1$) of a model sufficiently gives almost the same accurate results as using two neurons in the OL ($K = 2$), however, it requires less learning time. The experimental results for two-class training data that used one neuron in OL are presented in chapter 5.

4.1.2 *K* Separate Models/One Output Layer Neuron Structure

K models each having one neuron in its OL, are separately constructed to build a MLP structure. Now, during the training the kth model is given the kth column of the matrix **d** given in section 3.4.41 as the desired output vector. Each model is trained separately by using the training data of a particular class of *K* classes; in such a way that each model received its own characteristics, best variable parameters, after separately training the 10-fold cross validation over the training data of its associated class. Therefore, each model may be characterized by different values of best variable parameters, i.e. different number of HL neurons, different values of learning rates, and different number of iterations during learning phase.

4.1.3 *K* Joint Models/One Output Layer Neuron

This structure is exactly same as the second structure except the learning approach, in which all models are jointly trained across 10-fold cross validation over its own associated class, and thereby they are characterized by the same best variable parameters; i.e. same number of neurons in hidden layer, same learning rates, and same number of iterations.

4.2 Simulated Example

To better understand the three different MLP structures and the differences between them, let us simulate the experiment by assuming a scaled and shuffled training data that has three classes $K = 3$ is provided to each structure. The training data consists of a matrix **X** that has P patterns and its labeled desired output **d**. 10-fold cross validation was applied on training data while using the fast neural network algorithm.

4.2.1 One Model/*K* OL Neurons

One MLP model that has $K = 3$ neurons in OL is depicted by figure (4.1). The trained model, which applied the fast neural network algorithm, estimates the best variable parameters (J, η, *iterations*) that classify the training data. For graphical representation, $J = 2$ is chosen as number of neurons in the HL. By denoting **V** as HL weights and **W** is the OL weights, the training data is propagated through the network layer by layer until it reaches the OL by three output values, o_1, o_2, and o_3. All the dimensions of the matrices involved are given in a presentation of the fast batch algorithm in section 3.4.2. The max operation is performed to those three values to apply the winner-takes-all approach that classifies pattern **x** to a winner class.

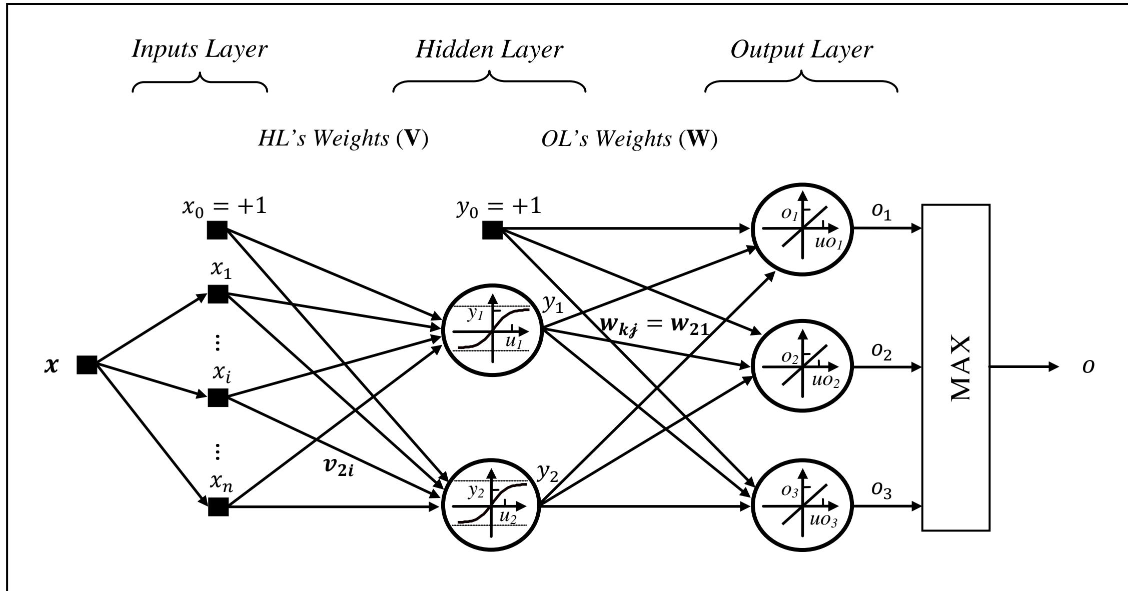

Figure 4.1: *One Model/K OL Neurons* Structure

4.2.2 *K* Separate Models/One OL Neuron

K models each having one neuron in their OL, were trained separately on the training data in such a way that each model was using a matrix of inputs **X** and its associated vector of labeled desired outputs **d**. As it is noticeable in figure (4.2), there are three different models extracted as follows: the first model is associated with the first class of the training data and it estimated two neurons in the HL $J = 2$ as being the best number of neurons to separate the first class data from the other classes. The second model is associated with the second class and it has found that the best number of HL neurons is $J = 4$. Similarly, and this is shown in the figure, the best number of HL neurons for the third model $J = 3$. 10-fold cross validation technique that is applied implicitly in the fast neural network was used separately for each model alone. After training phase is finished, each model produces an output o_1. Therefore, three outputs come out from three models: o_1, o_2, and o_3.Thus, by using winner-takes-all technique, the max operation is used in such a way that the input patterns given in **X** are classified possibly to the correct class.

4.2.3 *K* Joint Models/One OL Neuron

K joint models/One OL neuron has the same model structures as *K separate models/One OL neuron*; however, the learning is changed here. Figure (4.3) depicts the *K joint models/One OL neuron* structure. Each single models is associated with a specific class, but they are jointly (simultaneously) trained on the whole input **X** by using a 10-fold cross validation technique which is implicit in the fast neural network algorithm to produce an output of each model as o_1, o_2, and o_3. Therefore, after the training each NN will have same number of HL neurons (here we have shown 2 HL neuron as being the best (J = 2) for all the models. As described before, a winner-takes-all technique is used for classification by using max operator.

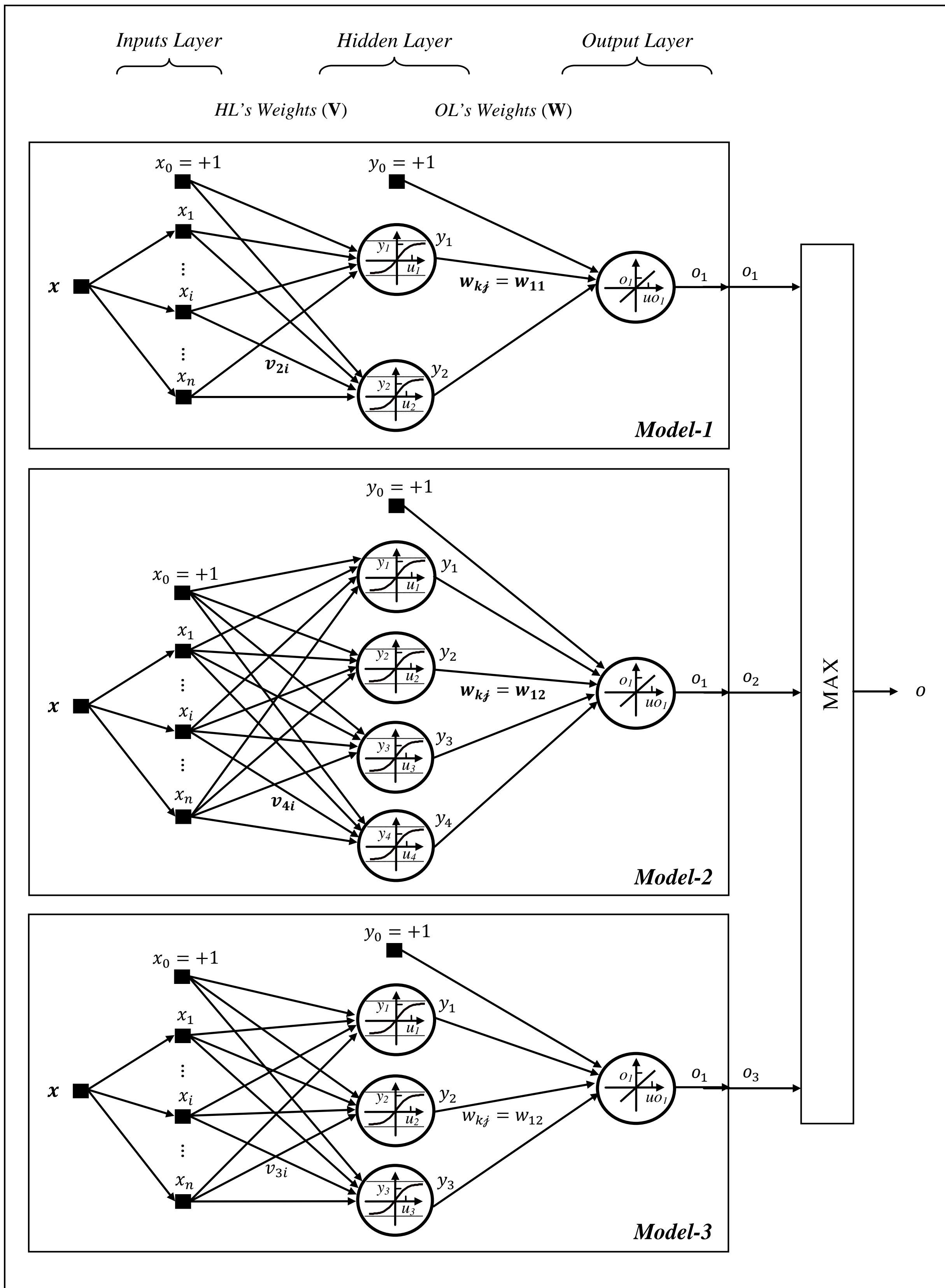

Figure 4.2: *K Separate Models/One OL Neuron* Structure

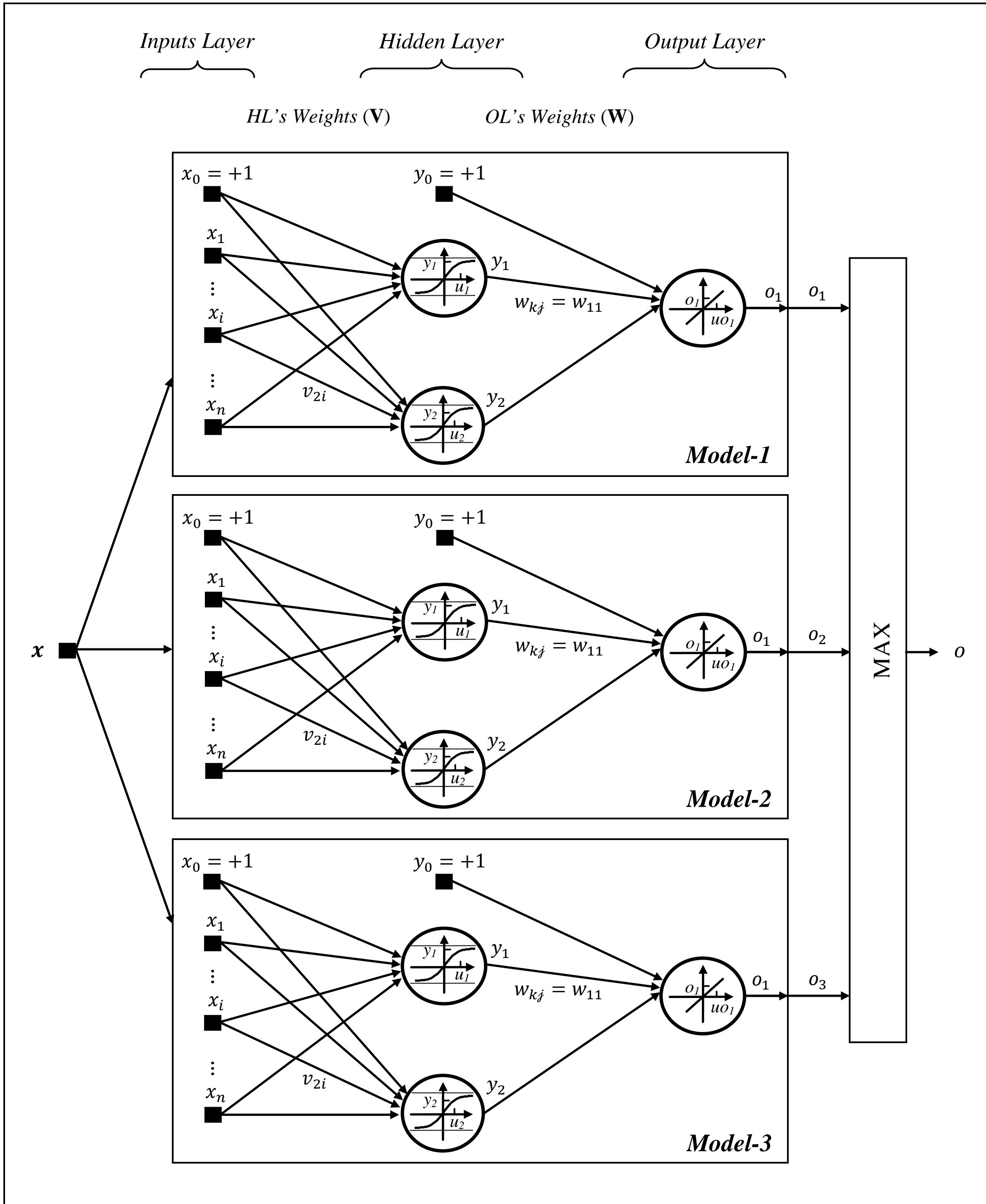

Figure 4.3: *K Joint Models/One OL Neuron* Structure

5 Experimental Results and Discussion

5.1 Controlling the Experimental Environment

The fast neural network algorithm is applied for three different MLP structures: o*ne model/K OL neurons*, *K separate models/One OL neuron*, and *K joint models/One OL neuron*, to solve nonlinear multiclass classification tasks. Hyperbolic tangent activation function was applied in the HL for all three structures, and linear activation function was applied in their output OL. Linear activation function in the OL indicates that the **W** weights were directly computed by using pseudo-inverse method that always finds the local minimum of a cost function for a given HL weights **V**. Fixed parameters (table 5.1), and variable parameters (table 5.2) were also constant during the experiment.

Table 5.1: Experimental Fixed Parameters

Parameters	Values
η_m	0.75
kw	0.1
W-direct	1
K-fold	10
Validation	1

Table 5.1 involves the values of experimental fixed parameters that were used during all of the experiment. The momentum term was used to speed up the convergence to the minimum of a cost function. Therefore, the value $\eta_m = 0.75$ was perfect for convergence purposes. The value *kw* = 0.1 is used to initiate the HL weights **V** in the small range [-0.1, +0.1]. The parameter *W-direct* indicates that the weights **W** of the OL were calculated directly by using the pseudo-inverse method. *K-fold* parameter specifies that 10-fold cross validation was used during the training process. The last parameter, *Validation*, shows the number of validation parts after the training process is finished. Using *Validation* = 1 means that all patterns (or instances) of a given data set have been used during validation phase.

Moreover, experimental results are obtained by using the variable parameters listed in table 5.2, which were constant for all experimental data sets that trained in all three MLP structures. The purpose behind making the variable parameters constant during training phase of all three MLP structures is to control the experimental environments which could affect the accuracy and/or time consumption. The variable parameters are number of neurons in the HL *J*, learning rate η, and number of iterations during training phase *iterations*. As a first step in the experiment, the network weights **V** and **W** were initiated at random, thus, by using *seed* =1, **V** and **W** always have the same initiation matrices during all experiments.

Table 5.2: Experimental Variable Parameters

Variable Parameters	Values
J	[2:2:24]
η	[0.0000001, 0.00001, 0.0001, 0.001, 0.004, 0.005, 0.025, 0.010]
iterations	[100, 250, 400, 550, 700, 1000]
Seed	1

5.2 Comparison of Three Different MLP Structures

In term of accuracy, experimental results show that *K joint models/One OL neuron* is the overall best MLP structure that applied the fast neural network algorithm to solve multiclass classification tasks. However, it has the largest structure, meaning the biggest number of neurons in the HL, and thus its training CPU time was the longest. The following subsections will discuss the results in terms of accuracy, structure size and the CPU time consumption.

5.2.1 Comparison of Three Different MLP Structures in Term of Accuracy

Table 5.3 summarizes the experimental accuracy values and the overall averages of applying the fast neural network algorithm on three different MLP structures for eleven data sets (described in section 3.5). For every data set, each accuracy value appeared is an average accuracy of overall data set that is obtained after training all patterns on the best found variable parameters (*J, η, and iterations*). The best chosen variable parameters are based on finding the minimum averaged error of unseen testing data for all 10-fold cross validation parts. A bold value in each data set shows the best MLP structure in its averaged accuracy among others. The star (*) symbol indicates that the obtained averaged accuracy is for two-class data sets. Averages on last row of table 5.2 emphasize which its associated MLP structure is the best on overall accuracy of eleven data sets.

Table 5.3: Accuracy of Three MLP Structures

Data Sets	One Model / *K* OL Neurons	*K* Separate Models/One OL Neuron	*K* Joint Models/One OL Neuron
Iris	98.67	**99.33**	98.00
Glass	92.99	88.32	**94.39**
Vote	96.98	96.98	**97.41***
Wine	**100.00**	**100.00**	**100.00**
Teach	**90.73**	80.79	85.43
Sonar	**100.00**	**100.00**	**100.00***
Cancer	90.40	90.40	**91.41***
Dermatology	**98.91**	98.36	98.36
Heart	66.67	62.96	**71.38**
Prokaryotic	97.39	**97.49**	**97.49**
Eukaryotic	82.04	**90.52**	**90.52**
Average	**92.25**	**91.38**	**93.13**

Experimental results on table 5.3 for eleven data sets show that the overall averaged accuracy of *K joint models/One OL neuron* structure is better than o*ne model/K OL neurons*, and then the o*ne model/K OL neurons* structure is better than *K separate models/One OL neuron*. Two data sets (Wine, and Sonar) gained 100% accuracy for all three MLP structures. Thus, by eliminating them, the number of data sets decreased from eleven to nine. Ranking technique was used in order to give a score to every structure and then evaluate them fairly. The *K joint models/One OL neuron* has the best score of 50. In this scoring system, the higher the value the better the structure is. Figure 5.1 graphically represent the scores of ranking three different MLP structures.

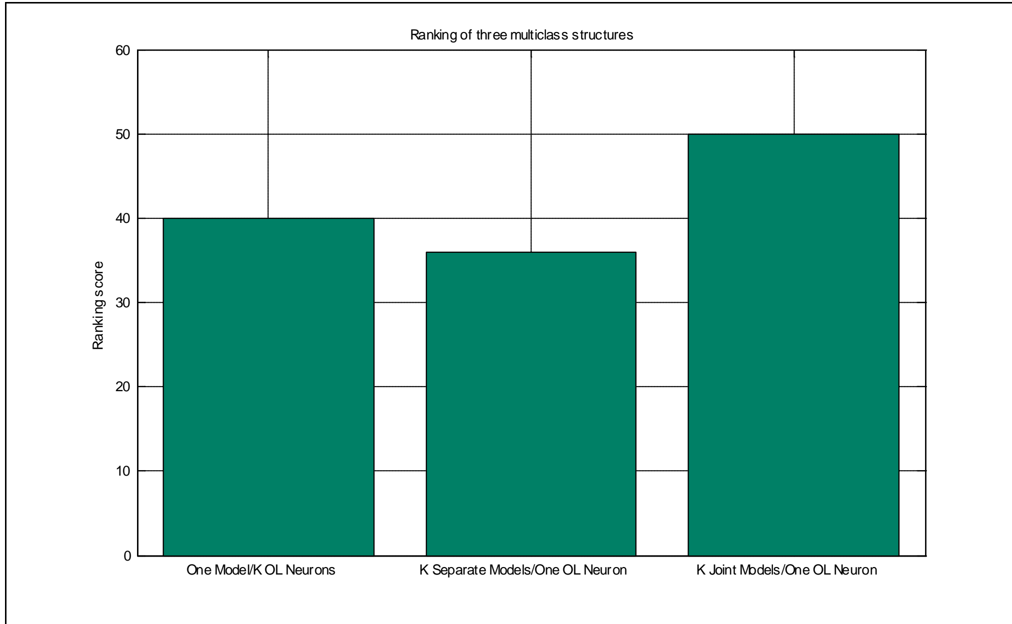

Figure 5.1: The Scores of Ranking Three Different MLP Structures

The following graph depicts the accuracy of different MLP structures that is listed in table 5.3.

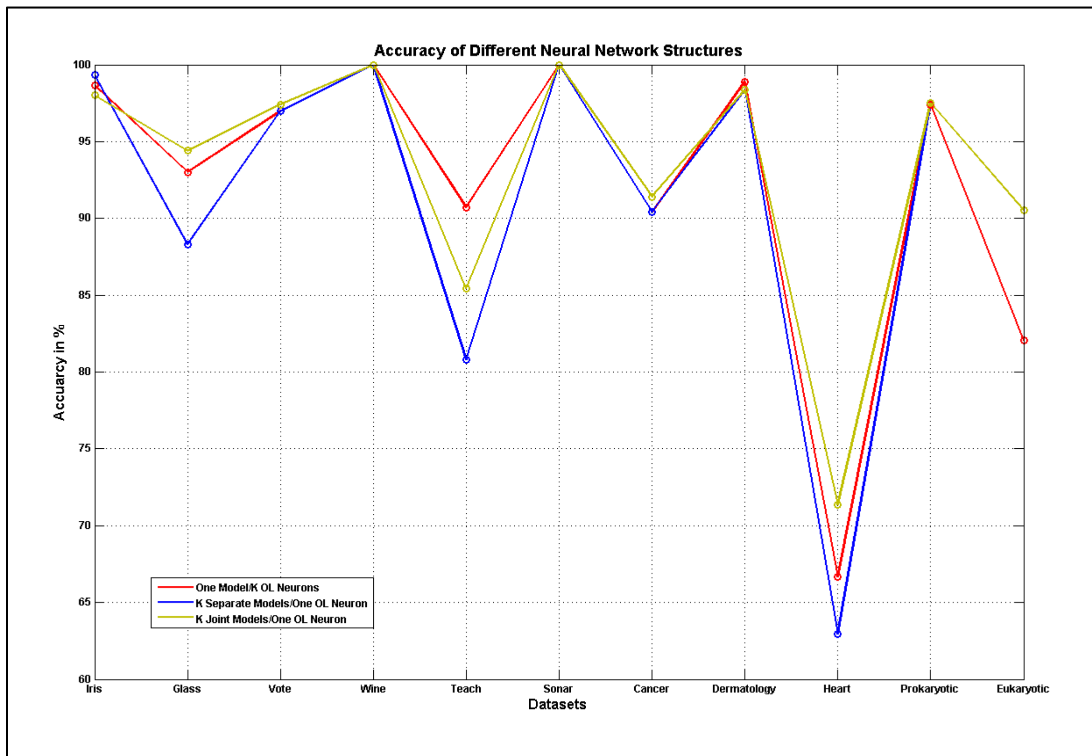

Figure 5.2: The Accuracy of Different MLP Structures of Eleven Data Sets

Three averaged accuracy plots are depicted in figure 5.2 for eleven data sets. The red curve represents the average accuracy of *one model/K OL neurons* structure, the blue is the average accuracy line of *K separate models/One OL neuron* structure, and finally the average accuracy curve of *K joint models/One OL neuron* structure is depicted by the olive color. As it is shown in figure 5.2, all three structures have the same average accuracies on the two data sets Wine and Sonar at 100%. It also confirmed that the best average accuracy curve is the *K joint models/One OL neuron*, since its curve is the upper among other curves. Furthermore, the curve of *one model/K OL neurons* obviously is located between other two curves. Finally, the *K separate models/One OL neuron* almost has the lowest averaged accuracy among other structures which is shown by its curve that almost always fell under all other curves. Despite having the lowest average accuracy, this structure achieved the highest accuracy for the Iris data set.

Another representation of experimental results is in figure 5.3 which shows the averaged accuracy of three MLP structures, *one model/K OL neurons*, *K separate models/One OL neuron*, and *K joint models/One OL neuron* that are depicted in red, blue, and olive columns respectively.

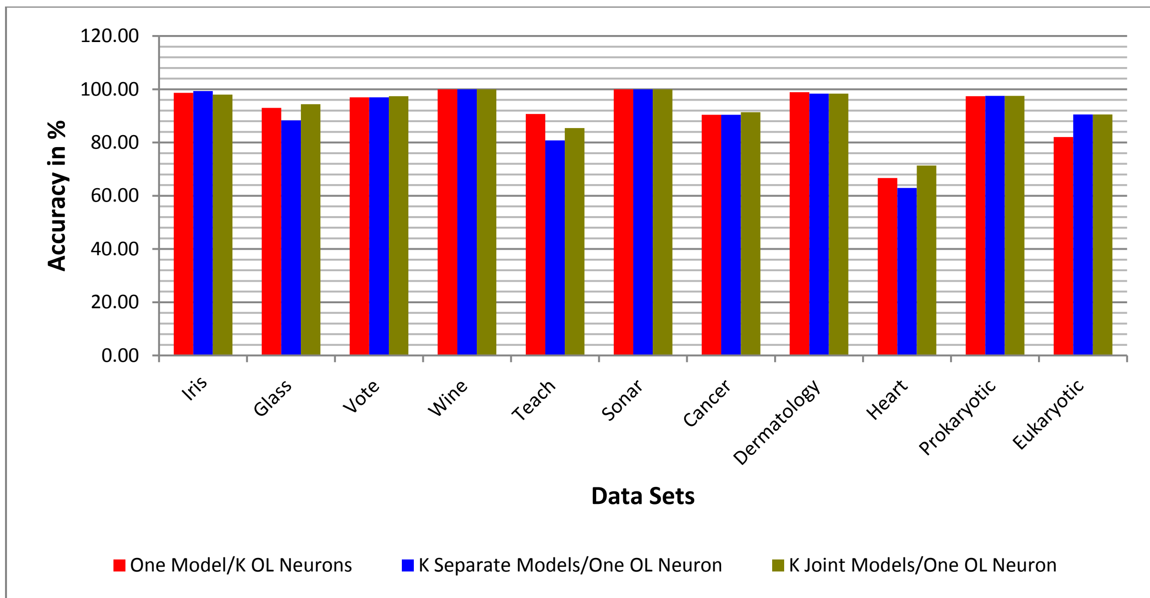

Figure 5.3: The Accuracy of Different MLP Structures of Eleven Data Sets

By almost eliminating the similar averaged accuracy among three structures, which are 6 data sets, Iris, Vote, Wine, Sonar, Cancer, Dermatology, and Prokaryotic, the other 4 data sets, Glass, Teach, Heart, and Eukaryotic, have a clear diversity in their averaged accuracy between three MLP structures. Consequently, it is obvious that *K joint models/One OL neuron* structure is the best structure in term of accuracy in 3 out of 4 data sets. The reason why the *K joint models/One OL neuron* performs the best on average is possibly coming from the famous theorem in optimization that '*Sum of Optima, Is Not Optimal*'. Applied to our structures, it basically says that only by optimizing all the sub-models jointly leads to their best overall performance. In machine learning, the *K joint models/One OL neuron* structure is usually named a One-versus-All (OvA) model.

5.2.2 Comparison of Three Different MLP Structures in Terms of Structure Size

Table 5.4 shows the MLP structure size in terms of number of neurons in the HL. The small size is desirable. The bold value shows the smallest size of a MLP structure among others. It is obvious from the averaged size of all eleven data sets that the *one model/K OL neurons* structure always has the smallest structure, then it is followed by *K separate models/One OL neuron* and at last, *K joint models/One OL neuron* structure has the biggest averaged size.

Table 5.4: Number of HL Neurons of Three MLP Structures

Data Sets	One Model / *K* OL Neurons	*K* Separate Models/One OL Neuron	*K* Joint Models/One OL Neuron
Iris	20	26	**12**
Glass	**12**	40	96
Vote	**14**	28	16
Wine	**14**	22	54
Teach	**22**	52	54
Sonar	**24**	48	36
Cancer	**2**	4	4
Dermatology	**18**	74	144
Heart	**4**	10	60
Prokaryotic	**22**	54	42
Eukaryotic	**24**	84	96
Average	**16**	**40**	**56**

Table 5.4 is also represented graphically in figure 5.4. The red column represents *one model/K OL neurons* structure which has always the smallest size among other structures for 10 data sets out of 11. The *K joint models/One OL neuron* structure which is depicted as olive column has the biggest averaged size. *K separate models/One OL neuron* structure (depicted as blue) is between other two structures, but it is almost close in its size to the *K joint models/One OL neuron* structure than *one model/K OL neurons* structure.

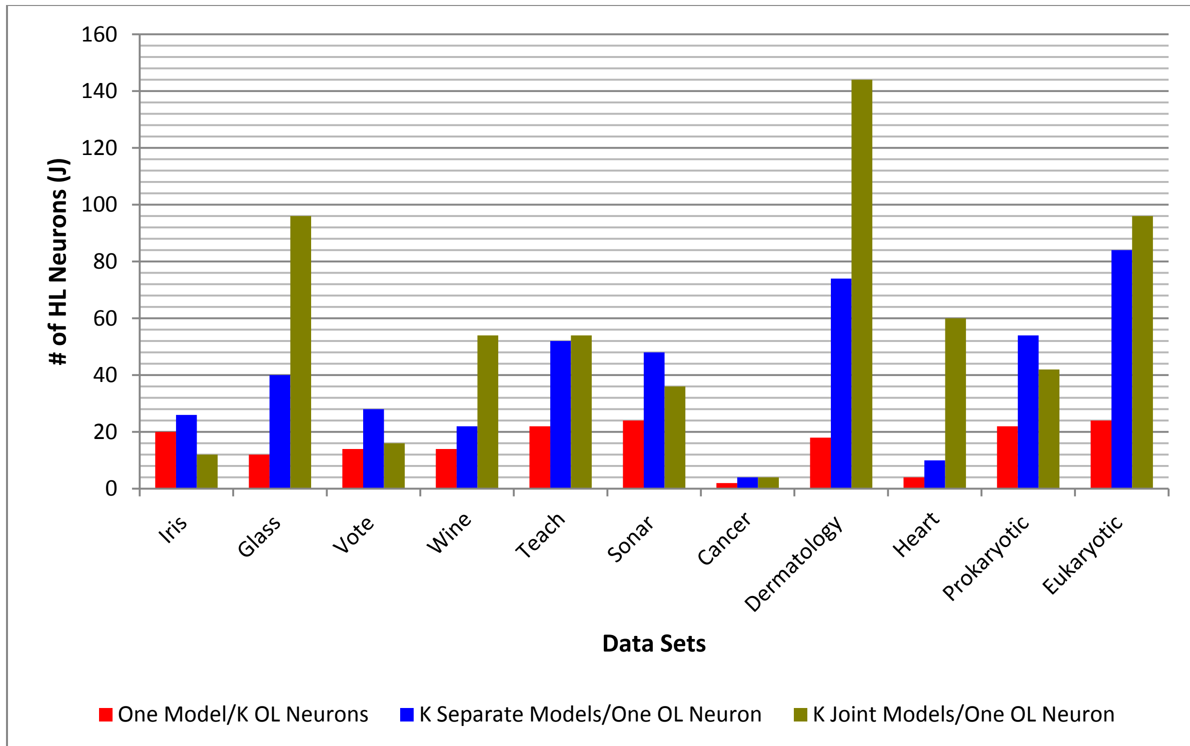

Figure 5.4: The Structure Size of Three MLP Structures for Eleven Data Sets

5.2.3 Comparison of Three Different MLP Structures in Term of Time Consumption

The experiment additionally discusses the experimental CPU training time in hours for all three MLP structures which is depicted in figure 5.5. A red curve represents experimental training time of *one model/K OL neurons*. A blue curve and an olive curve represent the experimental training time for *K separate models/One OL neuron* and *K joint models/One OL neuron* respectively.

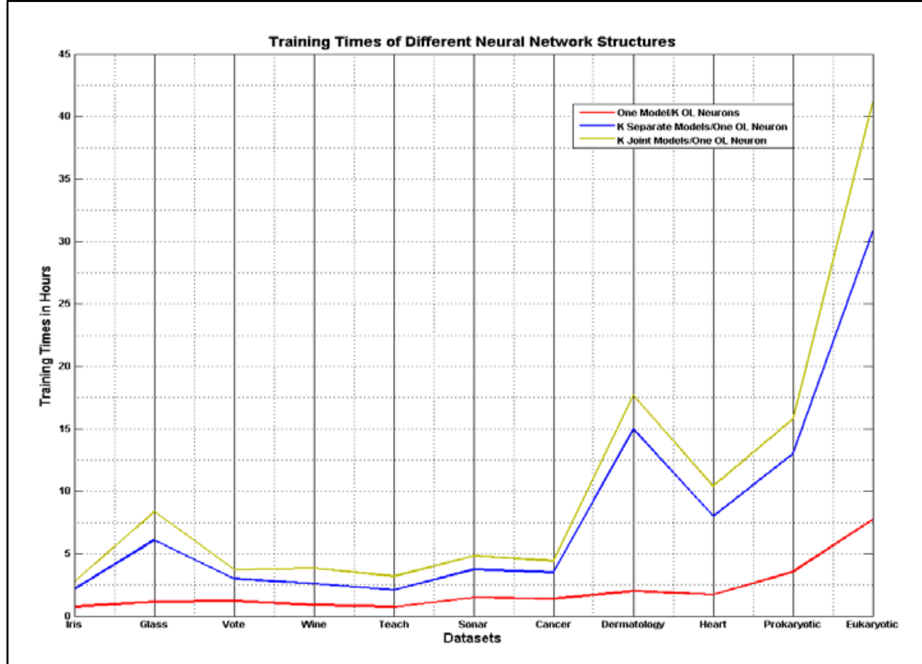

Figure 5.5: Training time of Different MLP Structures of Eleven Data Sets

For eleven data sets, figure 5.5 shows that the longest learning time is for *K joint models/One OL neuron* structure. *One model/K OL neurons* is the fastest MLP structure in its learning training time. Between previous structures, *K separate models/One OL neuron* is in the middle, but its training time is closer to *K joint models/One OL neuron* structure than *one model/K OL neurons* structure. The main reason for the difference between the two structures of *Ks (jointly and separately) models* and *one model* structure is due to the need to build *K* models, and then to spend time for training each model alone. This difference becomes huge when a large number of instances in a training set are considered, such as the last data set (Eukaryotic). In addition, the *K joint models/One OL neuron* structure has the biggest structural size in terms of number of HL neurons, and thus the training time needed for learning the structure was the longest one. Moreover, recall that for two-class data sets, we have used one neuron in OL of *one model/K OL neurons* structure and this required less time during the training phase of the

experiment. The following section demonstrates the accuracy and the training time results for two-class data sets that are trained in *one model/K OL neurons*.

5.3 Using a Neuron in the OL of One Model/*K* OL Neurons Structure for Two-Class Data Sets

Empirically, for two-class data sets, using one neuron in OL of one-model/K neurons structure gives almost the same results as using two OL neurons, however, it took less training time. The following results are for a data set, Vote, which is a two-class data set that used in the experiment. By using *one model/K OL neurons* structure, the following table 5.5 shows the accuracy and learning training time in hour of using one or two OL neurons in the two-class data set, Vote.

Table 5.5: The Accuracy of Using One or Two OL Neurons in Vote Data Set

Performance Measurement	**Using One OL Neuron**	**Using Two OL Neurons**
Accuracy	96.98	96.98
Training Time	**1.25**	**1.28**

It is obvious from table 5.5 that by applying fast neural network algorithm in the *one model/K OL neurons* structure, the accuracy for using a single neuron in the OL of two-class data set (Vote) is exactly the same as using two neurons. However, an OL neuron took less training time in hours (1.25) than using two OL neurons which is slightly bigger than the first one (1.28).

6 Conclusions

6.1 The Conclusion

The thesis develops a fast batch EBP algorithm which uses the pseudo-inverse method to calculate its output weights when linear neuron(s) is (are) in the output layer. The algorithm is used within the three different MLP structures in order to find the best structure that solves the nonlinear multiclass classification problems for 11 benchmarking datasets. The three different MLP structure are *one model/K OL neurons*, *K separate models/One OL neuron* and *K joint models/One OL neuron*.

The *K joint models/One OL neuron*, with a hyperbolic tangent as its HL activation function and the linear OL activation function, was the best in terms of accuracy among three structures. However, it is the biggest in the size which results in the biggest training time. The model accuracy is more significant than the elapsed learning time and the structure size, because the data sets used fall into the category of small to middle size datasets. Thus, one can say that for such datasets the best choice of the NN structure is the NN having *K* joint models and each of

them having one OL linear neuron. However, if one wants to model bigger datasets (say with more than 10,000 patterns) the best choice may well be the single model with *K* OL neurons because its accuracy is very close to the *K joint models/One OL neuron* structure but it needs a significantly smaller training time and it is of a much smaller size, meaning it will be faster in an on-line prediction (applications).

6.2 Future Works

There are several possible extensions of the work done here. First, it may be interesting to compare the accuracies obtained on the 11 datasets used here to the accuracies provided by other machine learning approaches such as support vector machines, adaptive local hyperplanes, *k*-nearest neighbors, decision trees and others. Next, the suitability of the developed fast EBP algorithm for huge data sets should be investigated and compared to the others models accuracies and training speed. One interesting line of the research may also be to develop a parallel version of the existing code and see its performance in terms of the speed of the training. Finally, the research done here may possibly and the most likely continue in developing of a semi-batch algorithm for handling large and ultra-large datasets (say when there are more than 1 million samples). This may well be the most valuable extension of the work done here because NN are not being used for such datasets as of today.

References

[1] A. Reinhardt, T. Hubbard, Using neural networks for prediction of the subcellular location of proteins, Nucl. Acids Res. 26 (1998) 2230–2236.

[2] Asuncion, A., Newman, D. J., “UCI ML Repository”, [http://www.ics.uci.edu/~mlearn/MLRepository.html], Irvine, CA, University of California, School of Information and Computer Science, 2007.

[3] Du, K L, Du, & Swamy. (2006). Neural networks in a softcomputing framework. Springer-Verlag.

[4] Haykin, S S. (2009). Neural networks and learning machines.

[5] Kecman, V. (2001). Learning and soft computing. Cambridge Mass. [u.a.]: MIT Press.