EMPIRICAL ANALYSIS OF NEURAL NETWORKS TRAINING OPTIMISATION

By

Mutamba Tonton Kayembe

Student Number: 678213

Supervisor: Ms. Nothabo Ndebele

A Dissertation submitted to the Faculty of Science, University of the Witwatersrand, Johannesburg, in fulfilment of the requirements for the degree of Master of Science

in

Mathematical Statistics

School of Statistics and Actuarial Science

October 2016

DECLARATION

I hereby declare that this Dissertation is my own genuine research work. It is being submitted for the Degree of Master of Science at the University of the Witwatersrand, Johannesburg. It has not been submitted for any degree or examination at any other University.

Mutamba Kayembe

October 2016

ABSTRACT

Neural networks (NNs) may be characterised by complex error functions with attributes such as saddle-points, local minima, even-spots and plateaus. This complicates the associated training process in terms of efficiency, convergence and accuracy given that it is done by minimising such complex error functions. This study empirically investigates the performance of two NNs training algorithms which are based on unconstrained and global optimisation theories, i.e. the Resilient propagation (Rprop) and the Conjugate Gradient with Polak-Ribière updates (CGP). It also shows how the network structure plays a role in the training optimisation of NNs. In this regard, various training scenarios are used to classify two protein data, i.e. the Escherichia coli and Yeast data. These training scenarios use varying numbers of hidden nodes and training iterations. The results show that Rprop outperforms CGP. Moreover, it appears that the performance of classifiers varies under various training scenarios.

ACKNOWLEDGEMENTS

The successful completion of this study is the result of invaluable support and contribution from many people. Firstly, I would like to declare my wholehearted gratefulness to my supervisor, Ms. Nothabo Ndebele, for her invaluable guidance and advice throughout this project. Her support and thoughtfulness are highly appreciated.

I cannot forget to thank the professors and lecturers at the School of Statistics and Actuarial Science for their invaluable contribution to my knowledge. I am honoured to have learned from them. Also, I am grateful to all my colleagues at the School, for their amity and encouragement. I wish them success for their forthcoming endeavour.

Lastly, I would like to give special recognitions to my family, especially my parents Dieudonné and Odette Kayembe, for their motivation, inspiration, love and constant support. I am grateful to have them in my life.

CONTENTS

LIST OF TABLES

LIST OF FIGURES

BASIC NOTATION AND ABBREVIATIONS

ANN: Artificial Neural Network

OA: Overall Accuracy

OAtrain: Overall Accuracy on training set

OAtest: Overall Accuracy on test set

BOAtrain: Best Overall Accuracy on training set

BOAtest: Best Overall Accuracy on test set

CG: Conjugate Gradient

GD: Gradient Descent

CGP: Conjugate Gradient Algorithm with Polak-Ribière update

FNN: Feedforward Neural Network

Rprop: Resilient Propagation Algorithm

MSE: Mean Squared Error

CHAPTER 1: STUDY INTRODUCTION

1.1 Introduction

This study is about optimising the training of artificial neural networks (ANNs) using empirical analysis. Understanding the basic problems faced in neural networks (NNs) training is necessary to comprehend the development of this study. Hence, this chapter introduces this study by explaining the basics of NNs training optimisation. Section 1.2 gives the background and states the problem of the study. The objectives of this study are discussed in Section 1.3. Finally, the structure of the report is given in Section 1.4.

1.2 Background

ANNs are techniques that simulate the physiological structure and functioning of human brain structures that can model very complex functions (McCulloch and Pitts, 1943). Based on biological NNs, ANNs are structured as interconnections of nodes referred to as artificial neurons, which transmit information between each other. Information is transmitted between nodes via synapses (connections between two neurons), which store parameters referred to as "weights". These weights quantify the strength of the connections and are adjusted in the manipulation of information, making ANNs adaptive to inputs and capable of learning (Gershenson, 2001).

ANNs have been implemented to solve complex classification and prediction problems in various areas such as medicine, geology, finance and engineering. Basically, the principal task in ANNs is to find an optimal mapping between the inputs and outputs of a process by minimising the error between the output and the target values of this process. This is achieved by using a training algorithm that

minimises the error function of the network (Rosenblatt, 1958; Duda, Hart and Stork, 2000).

Complex networks are likely to have error functions with attributes such as saddle-points, local minima, even-spots and plateaus that make training of ANNs very difficult. The presence of saddle points surrounded by high error plateaus can drastically slow down the training process of NNs. The presence of local minima creates situations where the training algorithm may be trapped in one local minimum instead of converging to the global minimum of the NN error functions; and as a result, the predictive and classification accuracies of NNs become very poor (Fukumizu and Amari, 1999; Anastasiadis, 2005; Akarachai and Daricha, 2007). Thus, the ability of a NN training algorithm to minimise such complex error functions is the critical point for the performance of ANNs (Riedmiller and Braun, 1993; Plagianakos, Magoulas and Vrahatis, 2001b; Magoulas, Vrahatis and Androulakis, 1997a; Igel and Husken, 2003). The problem of NNs training is very consistent with the problem of unconstrained and global optimisation theory (Livieris and Pintelas, 2009).

In the literature, various learning algorithms have been proposed to enhance the ANN's performance. Batch learning methods such as the back propagation algorithm are the most common. In the process of batch learning, the weight parameters are updated in the steepest descent direction of the gradient, using different adaptive learning rate for each weight (Prasad, Singh and Lal, 2013). However, learning algorithms such as the back propagation are characterised by slow training; and they do not converge to the global minimum from any starting set of weights. They often converge to local minima when training is initialised from a remote point to the global minimiser (Magoulas *et al.*, 1997a; Treadgold and Gedeon, 1998; Igel and Husken, 2003). Methods based on unconstrained and global optimisation theory, which apply the second derivative related information of the error function of NNs to speed up the learning process, have been proposed (Battiti, 1992; Moller, 1993; Van der Smagt, 1994; Magoulas, Vrahatis and Androulakis, 1997b). However, the extra computational cost required by these approaches does not guarantee acceleration of

the minimising process for non-convex functions (Nocedal, 1992; Anastasiadis, Magoulas and Vrahatis, 2003).

Other works such as Saurabh (2012), and Sheela and Deepa (2013), have been focused on developing methods for designing optimal NNs architecture (i.e., optimal number of hidden layers and optimal number of nodes in each hidden layer) to accelerate and optimise the training of NNs. While the actual problem being addressed by NNs easily gives the number of nodes to be used in the input and output layers, determining the optimal number of hidden nodes to be used has proven to be a challenging task. If a small number of hidden nodes that is inadequate to deal with the complexity of the problem data under study is used, then the NN may not be able to effectively fit the underlying pattern or relationship in the data (under-fitting). On the other hand, if excessive hidden nodes are used, then the NN may fit the underlying pattern and also the noise in the data (over-fitting). Besides, there is no consensus on the best approach to apply for determining the appropriate number of hidden nodes. It is generally argued that the merit of each approach is problem dependent (Lawrence, Giles and Tsoi, 1996; Saurabh, 2012; Sheela and Deepa, 2013). Hence, empirical analysis using real world data problems may better reveal the performance of particular approaches.

This dissertation investigates empirically how training algorithms and the structure (adequate number of hidden nodes) of the network impact the training optimisation of NNs. It presents two training algorithms believed to have good convergence ability and devised to overcome the drawbacks of the batch back propagation algorithm, namely the Resilient propagation (Rprop) algorithm and the Conjugate Gradient (CG) algorithm with Polak-Ribière updates (CGP). The Rprop is a Gradient Descent (GD) based training algorithm. It is based on the idea of mitigating the blurring effect in the adaptation process provoked by the unforeseeable behaviour of the size of the partial derivative on the weight update step, in the implementation of the back propagation algorithm. If the partial derivative size is too big, the algorithm can jump over the minimum of the error function without giving any indication (Riedmiller and Braun, 1993). For this reason, the Rprop is designed such that the direction of the weight update is only influenced by the sign and not the size of the

derivative. The magnitude of weight update is solely controlled by a specific weight "update-value". The Rprop is one of the best algorithms in terms of accuracy, robustness and convergence speed with regard to its learning parameters (Riedmiller, 1994; Anastasiadis, Magoulas and Vrahatis, 2005; Prasad *et al.*, 2013). The CGP algorithm is derived from the CG methods, which are used for large scale nonlinear unconstrained optimisation problems. The CGP is devised to converge faster than the GD based methods (Sharma and Venugopalan, 2014), and update the weights of the NN error function in conjugate directions. The CGP is one of the best performing CG algorithms (Jonathan, 1994; Hager and Zhang, 2006; Andrei, 2011; Ioannis and Panagiotis, 2012). To determine the adequate number of hidden nodes, this dissertation illustrates the use of the "trial and error" method which takes into consideration the complexity of the NN application problems. In the "trial and error" method, repeated training and checks are done with varying numbers of hidden nodes, until the optimal solution is found.

1.3 Objectives

The main objective of this study is to empirically investigate the capabilities of NNs training algorithms which are based on unconstrained and global optimisation theory. The aim is to assess their ability to globally minimise and converge, when dealing with extremely complex and non-convex error functions which result from NNs with many input and output values. Two learning algorithms designed for this purpose are considered, namely the Rprop algorithm and the CGP algorithm. Hence, in particular, the objectives of this study are as follows:

1. Implementation and analysis of a GD based training algorithm, specifically the Rprop algorithm.
2. Implementation and analysis of a CG based training algorithm, specifically the CGP algorithm.

3. Performance analysis and comparison of the two proposed training algorithms with regard to their efficiency (training speed), robustness (minimisation or convergence ability) and accuracy (generalisation ability).
4. Apply the trial and error method to analyse the performance of NNs training for varying number of hidden nodes.

1.4 Structure of the Dissertation

This dissertation is structured in six chapters. Chapter 1 introduces the study; it provides the background and states the problem of the study, and outlines the structure of the dissertation. Chapter 2 gives the literature review regarding the problem related to NNs training and optimisation. Chapter 3 discusses the NNs training and optimisation methods proposed to reach the objectives of this study. Also, it gives a detailed discussion of the performance measures. Chapter 4 describes the data and the experimental design used. Chapter 5 gives and discusses the results of the analysis. Chapter 6 gives the discussion and conclusions of the study.

CHAPTER 2: LITERATURE REVIEW

2.1 Introduction

In this chapter, an overview of the literature concerning ANNs is given. Section 2.2 describes the formulation of the training problem. Section 2.3 explains the difficulty of ANNs training in an optimisation context. Section 2.4 explains the optimisation methods for ANNs training. Section 2.5 focuses on the Gradient Descent (GD) based training methods, while Section 2.6 outlines the Conjugate Gradient (CG) based training methods. Finally, Section 2.7 gives the summary of this chapter.

2.2 Formulation of the Training Problem

ANNs have been applied in various problems such as signal processing and pattern recognition (Bishop, 1995), classification and approximation (Basheer and Hajmeer, 2000; Ferrari and Stengel, 2005), and load forecasting (Myint, Khin and Marlar, 2008; Frimpong and Okyere, 2010). They have been broadly proven to be powerful techniques for classification (Anastasiadis, 2005). Learning or training is crucial to the performance of NN models. For unsupervised learning, the network is only provided with input samples (Haykin, 1994; Sharma and Venugopalan, 2014).

Basically, learning is a process in which parameters (weights) of the network are iteratively updated, in order to minimise the error between the desired outputs (the outputs that should be produced by the network based on the problem data inputs being analysed and fed to the network) and actual outputs of the network. The goal is to create a correct input-output mapping so that, when presented with unseen inputs, the network can accurately predict their outputs (Haykin, 1994; Rojas, 1996). When the network is fed with input together with desired output samples, the learning algorithm is referred to as supervised. Supervised learning algorithms have been the

most employed to the training of ANNs (Basheer and Hajmeer, 2000; Anastasiadis, 2005). Thus, this study will focus on the analysis of well-performing supervised learning algorithms to implement on classification problems. Multilayer Feed-Forward NN (FNN) is one of the broadly applied classes of supervised NNs (Anastasiadis, 2005). For this class of networks, information is propagated only in the forward direction (i.e., information is transmitted in only one direction, forward; from input to hidden nodes, and from hidden to output nodes. The connections between nodes do not form a cycle). The input of each layer is the output of the preceding layer, and each layer is only connected to the preceding layer (Bishop, 1995; Rumelhart, Hinton and Williams, 1986). FNNs are powerful nonlinear models that can predict and classify more easily and quickly than other models. It has been proven that FFNs provide similar results to those of nonlinear statistical models (Ripley, 1993). FNNs can also be trained using nonlinear model methods such as CG and Levenberg-Marquardt algorithms (Hager and Zhang, 2006; Sharma and Venugopalan, 2014). Moreover, FNNs have demonstrated more efficiency at learning functions with discontinuities, than a number of other approaches can with stronger smoothness assumptions (Bishop, 1995).

A Feed-Forward Network can be formulated as follows:

$$net_j^l = \sum_{i=1}^{n_{l-1}} \omega_{ij}^{l-1,l}\, y_i^{l-1}, \qquad y_j^l = f(net_j^l) \tag{2.1}$$

where l stands for the number of layers in the network, net_j^l is the sum of the weighted inputs for the j-th node in the l-th layer ($j = 1, \dots, n_l$). The term $\omega_{ij}^{l-1,l}$ defines the weights from the i-th node at the $(l-1)$ layer to the j-th node at the l-th ($l = 2, \dots, L$) layer, and y_j^l is the output of the j-th node belonging to the l-th layer. The activation function of the j-th node is represented by $f(net_j^l)$.The weight parameters of the NN may be formulated utilising vector notation $\omega \in R^n$, as:

$$\omega = \left(\dots, \omega_{ij}^{l-1,l}, \omega_{i+1\,j}^{l-1,l}, \dots, \omega_{n_{l-1}j}^{l-1,l}, \theta_j^l, \omega_{i\,j+1}^{l-1,l}, \omega_{i+1\,j+1}^{l-1,l}, \dots\right)^T, \tag{2.2}$$

where T is the transpose symbol, θ_j^l represents the bias of the j-th node at the l-th layer, and n represents the total number of weights and biases in the network. Biases are weights connected with vectors which lead from a single node whose location is outside of the principal network and whose activation is always 1. The utilisation of biases in a NN augments the network capacity to solve problems by permitting the hyper-planes that discriminate individual classes to be offset for superior positioning (Reed and Marks, 1999; Anastasiadis, 2005).

The squared error over the training set, for a fixed and limited set of input-output samples p is:

$$E(\omega) = \sum_{k=1}^{p} \sum_{j=1}^{n_L} (y_{j,k}^L - t_{j,k})^2 = \sum_{k=1}^{P} \sum_{j=1}^{n_L} [\sigma^L(net_j^L + \theta_j^L) - t_{j,k}]^2 \qquad (2.3)$$

This equation defines the NN error function to be minimised; where $t_{j,k}$ is the target response at the j-th output node for the k-th sample $(k = 1, \dots, p)$ and $y_{j,k}^L$ is the output of the j-th node at layer L that depends on the network weights. The parameter sigma $\sigma = f(x)$, is often set to a nonlinear activation function such as the widely used logistic function $\frac{1}{(1+e^{-x})}$ or the hyperbolic tangent function $\tanh(x)$, where x is the sum of the weighted inputs to a node.

The power of FNNs has been well established by the following universal approximation property: "standard multilayer feedforward networks are capable of approximating any measurable function from one finite dimensional space to another, to any desired degree of accuracy, provided sufficiently many hidden units are available" (Hornik, Stinchcombe and White, 1989).

A multilayer feedforward network is said to be standard if it has one hidden layer. The above universal approximation property suggests that multilayer feedforward networks with as few as one hidden layer, can satisfactorily approximate any function faced in NNs applications. Theoretically, there are no limitations for the success of adequately configured standard feedforward networks to approximate any function. The lack of success can therefore be attributed to insufficient learning, inadequate number of hidden nodes or the lack of a deterministic relationship

between inputs and targets of the application (Hornik, Stinchcombe and White, 1989).

As useful as the above universal approximation property may be, it only gives the optimal architecture of a network in terms of number of layers, and does not give any indication on how to choose and update the weights and biases in order to obtain the desired accuracy. Besides, no indication is given on how to determine the appropriate number of nodes in the hidden layer. Hence, learning algorithms and methods for defining appropriate number of hidden nodes are required for training NNs to reach the optimal solution.

2.3 The Difficulty of Training

The ultimate goal of all supervised NNs is to reach the optimal solution, which has the smallest error between the actual outputs and the desired outputs of the networks. In optimisation terms, this solution is called the global minimum of the error function, and it is therefore the best possible solution of the NN (Haykin, 1994; Livieris and Pintelas, 2009; Livni, Shalev-Shwartz and Shamir, 2014). The NN error function is a multidimensional surface whose dimension depends on the number of connection weights of the network. The morphology of NN error functions has been found to be extremely complex in various studies (Rumelhart *et al.*, 1986; Haykin, 1994, Dauphin, Pascanu, Gulcehre, Cho, Ganguli and Bengio, 2014), because it is composed of many local minima and narrow steep regions next to wide flat ones. Therefore, the minimisation process of such error functions is a very difficult task (Igel and Husken, 2003; Anastasiadis, 2005; Akarachai and Daricha, 2007).

Moreover, the success of ANNs training is subject to many other parameters such as, the architecture (optimal number of hidden layers and hidden nodes), the number of training updates of weights, the activation functions, etc. (Sheela and Deepa, 2013). For instance, "the complexity of NNs training increases when dealing with optimisation problems related to arbitrary decision boundary with rational activation

functions" (Saurabh, 2012). For such problems, networks with two or three hidden layers are necessary to obtain an arbitrary degree of accuracy. Estimating the number of hidden nodes is of crucial importance. If the number of hidden nodes is inadequate to deal with the complexity of the data of the classification problem, then "under-fitting" may occur. That is, there are too few hidden nodes to effectively detect and model the signals in the problem data. If excessive hidden nodes are used, then "over-fitting" may occur; that is, there are too many hidden nodes which cause the NNs to fit the noise instead of the underlying relationship in the data. In other words, over-fitting occurs when the network starts to memorise the training data, instead of learning to generalise from the underlying trend (Saurabh, 2012).

Hence, determining the appropriate number of hidden layers and number of nodes in each hidden layer with regard to the complexity of the problem data, to prevent under-fitting and over-fitting, is of major importance in classification problems. Numerous methods have been suggested in the literature for this purpose (Sartori and Antsaklis, 1991; Blum, 1992; Arai, 1993; Hagiwara, 1994; Boger and Guterman, 1997; Berry and Linoff, 1997; Saurabh, 2012; Sheela and Deepa, 2013). For instance, Blum (1992) suggests that the number of hidden nodes should be between the number of input nodes and the number of output nodes. Boger and Guterman (1997) propose that the number of hidden nodes should be 2/3 (or 70% to 90%) of the number of the input layer nodes. If this does not produce satisfactory results, then the number of output layer nodes should be added to improve the results. Berry and Linoff (1997) suggest that the hidden nodes number should be less than twice the input nodes number. Recently, Sheela and Deepa (2013) have tested various criteria based on statistical errors to determine the hidden nodes number. They propose a criterion that estimates the hidden nodes number as a function of inputs nodes n, i.e. $(4n^2 + 3)/(n^2 - 8)$. They argue that this criterion can be appropriate for wind speed prediction after experimental study was done using real-time wind data.

However, none of the above described approaches for calculating the appropriate number of hidden layers and nodes provide a standard formula that is widely accepted. All these approaches are problem dependent. Most often, the complexity of data influences the determination of the number of hidden layer nodes. The broadly

used approach by researchers to determine the adequate number of hidden nodes is the "trial and error" method (Kazuhiro, 2010; Stuti and Rakesh, 2011; Saurabh, 2012; Sheela and Deepa, 2013). The "trial and error" method starts by randomly choosing a small number of hidden nodes to train the network, utilising the data sample of interest. If the network does not converge after a reasonable training time or epochs (training iterations), training is restarted a few more times (maybe 5 times) in order to make sure that it is not trapped in a local minima. If the network still does not converge, then the number of hidden nodes is increased and the network is allowed to train again. For a couple of times, the process of increasing the number of hidden nodes and checking for convergence (trial and error) is repeated. If there is still no improvement, then it may be necessary to start increasing the number of hidden layers until the network converges.

2.4 Optimisation Methods

A number of methods have been applied for supervised learning of NNs. The most used are the class of first order gradient based algorithms, which are linear approximations and do not require a large amount of computation per iteration (Battiti, 1992; Looney, 1997). Among the first order gradient based training algorithms, adaptive step size ones are the most used. Adaptive step size based algorithms search for the best step size by fine tuning it after each weight update. They operate by regulating the amount of changes in the weights space during learning, in order to simultaneously maximise the speed of the minimising process, and avoid oscillations in the search (Riedmiller and Braun, 1993).

Other methods based on unconstrained optimisation theory have been proposed. These methods apply second derivative related information of the error function to speed up the training process (Battiti, 1992; Moller, 1993; Van der Smagt, 1994). The Broyden-Fletcher-Goldfarb-Shanno (BFGS) (Gill, Murray and Wright, 1981), and CG (Moller, 1993) algorithms, are widely proposed in the literature. Another alternative to the widely known line search approach is the Levenberg-Marquardt

algorithm based on the model-trust region approach (Fletcher, 1981; Hagan and Menhaj, 1994). The above described algorithms are broadly applied for training FNNs. However, these algorithms are computationally expensive because of the second derivative based information they utilise when minimising the error function. Also, in many instances, these algorithms do not guarantee acceleration of the minimising process for non-convex functions, especially when starting far away from a minimum (Nocedal, 1992; Anastasiadis, 2005).

The problem of high computational cost has been reduced nowadays by the improvement of the processing capacity of modern computers. Still, there are some drawbacks in the application of these powerful second order algorithms in some cases. For instance, too many weights may make the direct use of second order algorithms impractical. Furthermore, these algorithms utilise approximations of the Hessian matrix. Sometimes this Hessian matrix may be badly scaled or close to singular during training. Consequently, the algorithms may yield poor results.

An intrinsic problem to first order and second order learning algorithms is that they converge sometimes to local minima. Although some local minima may produce satisfactory results, they frequently cause poor network performance. Global optimisation methods can be used to overcome this difficulty (Burton and Mpitsos, 1992; Plagianakos, Magoulas and Vrahatis, 2001a; Plagianakos *et al.*, 2001b; Treadgold and Gedeon, 1998).

2.5 Gradient Descent based Training Algorithms

Gradient Descent (GD) is the most popular category of algorithms applied for supervised NNs training. Batch Back-Propagation (BP) is the most broadly utilised algorithm of this category (Rumelhart *et al.*, 1986). This first order method follows the steepest descent direction of the gradient by updating the weight parameters with the objective of minimising the error function of the networks (Battiti, 1992). The weight update can be formulated as follows:

$$\omega(t+1) = \omega(t) - \eta g\{E(\omega(t))\} \tag{2.4}$$

where $\omega(t+1)$ is the weight at iteration $(t+1)$, and $\omega(t)$ is the weight at iteration t. The quantity $g\{E(\omega(t))\}$ is the gradient of the batch error function $E(\omega(t))$, and is calculated by using the chain rule on the layers of the FNN. The parameter η is the learning rate, whose optimal value depends on the shape of the error function (Rumelhart *et al.*, 1986). The learning rate is a very important parameter, and it is used in order to prevent the algorithm from converging to a maximum or a saddle point. Heuristically, it is suggested to choose a small learning rate as $(0 < \eta < 1)$. This is to guarantee convergence and prevent oscillations of the BP algorithm in the steepest descent of the error surface.

The BP algorithm is characterised by some serious limitations; which are, slow training and convergence to local minima. The presence of local minima in the error surface can cause the algorithm to reach a suboptimal solution instead of the global one. This leads to poor performance of NNs. This situation is a consequence of inadequate number of hidden nodes, along with inappropriate initial weight parameters (Gori and Tesi, 1992). The setting of the learning rate for each weight direction is very critical. Often, it happens that the learning rates are different for different weight directions of the error surface (Jacobs, 1988).

Significant improvements have been observed in the learning speed and convergence capability of first order adaptive learning rate based algorithms (Magoulas *et al.*, 1997b; Magoulas, Vrahatis and Androulakis, 1999; Magoulas and Vrahatis, 2000). The most remarkable is the performance of the Resilient propagation (Rprop) proposed by Riedmiller and Braun (1993). Rprop algorithm is the best in terms of accuracy, robustness and convergence speed with regard to its learning parameters (Igel and Husken, 2003; Anastasiadis *et al.*, 2003; Anastasiadis *et al.*, 2005; Prasad *et al.*, 2013).

Rprop is based on the idea of removing the bad effect that the size of the partial derivative has on the weight step. Therefore, the direction of the weight change is only influenced by the sign of the derivative. The magnitude of weight update is

solely controlled by a specific weight "update-value". Empirical evaluations have demonstrated that Rprop converges fast, but generally necessitates introducing or even fine tuning extra heuristics (Igel and Husken, 2003; Anastasiadis *et al.*, 2005). Furthermore, literature reveals the non-existence of theoretical results underpinning the development of Rprop adjustments. This is expected since heuristics may be unable to assure convergence to a local minimum of the error function when the computation of weight updates are based on adaptive learning rates for each weight. However, no assurance is given for a monotonic decrease of the network error function after each iteration, and for the convergence of the weight sequence to a minimum of the error function (Riedmiller and Braun, 1993; Igel and Husken, 2003).

Another method known as Improved Rprop (IRprop) algorithm has provided better convergence speed in comparison with prevailing Rprop related schemes, along with the BFGS and CG training schemes (Igel and Husken, 2003). It is obtained by modifying the Rprop algorithm, for which the choice to undo a step is rather subjective. Hence, the basic idea of IRprop consists of making the step reversal subject to the behaviour of the error. It proposes reverting weight updates that have provoked changes to the signs of the corresponding partial derivatives, only in case of an error increase. This technique is a backtracking to Rprop update for some or all of the weights, so that the decision about whether or not to revert a step is made for each weight individually (Riedmiller, 1994; Anastasiadis, 2005).

In general, the GD based training algorithm updates the weights by following the negative direction of the gradient, which is the direction in which the error function (performance function) is most speedily decreasing. However, this does not inevitably result in the fastest convergence (Hager and Zhang, 2006; Sharma and Venugopalan, 2014). On the other hand, in the CG algorithms discussed in section 2.6 below, the search is done by using conjugate directions, which mostly results in faster convergence than GD directions. Moreover, the CG algorithms necessitate only slightly more storage than the other algorithms. Hence, CG algorithms are suitable for networks with a huge weights number (Hager and Zhang, 2006; Livieris and Pintelas, 2009).

2.6 Conjugate Gradient based Training Algorithms

Conjugate Gradient (CG) methods are essential for minimising smooth functions, particularly when the dimension is high (Andrei, 2008; Ioannis and Panagiotis, 2012). They can be defined as conjugate direction or gradient deflection methods which are a mix of the GD based methods and Newton's methods (Newton's methods are the widely applied numerical methods for solving nonlinear equations of several variables, and for finding a root of the gradient in optimisation problems. Their detailed description can be found in Battiti (1992) and Murray (2010)). The principal advantage of CG methods is that they are not required to store any matrices as in Newton's methods or as in quasi-Newton methods, and they are devised to converge faster than the GD based methods. Moreover, they are usually much more stable to train and easier to check for convergence. The speed benefits of CG methods come from using the conjugate information during optimisation (Li, Tang and Wei, 2007).

The CG method was first developed by Hestenes and Stiefel (1952). In this seminal paper, they proposed an algorithm for solving symmetric, positive-definite linear algebraic systems. After a period of stagnation, the CG method was revisited and became the main active field of research in unconstrained optimisation. This method was first applied to nonlinear problems by Fletcher and Reeves (1964), which is commonly considered as the first nonlinear CG algorithm. Since then, various CG algorithms have been proposed. A survey of 40 nonlinear CG algorithms for unconstrained optimisation is provided by Andrei (2008). Although CG methods have been devised for more than five decades now, they still remain of great interest when it comes to solving large-scale unconstrained optimisation problems. This is due to their convergence properties, efficiency and simplicity in their implementation using computer codes.

CG methods are designed to converge in at most n iterations when applied to unconstrained quadratic optimisation problems in R^n, by following exact line searches. Nonetheless, they are used as well for non-quadratic problems, since

smooth functions display quadratic behaviour in the neighbourhood of the optimum. For such cases, the algorithm is restarted after every n iterations in order to enhance the convergence rate (Yabe and Takano, 2004).

CG weights $\{\omega_t\}$ update process can be formulated as follows:

$$\omega_{t+1} = \omega_t + \eta_t d_t \,, \qquad t = 0,1, \ldots \tag{2.5}$$

where t represents the current iteration, $\omega_0 \in R^n$ is a given initial point, $\eta_t > 0$ represents the learning rate, and d_t is a descent search direction described as:

$$d_t = \begin{cases} -g_0, & if\ t = 0, \\ -g_t + \beta_t d_{t-1}, & otherwise, \end{cases} \tag{2.6}$$

where $g_t = g\{E(\omega(t))\}$ is the gradient of $E = E(\omega(t))$ at $\omega_t = \omega(t)$, and β_t is the scalar parameter. Different choices of β_t have been suggested in the literature, which engender different CG methods. The widely known methods comprise the Fletcher-Reeves (FR) method (Fletcher and Reeves, 1964), the Hestenes-Stiefel (HS) method (Hestenes and Stiefel, 1952), and the Polak-Ribi`ere (PR) method (Polak and Ribi`ere, 1969).

In practical computation, the PR method works similarly to the HS method, and it is commonly said to be one of the most effective CG methods (Jonathan, 1994; Ioannis and Panagiotis, 2012). Despite the practical benefits of this method, it has the main weakness of lacking the global convergence ability for general functions; and consequently, it may be trapped in an infinite iterative process, without showing any significant progress (Powell, 1984). In order to remedy the convergence inability of the PR method, Gilbert and Nocedal (1992), inspired by Powell (1986), suggested that the update parameter β_t is restricted to being nonnegative. This results into a globally convergent CG method (PR+).

Additionally, though the PR method and the PR+ method generally have better performance than the other CG methods, they cannot ensure the production of descent directions. Therefore, restarts are used in order to ensure convergence

(Powell, 1977). However, there is a concern related to restart algorithms. Their restarts may be activated excessively, resulting in the degradation of the overall effectiveness and robustness of the minimisation process (Nocedal, 1992).

Significant efforts have been made in the past decade in the development of new CG methods that are, besides being globally convergent for general functions, computationally better than classical methods and are categorised in two categories:

1. The first category uses second-order information to speed up CG methods by employing new secant equations (Li and Fukushima, 2001; Li, Tang and Wei; 2007). As an example, we can find the nonlinear methods introduced by Zhou and Zhang (2006), and Zhang (2009). Another method is the multistep CG method suggested by Ford, Narushima and Yabe (2008), which is based on the multistep quasi-Newton methods.

CG methods based on the modified secant equation, applying both the gradient and function's values with higher orders of accuracy in the approximation of the curvature, were introduced by Yabe and Takano (2004), and Li *et al*. (2007). These methods are globally convergent, and occasionally, can numerically perform better than classical CG methods, if appropriate conditions are satisfied. But, these methods do not guarantee the production of descent directions; hence, in practical analysis the descent condition is always assumed.

2. The second category tries to develop CG methods which produce descent directions, with the objective of avoiding inefficient restarts that occur frequently. Motivated by this idea, the search direction was modified in order to guarantee sufficient descent. That is, $d_t^T g_t = -\|g_t\|^2$ (Note that d_t^T is the transpose of d_t) , independent of the performed line search (Zhang, Zhou and Li, 2007; Zhang and Zhou, 2008). Also, modification of the parameter β_t results in a new descent CG method, known as the CG-DESCENT method proposed by Hager and Zhang (2006). In fact, they suggested to modify the Hestenes-Stiefel formula β_t^{HS}. Another modification of the PR method was proposed by Yuan (2009). He suggested the introduction of a parameter *C* that basically controls the

relative weight between conjugacy and descent. It is argued that this method possesses an important property, which is the global convergence for general functions.

In view of the above survey of CG methods, it is worth highlighting that various formulations of the parameter β_t, result in various CG algorithms. Also, it is commonly accepted that the CG algorithm with Polak-Ribière (PR) updates is one of the most effective CG methods (Andrei, 2008; Ioannis and Panagiotis, 2012). This is why this study focuses on the CG with PR, which is described in section 3.3.

2.7 Summary of the chapter

To summarise, it is broadly established that the problem of NNs training is related to the problem of unconstrained optimisation theory. More specifically, it is formulated as the minimising process of the error function $E(\omega)$ of the network, described as the sum of squared errors between a set of output and target values of a process. A widely used method to solve this problem is the BP algorithm, which is a GD based training algorithm that minimises the error function $E(\omega)$ by updating the weight parameters ω in the steepest descent direction of the gradient. However, the drawbacks of the BP algorithms are slow training and convergence to local minima (the prospect of being trapped in a local minimum) of the error function, leading to poor performance of NNs. This situation may be the consequence of inadequate number of hidden nodes, insufficient number of weights updates (training iterations), etc., especially when dealing with complex NNs (Prasad *et al.*, 2013).

Different approaches have been proposed in the literature to overcome the drawbacks of the BP algorithm; such as the Rprop and CG algorithms. The Rprop is one of the best algorithms in terms of accuracy, robustness and convergence speed with regard to its learning parameters (Riedmiller, 1994; Anastasiadis, 2005; Prasad *et al.*, 2013). It is based on the idea of mitigating the harmful effect caused by the size of the partial derivative on the weight step. Hence, the direction of the weight change is

only influenced by the sign of the derivative. The magnitude of weight update is solely controlled by a specific weight "update-value". The CG methods are very important for unconstrained minimisation of functions, particularly when the dimension is high (Andrei, 2011; Ioannis and Panagiotis, 2012). They can be defined as conjugate direction or gradient deflection methods. They are devised to converge faster than the GD based methods. It is argued that the CG methods are suited to train complex NNs given their ability to solve large-scale unconstrained optimisation problems. This is due to their convergence properties, efficiency and simplicity in their implementation using computer codes. CGP is one of the best performing CG algorithms.

CHAPTER 3: METHODS

3.1 Introduction

This chapter discusses the two NNs learning algorithms used for the purpose of this study, and the methods used to assess the performance of classifiers. Section 3.2 gives a detailed discussion of the Resilient propagation algorithm, and Section 3.3 focuses on the Polak-Ribière conjugate gradient algorithm. A detailed discussion of the measures of performance and the methods used to estimate them is given in Section 3.4.

3.2 Resilient Propagation

3.2.1 Description

Resilient propagation (Rprop) is one of the fastest (gradient descent) training algorithms in existence. It is a supervised batch learning method based on adaptive gradient with individual step sizes. The Rprop is very appropriate for cases where the gradient is approximated numerically and the error is noisy (Igel and Husken, 2003). It is straightforward to implement using computer program and is not subject to numerical problems (Patnaik and Rajan, 2000). The fundamental principle of the Rprop algorithm is to remove the bad effect that the size of the partial derivative has on the weight step in the basic back propagation algorithm. Therefore, the direction of the weight change is only influenced by the sign of the derivative. The magnitude of weight update $\Delta\omega_{ij}(t)$ is solely controlled by a specific weight called "update-value" $\Delta_{ij}(t)$; and Anastasiadis (2005) describes the update as follows:

$$\Delta\omega_{ij}(t) = \begin{cases} -\Delta_{ij}(t), & if \ \frac{\partial E(t)}{\partial \omega_{ij}} > 0, \\ +\Delta_{ij}(t), & if \ \frac{\partial E(t)}{\partial \omega_{ij}} < 0, \\ 0, & otherwise, \end{cases} \tag{3.1}$$

where $\partial E(t)/\partial\omega_{ij}$ is the actual summed gradient information obtained over all patterns of the entire training set. The update values are defined as follows:

$$\Delta_{ij}(t) = \begin{cases} \eta^{+}.\Delta_{ij}(t-1), & if \ \frac{\partial E(t-1)}{\partial \omega_{ij}} \times \frac{\partial E(t)}{\partial \omega_{ij}} > 0, \\ \eta^{-}.\Delta_{ij}(t-1), & if \ \frac{\partial E(t-1)}{\partial \omega_{ij}} \times \frac{\partial E(t)}{\partial \omega_{ij}} < 0, \\ \Delta_{ij}(t-1), & otherwise, \end{cases} \tag{3.2}$$

where $0 < \eta^{-} < 1 < \eta^{+}$.

The update-value $\Delta_{ij}(t)$ is reduced by the parameter η^{-} each time the partial derivative of the corresponding weight ω_{ij} changes its sign. A change in sign indicates that the previous update was too big and the algorithm has skipped over the local minimum. If the partial derivative of the corresponding weight ω_{ij} does not change the sign, the update value is slightly increased. This is to speed up convergence in shallow areas of the error surface. Moreover, in case of change in sign, no adaptation should be made in the subsequent learning step. Empirically, this can be done by setting $\partial E(t)/\partial\omega_{ij} = 0$ in the adaptation rule. This technique helps to accelerate the convergence process when the derivative is negative. However, when the two derivatives are positive, this approach can be ineffective since in such situation the weight updates may direct the trajectory of the weight far-off from the minimum or in areas yielding bigger error function values. In order to mitigate these problems, Rprop uses a heuristic parameter Δ_{max} that fixes the upper bound of the update step size (Anastasiadis, 2005).

3.2.2 Algorithm

The following pseudo-code gives a detailed description of the Rprop algorithm. For this algorithm, the *min()*(or *max()*) operator gives the minimum (or maximum) of two numbers. The *sign()* operator returns +1, if the argument is positive. It returns −1, if the argument is negative. It returns 0 otherwise (Riedmiller and Braun, 1993; Riedmiller, 1994).

Initialise $t = 0$,

For all nodes *i* and *j*, where the node connections go from *j* to *i* (or $\forall_{ij}$)

$$\forall_{ij}: \Delta_{ij}(t) = \Delta_0,$$

$$\forall_{ij}: \frac{\partial E}{\partial \omega_{ij}}(t-1) = 0,$$

Repeat for $t = 1, \dots, t_{max}$

Compute the gradient vector $g\{E(t)\} = \frac{\partial E}{\partial \omega}(t)$

for all weights and biases {

if $\left(\frac{\partial \mathrm{E}(t-1)}{\partial \omega_{ij}} \times \frac{\partial E(t)}{\partial \omega_{ij}} > 0\right)$ **then** {

$$\Delta_{ij}(t) = \min\left(\Delta_{ij}(t-1) \times \eta^{+}, \Delta_{max}\right)$$

$$\Delta\omega_{ij}(t) = -sign\left(\frac{\partial E(t)}{\partial \omega_{ij}}\right) \times \Delta_{ij}(t)$$

$$\omega_{ij}(t+1) = \omega_{ij}(t) + \Delta\omega_{ij}(t)$$

$$\frac{\partial E(t-1)}{\partial \omega_{ij}} = \frac{\partial E(t)}{\partial \omega_{ij}}$$

}

else if $\left(\frac{\partial E(t-1)}{\partial \omega_{ij}} \times \frac{\partial E(t)}{\partial \omega_{ij}} < 0\right)$ **then** {

$\Delta_{ij}(t) = \max\left(\Delta_{ij}(t-1) \times \eta^{-}, \Delta_{min}\right)$

$\frac{\partial E(t-1)}{\partial \omega_{ij}} = 0$

}

else if $\left(\frac{\partial E(t-1)}{\partial \omega_{ij}} \times \frac{\partial E(t)}{\partial \omega_{ij}} = 0\right)$ **then** {

$\Delta\omega_{ij}(t) = -sign\frac{\partial E(t)}{\partial \omega_{ij}} \times \Delta_{ij}(t)$

$\omega_{ij}(t+1) = \omega_{ij}(t) + \Delta\omega_{ij}(t)$

$\frac{\partial E(t-1)}{\partial \omega_{ij}} = \frac{\partial E(t)}{\partial \omega_{ij}}$

}

}

Until converged or stopping criterion is satisfied.

An algorithm is said to have converged during NNs training, when the minimum of the error function is reached. The stopping criterion is usually a predefined number of training iterations or a pre-specified error target.

3.2.3 Parameters

The Rprop algorithm uses the following parameters: 1) the initial update values Δ_0, 2) the maximum weight step size Δ_{max}, 3) the minimum weight step size Δ_{min}, 4) the increase factor η^{+}, and 5) the decrease factor η^{-}. In the implementation of the

Rprop algorithm, Riedmiller (1994) suggests the following set up of parameters: $\Delta_0 = 0.1$ (the choice of this value was proven to be uncritical, since it is updated as learning proceeds), $\Delta_{max} = 50$ (this is of great importance since it prevents the weights from becoming too large), $\Delta_{min} = 10^{-6}$, $\eta^+ = 1.2$, and $\eta^- = 0.5$.

3.3 Conjugate Gradient with Polak-Ribière Updates

3.3.1 Description

Conjugate gradient (CG) methods are powerful methods for solving large-scale unconstrained optimisation problems. They require low memory and have strong local and global convergence. Moreover, they are easy to implement using computer codes (Andrei, 2011). The basic idea of CG methods is to use conjugate information in determining the search direction of the minimum of an objective function. This is done by the linear combination of the negative gradient vector at the current iteration with the previous search direction as described by equation 2.6.

Various choices of the scalar β_t, known as the CG parameter, yield different CG algorithms. Hence, the formula definition of β_t is the fundamental element in any CG algorithm. For general non-linear objective functions, the Polak-Ribière formula of β_t (Polak and Ribière, 1969), which is a modification of the Fletcher-Reeves formula (Fletcher and Reeves, 1964), has demonstrated experimental superiority (Jonathan, 1994; Ioannis and Panagiotis, 2012); it is as follows:

$$\beta_t^{PR} = \frac{g_t^T y_{t-1}}{\|g_{t-1}\|^2}, \tag{3.3}$$

where g_t^T is the transpose of g_t which is as defined in (2.6); $y_{t-1} = g_t - g_{t-1}$ and $\|\cdot\|$ represents the Euclidean norm.

CG with Polak-Ribière updates (CGP) has been proven to be more robust and more efficient than the CG with Fletcher-Reeves updates (CGF) (Andrei, 2008; Andrei,

2011; Ioannis and Panagiotis, 2012). Its convergence is guaranteed by defining β_t as follows:

$$\beta_t^{PR+} = \max\{\beta_t^{PR}, 0\}. \tag{3.4}$$

Using β_t^{PR+}, is equivalent to restarting the CG search process if $\beta_t^{PR} < 0$. Restart the CG search process afresh in the direction of the steepest descent regardless of past search directions.

3.3.2 Algorithm

The following process gives a detailed description of the CGP algorithm for NNs training.

Step 1: Initialise ω_0, $0 < \sigma_1 < \sigma_2 < 1$, E_G, $\varepsilon \to 0$, and t_{max}; set $t = 0$.

Step 2: Compute $E_{\mathrm{t}} = E(\omega_{\mathrm{t}})$ and $g_{\mathrm{t}} = \frac{\partial E}{\partial \omega}(\mathrm{t})$, and set $g_{\mathrm{t}} = -d_{\mathrm{t}}$.

Step 3: Test the stopping criteria of training iterations. For instance, **if** $(E_t \leq E_G)$ or $(\|g_t\|_2 \leq \varepsilon)$, **then** stop; otherwise continue with step 4.

Step 4: Compute the step length (learning rate) η_t using the following strong Wolfe's line search conditions:

$$E(\omega_t + \eta_t d_t) - E_t \leq \sigma_1 \eta_t g_t^T d_t \tag{3.5}$$

$$|g(\omega_t + \eta_t d_t)^T d_t| \leq \sigma_2 |g_t^T d_t| \tag{3.6}$$

Step 5: Update the weights as follows:

$$\omega_{t+1} = \omega_t + \eta_t d_t \tag{3.7}$$

Step 6: Determine the modified Polak-Ribière scalar parameter β_t^{PR+}.

Step 7: Compute the descent search direction as:

$$d_t = -g_t + \beta_t^{PR+} d_{t-1} \tag{3.8}$$

Step 8: Check the restart criterion. For instance, if the restart criterion of Powell $|g_{t+1}^T g_t| > 0.2\|g_t\|^2$ is satisfied, then set $t = t + 1$, and go back to step 2.

3.3.3 Parameters

Any CG algorithm uses the following parameters: 1) the scalar CG parameter β_t, 2) step length (learning rate) η_t, and 3) the parameters of the strong Wolfe's line search conditions $0 < \sigma_1 < \sigma_2 < 1$. The most important of all parameters is β_t, because its modification produces a different type of CG method. The choice of parameters σ_1, σ_2 is of major importance, because it affects the direction of the line search. A wrong choice may prevent the Wolfe's strong conditions from generating a descent direction. Hence, $\sigma_1 = 0.01$, and $\sigma_2 = 0.1$ as proposed by Scales (1985). The value of η_t is dependent on the strong Wolfe's conditions.

3.4 Evaluating the performance of classifiers

The main objective of this study is to evaluate and compare the performance of Rprop and CGP algorithms in training NN classifiers using the E.coli and Yeast protein data sets. Model selection is important in our study since the best model to be selected for each method is the one that performs well in terms of classification accuracy, convergence and efficiency. The various performance measures are described in the following sections.

3.4.1 Estimating the Accuracy of Classifiers: Cross-validation Approach

The best classifier is the one that performs well on unseen data and therefore generalises well. In this section, an approach is presented that is used in an attempt to improve generalisation, and therefore, better estimate the overall classification accuracy of a classifier. This approach is the k-fold cross-validation technique (Kohavi, 1995). In this approach, a dataset D is randomly split into k mutually exclusive subsets (folds) $D_1, \cdots, D_k$ of approximately equal size. Then, k classifiers $\Phi_1, \Phi_2, \cdots, \Phi_k$ are trained. The classifier Φ_k is trained on the set $D_{train} = D|D_k$, and tested on the set $D_{test} = D_k$, where $k = 1,2, \ldots, k$. Figure 3.1 shows an example of 3-fold cross-validation. In this graph, the unions of the upper parts are utilised for training the classifiers, whereas the lower parts are utilised for testing the classifiers. The total number of classifiers Φ_k is 3. The disjoint union of the test sets gives the whole dataset, i.e. $\cup_{k=1}^{3} D_k = D$. Each set is therefore used once as a test set.

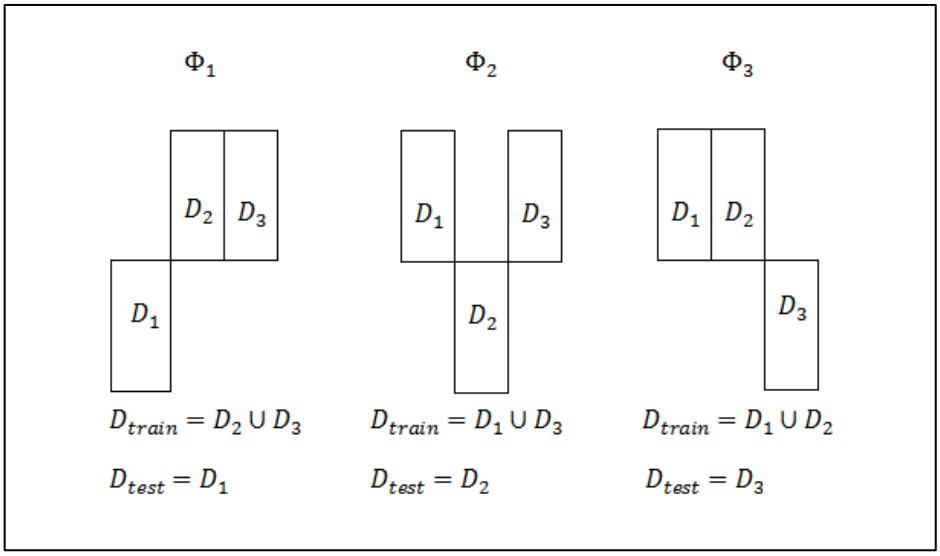

Figure 3.1: 3-fold Cross validation

The cross validation approximation of accuracy is the total number of correct classifications as a proportion of the total number of samples in the dataset. The basic assumption in cross-validation is: If Φ is a classifier trained on the whole data set *D*, and Φ_k are the classifiers trained on the sets $D|D_k$, then, the probability of correct classifications for Φ (i.e., $P(correct, \Phi)$) is equivalent to the probability of correct classifications for Φ_k for all $k = 1,2,\cdots,k$. Therefore, the estimate for the correct probability of the classifier Φ is the average of the estimates for the correct probabilities of the classifiers Φ_k. In practice, various numbers k of folds are often proposed; and in this study, $k = 10$. The rationale for utilising 10-fold cross-validation as the accuracy estimation method for this study is based on the work by Kohavi (1995); in which various real-world datasets are used to compare the performance of cross-validation (including the leave-one-out validation) and bootstrap methods, in estimating the accuracy of a classifier. Kohavi (1995)'s findings suggest that based on the trade-off between the variance and bias of the accuracy estimate, the 10-fold cross-validation produces better accuracies than bootstrap and the leave-one-out cross-validation, which requires more computation. The 10-fold cross-validation accuracy estimate is almost unbiased and has small variance, while bootstrap has small variance but extremely large bias on some problems. Also, "the leave-one-out validation" is almost unbiased; but it has high variance, leading to unreliable estimates" (Efron, 1983).

To estimate the overall accuracy (OA) of a classifier Φ, let us define first a confusion matrix *W* as follows:

$$W = \begin{pmatrix} N_{11} & N_{12} & \dots & N_{1z} \\ N_{21} & N_{22} & \dots & N_{2z} \\ \vdots & \vdots & & \vdots \\ N_{z1} & N_{z2} & \dots & N_{zz} \end{pmatrix} \tag{3.9}$$

where the diagonal elements N_{ii} are numbers of correct classifications, i.e. N_{ii} is the number of all s_i samples that are classified as s_i, and z is the number of classes. Also, the probability of correct classification of the classifier Φ is defined as follows:

$$P(correct, \Phi) = \sum_{i=1}^{z} P(s_i) P(\Phi(\mathrm{x}) = s_i | s_i) \tag{3.10}$$

where $P(\Phi(\mathrm{x}) = s_i | s_i)$ is the probability of correct classification of s_i, which is estimated as follows:

$$P(\Phi(\mathrm{x}) = s_i | s_i) = \frac{N_{ii}}{N_i} \tag{3.11}$$

By substituting (3.11) in (3.10) we obtain the following estimate:

$$\hat{P}(correct, \Phi) = \sum_{i=1}^{z} P(s_i) \frac{N_{ii}}{N_i} \tag{3.12}$$

Note that $P(s_i)$ is a prior probability which is defined as the proportion of class label *i* in the whole data *D* of size *N*, i.e. $P(s_i) = \frac{N_i}{N}$. Therefore, $\hat{P}(correct, \Phi)$ becomes

$$\hat{P}(correct, \Phi) = \sum_{i=1}^{z} P(s_i) \frac{N_{ii}}{N_i} = \sum_{i=1}^{z} \frac{N_i}{N} \frac{N_{ii}}{N_i} = \frac{1}{N} \sum_{i=1}^{z} N_{ii} \tag{3.13}$$

which is the proportion of correct classified samples. This means that $\hat{P}(correct, \Phi)$ is the OA.

The OA is derived from the whole dataset *D*. It is referred to as overall accuracy on training set (OAtrain) when derived from the training set D_{train}, and overall accuracy on test set (OAtest) when derived from the test set D_{test} (Breiman, Friedman, Olshen and Stone, 1984; Kohavi, 1995; Duda, Hart and Stork, 2000; Rudner, 2003).

3.4.2 Estimating the Convergence of Classifiers

The main target of training algorithms for NNs is to minimise the NN error function *E*. The convergence capability of a training algorithm is understood as the ability of this algorithm to converge to a minimum of the error function *E*. This is, starting from almost any initial set of weights, the sequence of the weights generated by the learning task will converge to a minimum of the error function. This minimum can be either local or global. In this context, the globally convergent algorithms are

different from the global optimisation methods (Nocedal, 1992; Treadgold and Gedeon, 1998). A strict mathematical understanding of global optimisation is, finding the complete set of the globally optimal solutions (global minimisers) ω^* of the objective (error) function E, with the corresponding global optimum value $E^* = E(\omega^*)$; whereas globally convergent algorithms converge with certainty (always) to a minimum, either local or global, from any remote starting point (Nocedal, 1992).

The common approach used in various NNs studies to empirically evaluate the convergence performance of different training algorithms is to conduct many training trials for each algorithm and calculate the percentage of trials for which each algorithm has converged (reached the minimum of the error function or the pre-specified error goal based on previous experiments). The best performing algorithm is the one with the highest percentage of converging trials (Veitch and Holmes, 1991; Anastasiadis, 2005). This approach is referred to as repeated training trials and is applied in this study. The repeated training trials approach has the merit of using different initial weights for each trial in order to address the issue of global convergence when training starts from any remote points (initial weights vectors) (Anastasiadis, 2005). In this study, the network error function utilised as measure of performance is the mean squared errors (MSE), which is minimised during training for the network to converge (Zhang, 2000; Sharma and Venugopalan, 2014). The MSE between the network's outputs $y_1, y_2, \dots, y_n$ and targets (desired outputs) $d_1, d_2, \dots, d_n$ can simply be formulated as follows:

$$MSE = \frac{1}{N}\sum_{n=1}^{N}(y_n - d_n)^2 \tag{3.14}$$

where N is the sample size.

For classification problems, a NN can be described as a mapping function between a set of inputs and outputs fed to the network, which is estimated in order to perform a particular classification task. For a particular classification problem, the process followed by NNs to calculate the MSE is illustrated using a single sample as follows. Consider the task of assigning an input vector X $\{x_i: i = 1, \dots, D\}$ to one of the Z classes $\{C_i: i = 1, \dots, Z\}$, where D is the number of attributes of X. Let define the

corresponding class of X as C_j, the network's actual outputs as $\{y_i(X): i = 1, \dots, Z\}$, and the network' s targets (desired outputs) for all output nodes as $\{d_i: i = 1, \dots, Z\}$. Note that the actual output of the network is a function of the input X, while the target of the network is a function of the class C_j to which X belongs. For a 1 of Z classification task, $d_i = 1$ if $i = j$ (X belongs to C_j) and 0 otherwise.

For training, the network parameters are selected in order to minimise the following objective function known as squared error function:

$$\Delta = E\{\sum_{i=1}^{Z}[y_i(X) - d_i]^2\} \tag{3.15}$$

where $E\{\cdot\}$ is the expectation operator. If we define $p(X, C_i)$ as the joint probability of the input X and the ith class C_i, and use the definition of expectation as provided by Richard and Lippmann (1991) and Zhang (2000), (3.15) can be expressed as follows:

$$\Delta = \int \sum_{j=1}^{Z}\{\sum_{i=1}^{Z}[y_i(X) - d_i]^2\}p(X, C_j)dX \tag{3.16}$$

Equation (3.16) is the sum of squared, weighted errors, which incorporates Z errors for each input-class pair. For a particular pair of input X and class C_j, each error, $y_i(X) - d_i$ is simply the difference of the actual network output $y_i(X)$ and the corresponding desired output d_i. The Z errors are squared, summed, and weighted by the joint probability $p(X, C_j)$ of the particular input-class pair. Expanding (3.16) as described by Richard and Lippmann (1991), yields:

$$\Delta = E\{\sum_{i=1}^{Z}[y_i(X) - E\{d_i|X\}]^2\} + E\{\sum_{i=1}^{Z} Var\{d_i|X\}\} \tag{3.17}$$

where $Var\{d_i|X\}$ is the conditional variance of d_i. The second term of the right-hand side is independent of the network outputs $y_i(X)$ and is called the approximation error. It reflects the inherent irreducible error due to the randomness of the data. The first term is affected by the effectiveness of the NN mapping and is known as the estimation error. Minimisation of Δ is achieved by choosing network parameters to minimise the first term, which is simply the MSE between the network outputs $y_i(X)$

and the conditional expectation of the desired outputs. Hence, when network parameters are selected to minimise a squared error objective function, "outputs estimate the conditional expectations of the desired outputs so as to minimise the mean-squared estimation error. For a 1 of Z problem, d_i equals one if the input X belongs to class C_i and zero otherwise" (Richard and Lippmann, 1991).

3.4.3 Estimating the Efficiency of Classifiers

Efficiency is another important factor that is worth considering in the development of NNs training algorithms. It is defined as the learning speed or convergence speed of an algorithm, i.e. how fast an algorithm reaches the minimum of the error function E during training. A broadly used approach to estimate the efficiency of NNs training algorithms is to measure the CPU time elapsed and the number of training iterations until convergence (Livieris and Pintelas, 2009). Hence, in this study, efficiency is estimated by the training time spent by an algorithm to achieve particular MSE values, and the training time is measured in seconds.

CHAPTER 4: DATA AND DESIGN OF EXPERIMENTS

4.1 Introduction

This chapter discusses the data and the experimental design applied to reach the objectives of this study. Section 4.2 focuses on the datasets; it gives a detailed description of the Escherichia coli (E.coli) proteins and the Yeast proteins classification problems. Section 4.3 is based on the experimental design, starting from the formulation of the classification tasks to the steps involved in the implementation of the experimental design. Finally, Section 4.4 gives the details on the software and computer used to implement the experiments, while Section 4.5 gives the summary of this chapter.

4.2 The Datasets

4.2.1 The Escherichia coli problem

The E.coli problem involves the classification of protein localisation patterns into eight classes. The dataset consists of 336 different proteins labelled according to 8 localisation sites, and can be found in the UCI Repository of Machine Learning database (Murphy and Aha, 1994).

As a prokaryotic gram-negative bacterium, E.coli is an essential element of the biosphere that settles in the lower intestine of animals to survive. Being a facultative anaerobe, it spreads to new hosts when released to the natural environment (Lodish, Berk, Zipursky, Matsudaira, Baltimore and James, 2003). E.coli is characterised by three principal and distinct types of proteins namely, enzymes, transporters and regulators. The enzymes constitute 34% (including all the cytoplasm proteins) of the E.coli proteins. The genes for transport functions come second followed by the genes for regulatory functions with 11.5%.

As proposed by Horton and Nakai (1996), the following 7 different attributes calculated from the amino acid sequences are used for this classification problem:

1. mcg: McGeoch's method for signal sequence recognition;
2. gvh: von Heijne's method for signal sequence recognition;
3. lip: von Heijne's Signal Peptidase II consensus sequence score (Binary attribute);
4. chg: Presence of charge on N-terminus of predicted lipoproteins (Binary attribute);
5. aac: score of discriminant analysis of the amino acid content of outer membrane and periplasmic proteins;
6. alm1: score of the ALOM membrane spanning region prediction program; and
7. alm2: score of ALOM program after excluding putative cleavable signal regions from the sequence.

The proteins in the E.coli dataset are distributed in 8 classes (localisation sites) as shown in Table 4.1.

Table 4.1: The E.coli proteins class distribution

Classes	Patterns
cytoplasm (*cp*)	143
inner membrane without signal sequence (*im*)	77
perisplasm (*pp*)	52
inner membrane, uncleavable signal sequence (*imU*)	35
outer membrane (*om*)	20
outer membrane lipoprotein (*omL*)	5
inner membrane lipoprotein (*imL*)	2
inner membrane, cleavable signal sequence (*imS*)	2
Total	336

.

4.2.2 The Yeast problem

The Yeast problem concerns the classification of protein localisation patterns into ten classes. It is based on a drastically imbalanced (the distribution of samples in the different classes is very unequal) dataset of 1484 different proteins labelled according to 10 localisation sites. This dataset can also be found in the UCI Repository of Machine Learning database (Murphy and Aha, 1994).

Saccharomyces cerevisiae, known as Yeast, is the most elementary Eukaryotic organism. It is a more complex life form than E.coli, and has various categories of proteins associated to the cytoskeletal cell structure, the nucleus organisation, membrane transporters and metabolic associated proteins (i.e., as mitochondrial proteins). The Yeast membrane transporter proteins are the most important since they are in charge for nutrient uptake, salt tolerance, resisting to drug, cell volume control, evacuating undesirable metabolites and identifying extra-cellular nutrients (Lodish *et al.*, 2003; Anastasiadis, 2005).

The following 8 different attributes are used for classification of the Yeast proteins into different sites:

1. mcg: McGeoch's method for signal sequence recognition;
2. gvh: von Heijne's method for signal sequence recognition;
3. alm: Score of the ALOM membrane spanning region prediction program;
4. mit: Score of discriminant analysis of the amino acid content of the N-terminal region (20 residues long) of mitochondrial and non-mitochondrial proteins;
5. erl: Presence of "HDEL" substring (thought to act as a signal for retention in the endoplasmic reticulum lumen). Binary attribute;
6. pox: Peroxisomal targeting signal in the C-terminus;
7. vac: Score of discriminant analysis of the amino acid content of vacuolar and extracellular proteins; and
8. nuc: Score of discriminant analysis of nuclear localization signals of nuclear and non-nuclear proteins.

The proteins in the Yeast dataset are distributed in 10 classes (localisation sites) as shown in Table 4.2.

Table 4.2: The Yeast proteins class distribution

Classes	Patterns
Cytosolic or cytoskeletal (*CYT*)	463
Nuclear (*NUC*)	429
Mitochondrial (*MIT*)	244
Membrane protein, no N-terminal signal (*ME3*)	163
Membrane protein, uncleaved signal (*ME2*)	51
Membrane protein, cleaved signal (*ME1*)	44
Extracellular (*EXC*)	35
Vacuolar (*VAC*)	30
Peroxisomal (*POX*)	20
Endoplasmic reticulum lumen (*ERL*)	5
Total	1484

.

4.3 Design of Experiments

4.3.1 Formulation of the Binary classifiers

The E.coli dataset and Yeast dataset under consideration in this study are multiclass datasets with number of classes, z greater than 2. Therefore, two different types of classifiers namely, multiclass classifiers and binary classifiers can be used to perform the classification task on these datasets. For multiclass classifiers, also known as "single machine" approaches, a multiclass classification problem is solved as a single optimisation problem to find z (which is the number of classes) functions simultaneously. When binary classifiers are applied to a multiclass classification problem, separate optimisation problems are solved; one for each of the binary

classification problems resulting from the multiclass classification problem (Erin, Robert and Yoran, 2000; Crammer and Singer, 2001; Hsu and Lin, 2002).

The separate optimisation problems underlying the binary classifiers approach have been proven less complicated to solve than single optimisation problems underlying the multiclass classifiers approach for solving multiclass classification problems. Also, when various binary classifiers are properly tuned and combined, they can be at least as accurate as a single multiclass classifier in solving a multiclass classification problem. In addition, the approaches for combining several binary classifiers for solving multiclass classification problems have a simple conceptual justification, and may be implemented to train faster and test as rapidly as the single multiclass classifier approach. It is therefore preferable and easier for practical purposes, to implement an approach that combines binary classifiers to solve multiclass classification problems (Rifkin and Klautau, 2004); which was done for the E.coli and Yeast classification problems under consideration in this study. This facilitates an understanding and comparison of the dynamic between different classes (Erin *et al.*, 2000), and allows performance comparison of the various binary classifiers.

The two widely applied approaches for combining binary classifiers to solve multiclass classification problems are the "One-Against-All" (OAA) and "One-Against-One" (OAO) approaches (Rifkin and Klautau, 2004). The OAA approach works as follows: for a multiclass classification problem with z as the number of classes, z different binary classifiers are trained; each one differentiates the samples in a single class from the samples in all the rest of classes. When a new sample needs to be classified, the z classifiers are run, and the one which produces the largest (most positive) value is selected. For the OAO also known as the "all-pairs" approach, $\binom{z}{2}$ different binary classifiers are trained; each one distinguishes a pair of classes. The classification of a new sample in the OAO approach is done similarly to that of the OAA approach (Erin *et al.*, 2000; Rifkin and Klautau, 2004).

The OAA approach, like the OAO approach, is conceptually simple, and can be as accurate as the single multiclass classifier approach (Rifkin and Klautau, 2004).

However, the OAA approach produces fewer binary classifiers than the OAO approach, which simplifies the presentation of results. For this reason, the OAA approach was applied for the purpose of this study. It is worth stressing that, "we are not stating that the OAA approach will perform substantially better than the other approaches. Instead, we are stating that it will perform just as well as these approaches, and therefore it is often to be preferred due to its computational and conceptual simplicity" (Rifkin and Klautau, 2004).

A classifier in our experiments refers to a fully connected feedforward NN that has an input layer, a hidden layer and an output layer. This section focuses on formulating the OAA approach that was applied in this study, to reduce a multiclass classification problem to z binary classification problems, where z is the number of classes. In the OAA approach, each class is compared to all others. For instance, we represent a binary classification problem for classes A and B by A/B. If B is the union of classes different from A, then we write $A/{\sim}A$ for A/B, which is interpreted as A (class1) versus no A (class2). Hence, a classifier $A/{\sim}A$ signifies that we consider the classification task of classifying A against all non A. The E.coli dataset and the Yeast dataset comprise 8 and 10 classes, respectively. This results in 8 binary classifiers and 10 binary classifiers for the E.coli dataset and the Yeast dataset, respectively. The E.coli binary classifiers and the Yeast binary classifiers are presented in Table 4.3 and Table 4.4, respectively.

Table 4.3: E.coli binary classifiers

Binary Classifiers	Total samples
$cp/\sim cp$	143/~163
$im/\sim im$	77/~259
$pp/\sim pp$	52/~284
$imU/\sim imU$	35/~301
$om/\sim om$	20/~316
$omL/\sim omL$	5/~331
$imL/\sim imL$	2/~334
$imS/\sim imS$	2/~334

Table 4.4: Yeast binary classifiers

Binary Classifiers	Total samples
$CYT/\sim CYT$	463/~1021
$NUC/\sim NUC$	429/~1055
$MIT/\sim MIT$	244/~1240
$ME3/\sim ME3$	163/~1321
$ME2/\sim ME2$	51/~1433
$ME1/\sim ME1$	44/~1440
$EXC/\sim EXC$	35/~1447
$VAC/\sim VAC$	30/~1454
$POX/\sim POX$	20/~1464
$ERL/\sim ERL$	5/~1479

The target coding differs between multiclass classifiers and binary classifiers. In our experiments, the following binary target coding for the output neurons is used; for

instance, in $cp/{\sim}cp$ (*cp* versus no *cp*), the targets have the following coding $\binom{1}{0}$ to denote class 1, i.e. *cp*, and $\binom{0}{1}$ to denote class 2, i.e. no *cp*.

4.3.2 Designing the architecture of the classifiers

Finding the most suitable FNN architecture (or appropriate number of hidden nodes) in terms of training speed and classification accuracy is one of the critical tasks, and often laborious, in ANN design (Blum, 1992; Boger and Guterman, 1997; Basheer and Hajmeer, 2000; Sheela and Deepa, 2013). Various rules of thumb for determining the appropriate number of hidden nodes are suggested in the literature (see section 2.3). When faced with new applications for which the networks may require the number of hidden nodes that do not conform to any of the already proposed rules of thumb, Basheer and Hajmeer (2000) suggest the use of the trial and error approach combined with one of the rules of thumb as a starting point. Another approach suggested by Basheer and Hajmeer (2000), is to utilise a small number of hidden nodes as the starting point, and build on as required to meet the model accuracy.

In this study, to determine the appropriate number of hidden nodes, we combined the above two approaches suggested by Basheer and Hajmeer (2000). The rule of thumb used as starting point for the number of hidden nodes is the one proposed by Sheela and Deepa (2013), where the number of hidden nodes is a function of the input nodes *n*, i.e. $(4n^2 + 3)/(n^2 - 8)$. For this rule of thumb, the approximate starting point for the number of hidden nodes is 5 for both the E.coli (for which the input nodes number is 7) and Yeast (for which the input nodes number is 8) ANN classifiers. Applying the trial and error approach, the number of hidden nodes was increased from 5 to 40 in order to address the issues of under-fitting and over-fitting by monitoring the change in network performance with regard to the change in the number of hidden nodes. For simplicity and clarity in the comparison and presentation of performance results of the different binary classifiers, only the

following 5 different numbers of hidden nodes in the interval of 5 to 40 was considered: 5, 10, 20, 30, and 40 hidden nodes. When there is no standard architecture for a particular ANN problem, it is required to conduct a set of preliminary experiments to find the most appropriate network architecture in terms of accuracy and training speed (Basheer and Hajmeer, 2000; Anastasiadis, 2005); suggesting that the choice of the various numbers of hidden nodes utilised to start with the preliminary experiments is subjective and left to the researcher or practitioner. There is no objective justification for using the above specified set of numbers for the hidden nodes as starting numbers for our experiments, since we had no prior knowledge as to the most appropriate architecture. One could have used a completely different set of numbers as starting points for the experiments. However, no matter what starting set of numbers are used, they will have to be continuously updated based on the network performance until the optimal ones are found.

Another important aspect to be considered in NN training is the determination of the optimum point (or moment) at which the training process should stop. It is common practice in NN studies to stop training when a particular conversion criterion has been satisfied (Hart and Stork, 2000; Anastasiadis, 2005). The widely used conversion criterion in the development of NN training algorithm is the minimum of the error function (Hart and Stork, 2000; Livieris and Pintelas, 2009). In other words, convergence occurs and training stops when the training algorithm reaches the minimum of the network error function. In this way, the conversion criterion influences the duration of the training process and the number of training cycles (or iterations) to be completed for training to stop. However, the training of complex networks may fail to converge, especially if the network error function is characterised by local minima and narrow valleys. Failure for the training algorithm to converge may be explained by the excessive oscillations that occur in narrow valleys of the error function during training. In this situation, if training is set to stop only when the minimum of the error function is reached (i.e. if the only specified convergence criterion is the minimum of the error function), then training might never stop, because the error may oscillate continuously (Rojas, 1994; Taguchi and Sugai, 2013).

To avoid the above described situation, and guarantee that training will stop in our experiments, we fixed the maximum numbers of iterations as stopping criterion for training. Note that fixing the maximum number of iterations will lead to the following situations. If a training algorithm is fast and the specified maximum number of iterations is too large, the algorithm will converge (the minimum will be reached) and training will stop before the maximum number of iterations is reached. But if a training algorithm is slow and does not converge to the minimum, then training will carry on and stop only when the maximum number of iterations is completed. In both cases, the training time and MSE achieved should be recorded. The faster converging algorithm will have shorter training time and smaller MSE (0 if convergence is achieved before the maximum number of iterations is reached). The slower converging algorithm will have longer training time and larger MSE. Therefore, fixing various numbers of iterations will not affect the results of the faster converging algorithm when compared to the slower one. In other words, the performance of the faster algorithm will not be underestimated and that of the slower algorithm will not be overestimated since the training times will depend on the number of iterations completed by both algorithms.

In this study, we applied various numbers of training iterations to evaluate the performance of the different binary classifiers with regard to the change in training iterations. Training for excessive number of iterations may result in overtraining (or over-fitting) of the network and training for insufficient number of iterations may result in under-fitting the network (Basheer and Hajmeer, 2000). Since we had no prior knowledge on the appropriate number of iterations to use, we applied the trial and error approach in our preliminary experiments to find the appropriate number of iterations for each binary classifier. The use of the trial and error approach for training iterations is justified by Basheer and Hajmeer (2000) who state that "the number of training cycles required for proper generalisation may be determined by trial and error. For a given ANN architecture, the error in both training and test data is monitored for each training cycle". The various numbers of iterations used for our experiments are as follows: 25, 50, 75, 100, 150, and 200 iterations. As for the

hidden nodes, one could have used a different set of numbers of iterations as starting point, and update them based on performance.

Based on the various numbers of hidden nodes and training iterations specified above, our experiments were conducted using 30 (i.e. combination of the 5 numbers of hidden nodes with 6 numbers of iterations) different training scenarios for each binary classifier. The notation 5hn, 10hn, 20hn, 30hn and 40hn corresponding to the various numbers of hidden nodes, and 25t, 50t, 75t, 100t, 150t and 200t corresponding to the various numbers of training iterations was used. For example, the notation 10hn_50t was used, respective of a network with 10 hidden nodes, trained for 50 iterations. Hence, for one binary classifier, we have 5 networks trained with 6 different numbers of training iterations. The number of input nodes for each network is equal to the number of attributes of the dataset used for training. Hence, for the E.coli dataset, all the networks have 7 input nodes; and for the Yeast dataset, all the networks have 8 input nodes. The number of output nodes for each network is 2, since we are dealing with binary classifiers. This design results in 30 different NNs for one classification task. Furthermore, this design allows studying the effect on the performance of a classifier for varying the number of hidden nodes and the number of training iterations. Figure 4.1 illustrates a NN classifier with P input nodes, 5 hidden nodes and 2 output nodes. P equals 7 for the E.coli classifier, and 8 for the Yeast classifier.

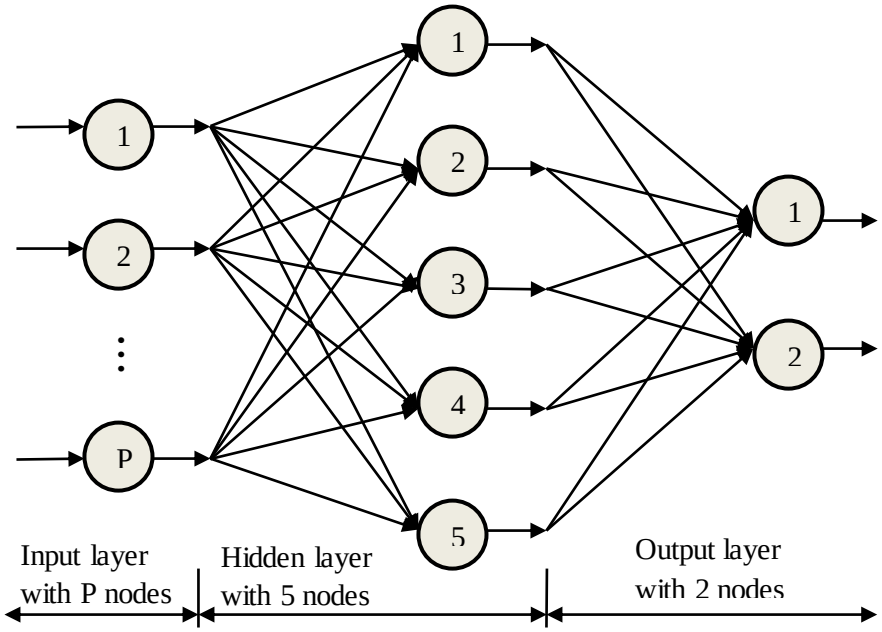

Figure 4.1: Neural network with P input, 5 hidden and 2 output nodes

4.3.3 Generalisation of the classifiers

As stated before, k-fold cross validation is one of the powerful methods used to generalise NNs classifiers. For our experiments, 10-fold cross validation (see Section 3.4.1 for justification of this choice) is applied to estimate how well the classifiers obtained will perform on unseen data (Kohavi, 1995). In this context, no specified error goal is used as early stopping criterion for our study. The only specified stopping criterion is the maximum number of training iterations to be completed. But if training is fast and the minimum of the error function is reached (when the MSE is 0) before the specified training iterations are completed, training will automatically stop at the minimum point (i.e. 0). It has been observed that the performance of ANN training algorithms is also most often dependent on the initial parameters such as the connection weights. Many trials with varying conditions of the initial weights are necessary to be able to compare the performance of different ANN training algorithms (Canu, 1993). For benchmarking experiments to be reliable, Harney (1992) argues that they normally need to be in the range between 25 and 100

independent trials; and it is common to report the mean results over the different trials (Veitch and Holmes, 1991; Anastasiadis, 2005). For our experiments in this study, 50 independent trials are conducted for each ANN binary classifier; and they results are averaged to obtain the overall performance. The 50 random initialisations of the weights are the same for both training algorithms. One could have used a different number of trials as long as this number is at least 25, as suggested by Harney (1992).

4.3.4 Initialising the weights

Practically, repeated random initialisation of weights using small values is sufficient to provide good convergence and avoid local minima. In our experiments, the Nguyen-Widrow function is used to initialise the first and second layer weights (Nguyen and Widrow, 1990). The transfer function used for both layers is the hyperbolic tangent sigmoid with the default parameters. The pre-processing of the data is done using the *mapminmax()* function. This function processes the data by mapping the minimum and maximum values of the input vectors to -1 and 1, respectively. The parameters of the Rprop and the CGP algorithms are set as described in Section 3.2.3 and Section 3.3.3, respectively.

4.3.5 Process implementation summary

This section describes the steps involved in the implementation of the experimental design for training the classifiers. The code developed for this purpose takes into account the variability of three parameters, i.e. the number of hidden layer nodes, the maximum number of training iterations, and the change of training and test sets in cross validation application to improve generalisation of the results on unseen data. Moreover, in order to regularise the results, many trials, i.e. 50, are used for every training task. The result for each classifier is the average performance over 50 trials.

Details of the steps involved in the process implementation can be found in appendix C. The flowcharts in Figures 4.2, 4.3 and 4.4 give a good visual of the 6 steps involved in the process implementation. Special attention is given to the depiction of for loops in steps 1 and 2. Figure 4.2 depicts step 1 of the whole process, while figure 4.3 does so for sub-process S of Figure 4.2. Note that sub-process S, is a sub-routine of step 1 that involves the implementation of the second, third and fourth for loops in step 1. The rest of the process (step 2 to step 6) is portrayed in Figure 4.4. Note that these flowcharts do not include all the variables and computations involved in each step. They just portray the main steps.

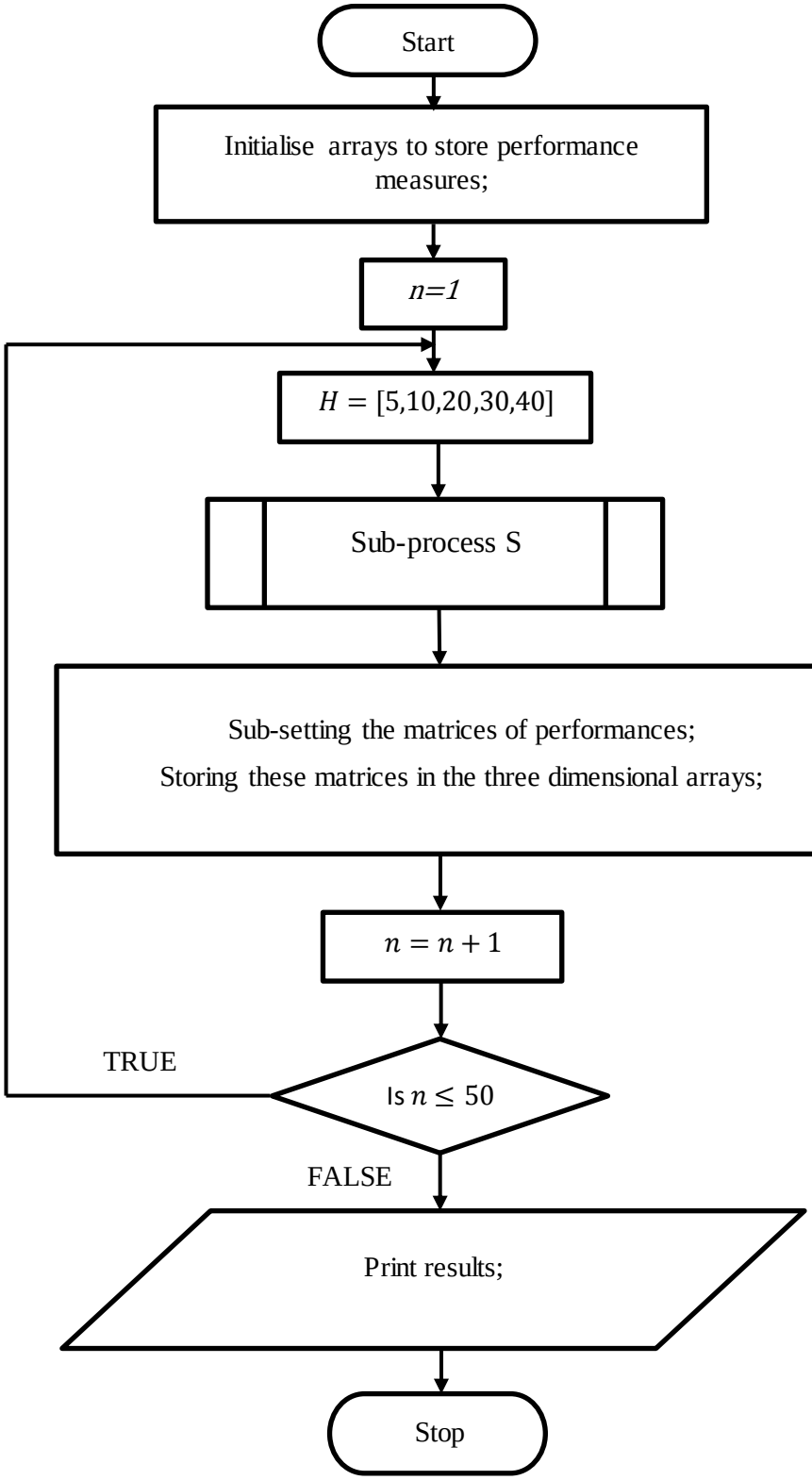

Figure 4.2: Step 1 (Process to get the performance measure estimates**)**

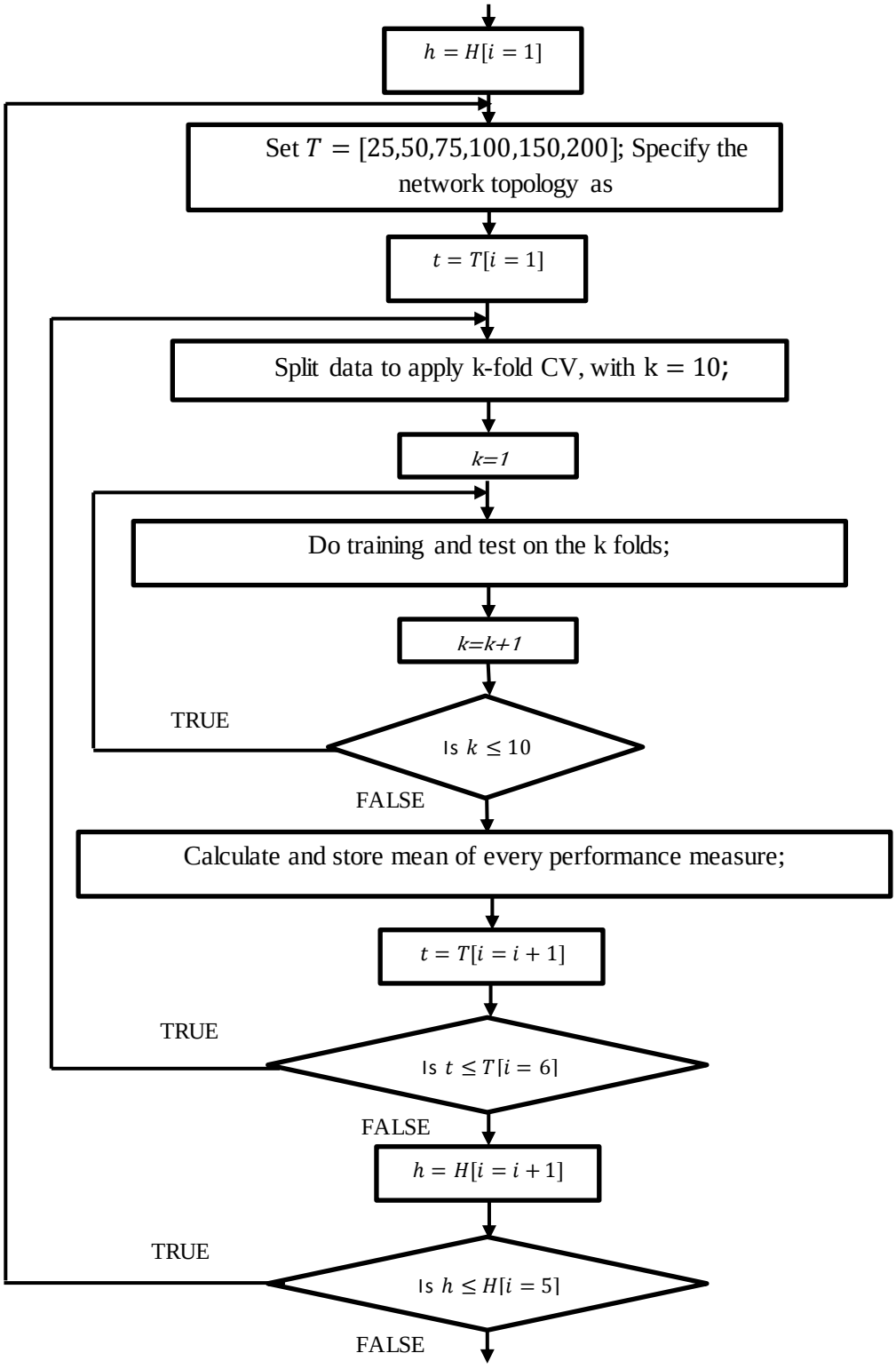

Figure 4.3: Sub-process S of Step 1

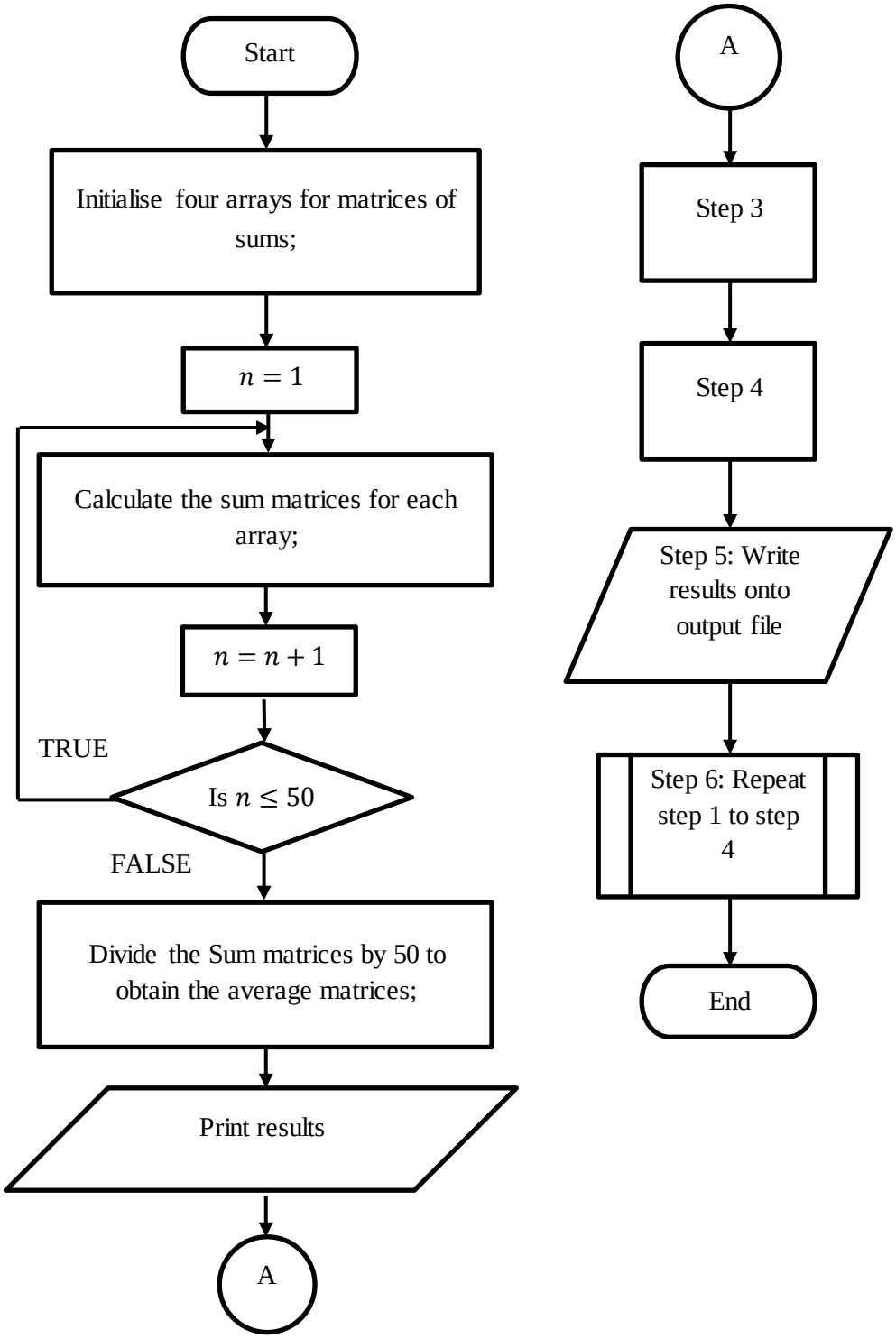

Figure 4.4: Steps 2 to 6 of process implementation

4.4 Software

The software used for the experiments is the Matlab Version "8.1.0.604 (R2013a)" (2013) whose NNs toolbox contains both the Rprop and CGP training algorithms as well as all the necessary functions for configuring and training the NN classifiers. All the graphs were created using the R ggplot2 package (Wickham, 2011). The computer used to perform these experiments is windows 7 (32-bit) operating system with an Intel® Celetron® CPU B815 @ 1.60GHz processor and 2 GB RAM. The reason for using this specific computer was that, it was the only computer that was accessible for 24 hours a day, and could facilitate the implementation of our experiments without disruption from any other users. The code created to implement all the training tasks can be found in the Appendix.

4.5 Summary of the chapter

The E.coli and Yeast problems (Murphy and Aha, 1994) are multiclass classification problems involving the classification of protein localisation patterns into eight classes for the E.coli and ten classes for the Yeast. It has been shown that "single machine" approaches that use single multiclass classifiers or approaches that use the combination of several binary classifiers can be applied to solve multiclass classification problems (Erin, Robert and Yoran, 2000; Crammer and Singer, 2001; Hsu and Lin, 2002) such as the E.coli and Yeast problems. Also, it has been argued that approaches that combine several binary classifiers to solve multiclass classification problems are less complicated to implement, have simple conceptual justification; and when well-tuned, they can be at least as accurate as single multiclass classifiers approaches. The several binary classifiers combination based approaches are therefore more appealing for practical purposes (Rifkin and Klautau, 2004).

For our experiments, among the two broadly applied several binary classifiers combination based approaches, the OAA approach have been chosen over the OAO

approach, because the OAA approach produces less binary classifiers (which are equal to the number of classes *k* of the multiclass classification problem) (Erin *et al.*, 2000; Rifkin and Klautau, 2004). Based on the OAA approach, the E.coli and Yeast classification problems will produce eight and ten binary classifiers, respectively. For our experiments, a binary classifier has been defined as a fully connected feedforward NN with an input layer, a hidden layer and an output layer. A brief summary of the training scenarios implemented and steps involved in the selection of the best binary classifier for each training scenario is as follows:

- For each binary classification task, a network with one hidden layer with various combinations of hidden nodes, i.e. 5, 10, 20, 30, 40 hidden nodes, and various combinations of maximum number of weights updates (maximum number of iterations), i.e. 25, 50, 75, 100, 150, 200 was trained. This was done for both Rprop and CGP algorithms.
- The performance measures, i.e. OAtrain, OAtest, MSE, and time were observed and recorded for each combination of variable parameters (hidden nodes and number of iterations). This gives 30 records for each performance measure.
- The best OAtrain (the maximum of the 30 accuracies on training set) reported, gives the best classifier on OAtrain for this binary classification task; the best OAtest (the maximum of the 30 accuracies on test set) reported, gives the best classifier on OAtest set; the best MSE (the minimum of the 30 MSEs) determines the best classifier on MSE; and the best time (the minimum of the 30 times) determines the best classifier on time for the same binary classification task.

In a nutshell, the best classifier for a particular binary classification task (with a given experimental design) is the one with the best performance on a particular performance measure.

Note that given the unbalanced nature of the E.coli (see section 4.2.1) and Yeast (see section 4.2.2) data analysed in this study, the classes with less than 10 cases were not

considered in the experiments. The following binary classifiers were therefore removed from our experiments: $imS/\sim imS$, $imL/\sim imL$ and $omL/\sim omL$ (see Table 4.3), and $ERL/\sim ERL$ (see Table 4.4). During training, when splitting the data into train/test sets, it is necessary to have at least one case in the training set and one case in the test set. Bearing in mind that the 10-fold cross validation approach (Kohavi, 1995) will be applied to generalise our classifiers, classes with less than 10 cases will not be represented in all the 10 different folds (training sets and test sets) and as a result, proper training and testing of the classifiers for these classes will be impossible. For this reason, 5 instead of 8 binary classifiers were trained for the E.coli data, and 9 instead of 10 binary classifiers were trained for the Yeast data.

CHAPTER 5: ANALYSIS AND RESULTS

5.1 Introduction

The main objective of this study was to investigate the optimisation methods of NNs training. The focus was on the capability of two NN training algorithms, i.e. Rprop and CGP. This was empirically done by evaluating and comparing their performances in classifying the E.coli and Yeast datasets. The performance measures being investigated here were the overall accuracy on the training set (OAtrain), the overall accuracy on the test set (OAtest), the level of convergence (or MSE) achieved, and the efficiency (the time required to reach the level of convergence achieved during training). Note that based on our experimental design (see Section 4.3), the level of convergence achieved during training is not necessarily the minimum of the network error function. The level of convergence is the MSE value achieved when training stopped. Since maximum numbers of iterations were specified (see justification in Section 4.3.2), training could have stopped in the two following cases: 1) at the minimum (which is 0) of the MSE function if this is achieved before the maximum iteration is completed; 2) after the maximum iteration is completed if the minimum of the MSE function is not reached before. Therefore, efficiency in our experiments refers to the time recorded when training stopped and not necessarily when convergence occurred.

Based on the performance results between the Rprop and CGP algorithms, the best algorithm was selected for further investigation on 1) the trade-off between training accuracy and test accuracy, 2) the trade-off between convergence and classification accuracy, and 3) the effect of varying the number of hidden nodes and number of training iterations on the performance of a classifier. In this chapter, results obtained from our experiments are presented and discussed. These results consist of the best performance measures for both Rprop and CGP, which gave the best classifier corresponding to each performance measure for each binary classification task. Precisely, the best classifier was obtained as outlined in Chapter 4.

5.2 Comparing Rprop and CGP Using the E.coli Proteins

The E.coli data set consists of 8 different classes. Based on the reasons highlighted in Section 4.5, we only analysed 5 instead of 8 binary classification tasks for the E.coli data. The 5 binary classification tasks were conducted following the previously described procedure. The best performance results for the 5 E.coli binary classifiers trained are presented in the following sections. The network configurations that produced the best results for each E.coli binary classifier are provided in Table 5.1. The notation I-H-O in this table stands for network with input nodes (I), hidden nodes (H) and output nodes (O). For the $cp/\sim cp$ for instance, the best network configuration was 7-5-2.

Table 5.1: The network configurations that produced the best E.coli binary classifiers

Binary classifiers	Network configurations (I-H-O)
$cp/\sim cp$	7-5-2
$im/\sim im$	7-10-2
$pp/\sim pp$	7-40-2
$imU/\sim imU$	7-20-2
$om/\sim om$	7-30-2

5.2.1 Accuracy comparison

Table 5.2 gives the results of the best OAtrain and OAtest for the 5 E.coli binary classifiers trained and for both Rprop and CGP. The results for Rprop are shown in column 1, and the results for CGP are in column 2 for both OAtrain and OAtest. The differences between Rprop and CGP for both OAtrain and OAtest are shown in column 3 and labelled as DOAtrain and DOAtest, respectively. The results in Table 5.2 show that in general, there were differences in the overall accuracies of the

classifiers for Rprop and CGP. It is evident that Rprop outperformed CGP for all the 5 binary classifiers and for both overall accuracies. On average, the Rprop achieved an OAtrain of 98.72% with a standard deviation (stdv) of 1.06%, while CGP produced an OAtrain of 95.62% with a stdv of 1.36%. It is worth noticing that Rprop achieved the highest OAtrain of 99.91% for the $om/\sim om$ binary classifier, while the highest OAtrain for CGP was 98.19% for the $om/\sim om$ binary classifier.

With respect to the OAtest, Rprop outperformed CGP. Their respective average accuracies were 94.84% with stdv of 2.62% and 92.55% with stdv of 2.71. It is also important to highlight that, as in the case of OAtrain, the highest OAtest for both training algorithms was achieved by $om/\sim om$, i.e. 98.4% for Rprop and 96.96% for CGP.

Table 5.2: E.coli Best OAtrain and Best OAtest

Binary classifiers	OAtrain (%)			OAtest (%)		
	Rprop	CGP	DOAtrain	Rprop	CGP	DOAtest
$cp/\sim cp$	99.29	95.43	3.86	96.85	93.28	3.57
$im/\sim im$	97.59	94.24	3.35	91.2	88.58	2.62
$pp/\sim pp$	99.51	95.38	4.13	95.01	92.26	2.75
$imU/\sim imU$	97.31	94.86	2.45	92.75	91.67	1.08
$om/\sim om$	99.91	98.19	1.72	98.4	96.96	1.44
Mean	98.72	95.62	3.1	94.84	92.55	2.29
Stdv	1.06	1.36	0.9	2.62	2.71	0.91

The differences of performance between Rprop and CGP based on OAtrain and OAtest are highlighted in Table 5.2 and the results show that the accuracy differences between Rprop and CGP were smaller on the test set than they were on the training set. On average, the DOtest was of 2.29%, while the DOAtrain was 3.1%. Moreover, the highest difference between the two algorithms for the training set was 4.13%, which was obtained with the $pp/\sim pp$ classifier, whereas the highest

difference for the test set was 3.57%, which was obtained with the $cp/\sim cp$ classifier. Table 5.2 also shows that the smallest difference on training set (i.e. 1.72%) between the two algorithms was obtained with $om/\sim imL$, whereas the smallest difference on test set (i.e. 1.08%) was obtained with $imU/\sim imU$.

To conclude in brief, accuracy comparison between Rprop and CGP algorithms has revealed that Rprop outperformed CGP for both OAtrain and OAtest. This was true for all the 5 E.coli binary classifiers considered in our experiments. Moreover, it appears that the accuracy differences between Rprop and CGP varied with regard to the various E.coli binary classifiers. For some binary classifiers such as $cp/\sim cp$ and $pp/\sim pp$, the difference on the training set was at least 3.86% and the difference on the test set was at most 3.57%. For others such as $imU/\sim imU$ and $om/\sim om$, the difference on the training set was at least 1.72% and the difference on the test set was at most 1.44%. The change in accuracy differences between the various E.coli binary classifiers may have been due to the differences in the class structures of the data that had to be classified by the various E.coli binary classifiers; the classifiers with differences of at least 3.86% may have been more complex to train, because they may have been dealing with more complex class structure, compared to the classifiers with lower differences. Further analysis of the differences in class structures and their effects on the various classifiers is presented in Sections 5.6 and 5.7.

5.2.2 Convergence comparison

Convergence performance of a NN training algorithm is the ability of the algorithm to reach the minimum of the error function, or else, the target error of the network under consideration. For our experiments, no error target was specified. The only stopping criterion was the different maximum numbers of weights updates (maximum number of training iterations). Therefore, the measure of convergence considered in this study was the minimum MSE value reached by each binary

classifier when training stopped (Section 5.1 describes the cases where training could have stopped).

Table 5.3 gives the best MSEs reached by the 5 E.coli binary classifiers considered in our experiments for both Rprop and CGP. The results in Table 5.3 show that, in general, Rprop outperformed CGP for the 5 E.coli binary classifiers. On average, Rprop achieved the smallest MSE of 0.0113, with a standard deviation of 0.0088, whereas CGP achieved the MSE of 0.0371, with a standard deviation of 0.0118. The smallest MSE achieved for Rprop was 0.0007, while the smallest MSE achieved for CGP was 0.0146. Both MSEs were achieved when training the $om/\sim om$ classifier.

Table 5.3: E.coli Best MSE

Binary classifiers	MSE	
	Rprop	CGP
$cp/\sim cp$	0.0071	0.0462
$im/\sim im$	0.0217	0.0469
$pp/\sim pp$	0.0052	0.0378
$imU/\sim imU$	0.022	0.0402
$om/\sim om$	0.0007	0.0146
Mean	0.0113	0.0371
Stdv	0.0088	0.0118

5.2.3 Efficiency comparison

The efficiency of a NN training algorithm is the measurement of how fast that algorithm converges. In other words, the efficiency of a NN training algorithm is an estimation of the time required by that algorithm to reach the minimum error or the error target during training. Efficiency was measured in our experiments, by the time taken during training, to reach the smallest recorded MSE's values for each binary classifier. For a fair comparison between the Rprop and CGP algorithms based on

their efficiency, we used the same error targets for both algorithms. Given that CGP yielded the biggest (the farthest from the minimum 0) MSEs, we assumed that the MSEs of CGP would be the easiest targets to reach for both algorithms. Therefore, we measured efficiency for both algorithms, by the time taken to reach the smallest MSEs achieved by CGP during training. Table 5.4 provides the time taken to reach the MSEs for CGP given in Table 5.3, for the 5 E.coli binary classifiers. The results in Table 5.4 show that Rprop took less time than CGP to converge to the specified MSE's values for the E.coli binary classifiers. On average, Rprop reached the target in 244 seconds, whereas CGP did so in 1133 seconds. The longest time for Rprop (i.e. 520 seconds) was achieved by $imU/{\sim}imU$, while that for CGP (i.e. 1847 seconds) was achieved by $im/{\sim}im$. The shortest time for Rprop (i.e. 18 seconds) was achieved by $om/{\sim}om$, while that for CGP (i.e. 136 seconds) was also achieved by $om/{\sim}om$.

Table 5.4: E.coli Training time to best MSE

Binary classifiers	Time (seconds)	
	Rprop	CGP
$cp/{\sim}cp$	76	1009
$im/{\sim}im$	395	1847
$pp/{\sim}pp$	213	1295
$imU/{\sim}imU$	520	1378
$om/{\sim}om$	18	136
Mean	244	1133
Stdv	189	567

5.2.4 Concluding remarks

The results for the E.coli data have shown that Rprop algorithm performed better than CGP. Comparison of the two algorithms was done based on 4 measures of performance, i.e. OAtrain, OAtest, convergence, and efficiency. Based on these 4

measures of performance, Rprop outperformed CGP for all the 5 E.coli binary classifiers analysed in our experiments. We can therefore conclude based on the the performance results for the E.coli binary classifiers that: 1) Rprop yielded more accurate results than CGP, 2) Rprop had better convergence capabilities than CGP, and 3) Rprop was more efficient and converged faster than CGP.

It is worth highlighting that the differences in accuracy between Rprop and CGP were very small (less than 2.5%) for some classifiers such as $imU/{\sim}imU$ and $om/{\sim}om$. In view of that, two questions could arise. Was this difference big enough to suggest that Rprop was better than CGP? Would it still be worthwhile to use CGP? The answer to these questions would depend on the application problems under study. For biological data such as the E.coli, Yeast, diabetes and cancer (Murphy and Aha, 1994), even the smallest difference could be of major importance. This difference could even be of greater importance if one was dealing with a larger dataset; because lesser accuracy would mean more misclassifications. For instance, if one had to classify 5000 different sites based on the presence or not of some E.coli proteins, 1% lesser accuracy would amount to 50 more misclassifications of sites. This would mean that 50 sites could be declared clean, while contaminated by some E.coli proteins (bacteria); which could result in many diseases.

Furthermore, when analysing simultaneously the accuracy and efficiency of both training algorithms, we observed that Rprop was on average 4.6 (1133 second/244 seconds) times more efficient than CGP. This means that Rprop was able to achieve very good accuracies in very short training times when compared to CGP. We can therefore suggest that Rprop was better than CGP for the E.coli binary classifiers. Based on this conclusion, Rprop was selected for further investigations on the behaviour and trade-off between the different performance measures, and the effect of varying the number of hidden nodes and the maximum number of training iterations. Results of these investigations are presented in the following sections.

5.3 Effect of hidden nodes and training iterations on E.coli classifiers using Rprop

The Rprop was selected for further investigation as it yielded better results compared to the CGP. In this section, we investigated the effect of the number of hidden nodes and the maximum number of training iterations on a classifier using the E.coli data set. Note that the various E.coli binary classifiers would probably exhibit different behaviours from one another, with respect to varying the numbers of hidden nodes and iterations. Since the aim of our experiments was to show how varying the numbers of hidden nodes and training iterations can play a role in the performance of a classifier, regardless of which classifier is used, the use of one single classifier was deemed good enough to illustrate this concept (varying the numbers of hidden nodes and training iterations). We therefore restricted our experiments to only one E.coli binary classifier (i.e. $cp/\sim cp$), instead of all the 5 E.coli binary classifiers, though each binary classifier was assumed to be of different complexity. Note that the choice of the $cp/\sim cp$ classifier was subjective. A different classifier could have been utilised to illustrate the above defined concept; and the results, though different, would still have been as informative as the ones presented in this section.

The evaluation of the effect on the above selected E.coli binary classifier, for varying the numbers of hidden nodes and training iterations, was done with respect to all the previously described performance measures, i.e. OAtrain, OAtest, convergence (or MSE achieved) and efficiency (or training time). The same training scenarios as the ones described in Section 4.5 were applied for the experiments in this section (i.e. networks with the following numbers of hidden nodes and training iterations: 5, 10, 20, 30, and 40 hidden nodes, and 25, 50, 75, 100, 150, and 200 training iterations). Different training scenarios could have been applied and the idea of varying the numbers of hidden nodes and iterations would still have been successfully illustrated, as long as the numbers of hidden nodes and iterations were kept below 40 and 200 respectively. Based on the accuracies obtained in previous experiments (see Table 5.2), we suspected that training scenarios for varying the numbers of hidden nodes and iterations that are greater than 40 hidden nodes and 200 iterations, would not have properly exhibited the changes in the performance of the E.coli classifiers. This

was because the accuracies obtained for the training scenarios defined in Section 4.5, were already very high, with eventually no room for improvement for further training.

5.3.1 Effect on convergence

The patterns in Figure 5.1 portray the effect of changing the number of hidden nodes and the pre-specified number of training iterations on the convergence of the $cp/\sim cp$ binary classifier. Figure 5.1 shows that the MSEs decreased when the number of hidden nodes was increased. For instance, the MSEs at 5 hidden nodes were larger than the MSEs at 10 hidden nodes. This was true for the various numbers of hidden nodes. The same negative relationship was observed between the MSEs and the various numbers of training iterations.

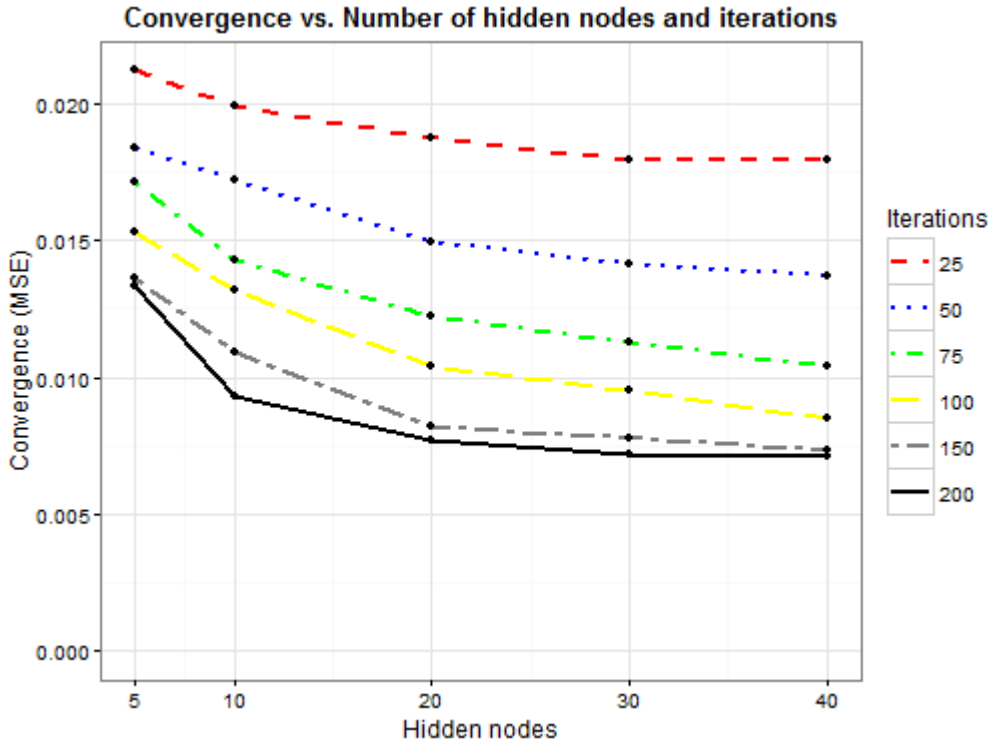

Figure 5.1: The MSE for varying the number of hidden nodes and training iterations for the cp/~cp binary classifier trained with Rprop

The MSEs decreased with an increase in the training updates (iterations). The lines for higher training iterations were situated at the smaller MSEs than the lines for smaller training iterations. For instance, the line for 200 training iterations was at the bottom and the line for 25 training iterations was at the top of all lines. Since the patterns in Figure 5.1 suggest that training convergence (or MSEs) improved with an increase in the hidden nodes and training iterations for the $cp/\sim cp$ classifier, we can therefore suggest that varying the numbers of hidden nodes and iterations may impact the convergence of NNs classifiers. The impact would probably vary with respect to various classifiers.

5.3.2 Effect on the accuracy on training set

The patterns in Figure 5.2 show the effect on the accuracy on the training set as the numbers of hidden nodes and maximum training iterations were varied for the

$cp/\sim cp$ classifier. The results in Figure 5.2 show a positive relationship between the numbers of hidden nodes and training iterations with the OAtrain. For an increase in the hidden nodes, there was an increase in the OAtrain. For instance, the OAtrain increased when the number of hidden nodes was increased from 20 to 30. For an increase in the number of training iterations, there was an increase in the OAtrain. For example, OAtrain for 200 training iterations were the highest, while OAtrain for 25 training iterations were the lowest. Based on the patterns in Figure 5.2, we can therefore say that increasing the numbers of hidden nodes and training iterations may have a positive effect on the OAtrain; this effect would probably vary with respect to the classifiers under consideration. Although this effect was rarely greater than 2% for the OAtrain, it could still be of great importance, depending on the application problems as explained in Section 5.2.4.

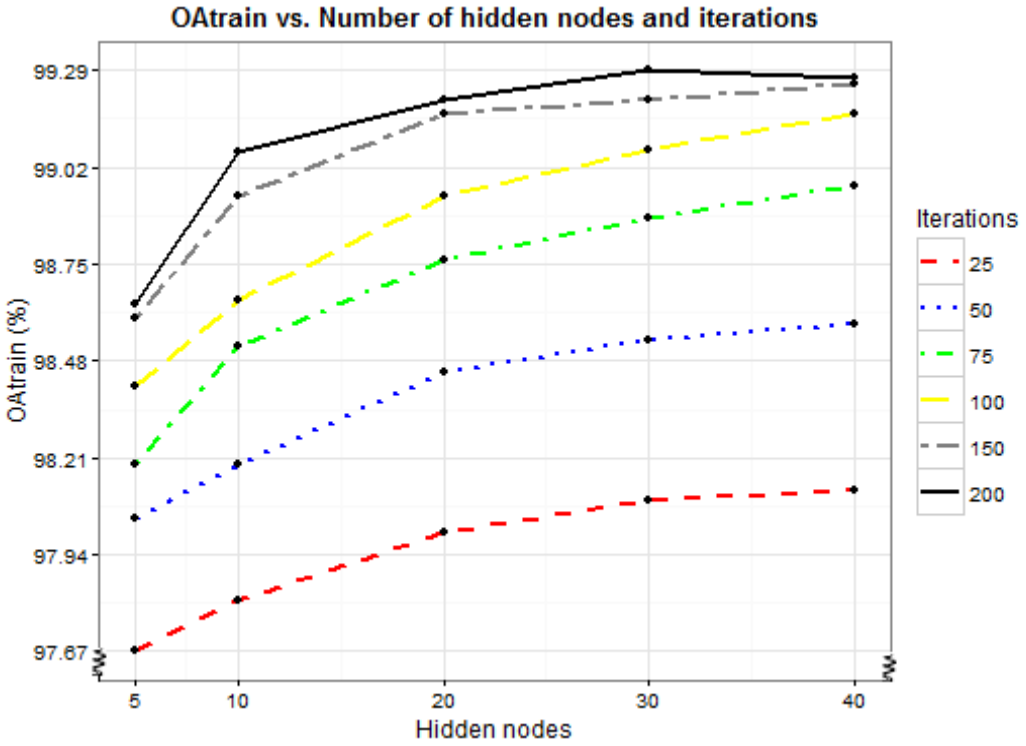

Figure 5.2: The OAtrain for varying number of hidden nodes and training iterations for the cp/~cp binary classifier trained with Rprop

5.3.3 Effect on the accuracy on test set

Figure 5.3 depicts the effect on the accuracy on the test set as the numbers of hidden nodes and training iterations were varied for the $cp/\sim cp$ classifier. The behaviour of the OAtest based on the change in the numbers of hidden nodes and iterations portrayed in Figure 5.3, does not exhibit a consistent pattern as the one for the OAtrain in Figure 5.2. Figure 5.3 shows that there was no constant trend in the change of OAtest as the numbers of hidden nodes and training iterations were varied. However, it appears that better solutions were found for networks that were trained for smaller numbers of iterations, i.e. 25 and 50. The superiority of the solutions found with 25 and 50 training iterations can be observed through the various numbers of hidden nodes in Figure 5.3. It is evident that a network with 5 hidden nodes trained for 25 iterations yielded the best OAtest. This accuracy decreased with an increase in the number of training iterations (interpretation was done vertically

from the top to the bottom for 5 hidden nodes). Note that the decrease in accuracy may seem not that much, but could have major implications in some application fields as explained in Section 5.2.4. Figure 5.3 also suggest that training that was done for more than 25 iterations (for 5, 10 and 40 hidden nodes) and 50 iterations (for 20 and 30 hidden nodes) over-fitted the training set, because beyond 25 and 50 iterations, the OAtest decreased as shown in Figure 5.3. Also, for 20 hidden nodes, the network trained for 200 iterations outperformed the one trained for 150 iterations. For 30 hidden nodes, the network trained for 200 iterations outperformed those trained for 100, and 150 iterations.

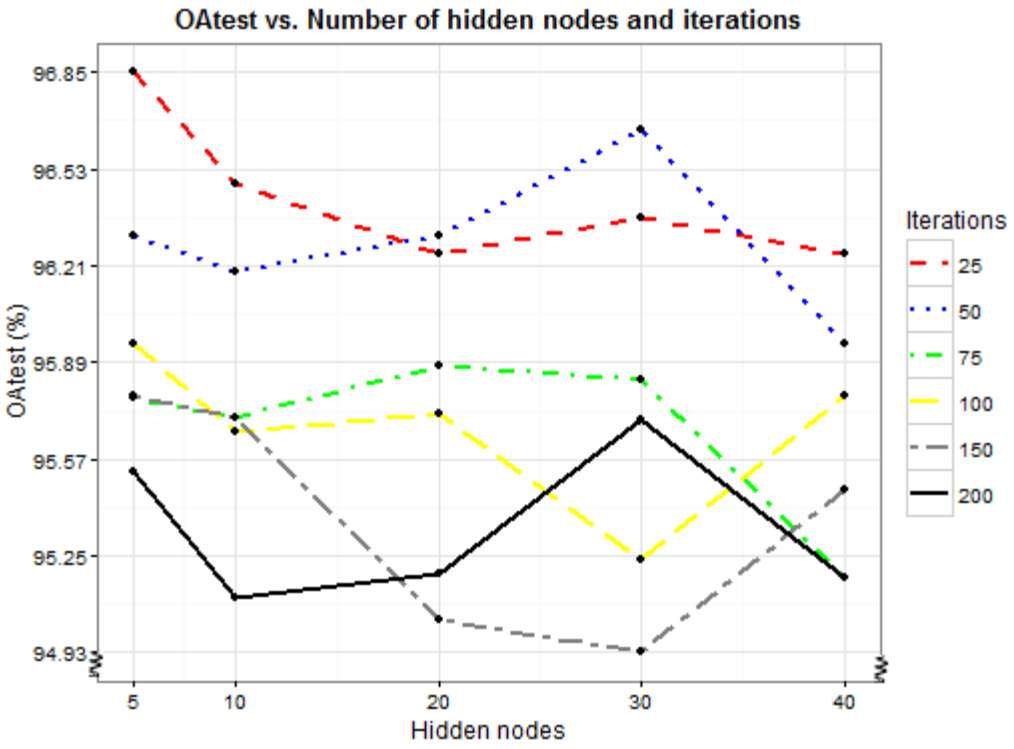

Figure 5.3: The OAtest for varying the number of hidden nodes and training iterations for the cp/~cp binary classifier trained with Rprop

Observation of the patterns in Figure 5.3 may also lead to the suggestion that an excessive increase of the number of hidden nodes may badly affect the performance accuracy of a classifier on unseen dataset (test set). Take for instance the decrease in

OAtest, as the number of hidden nodes was increased, for the classifier trained with 25 iterations (dashed red line). This is because a network with excessive number of parameters in the weights space (consequence of excessive number of hidden nodes) may start generating arbitrary complex regions in the weights space and end up over-fitting the data, if the number of iterations is not well monitored. Hence, finding of optimal number of hidden nodes should be coupled with the finding of optimal number of training iterations. For instance, the results in Figure 5.3 show that, networks with 5, 10, and 40 hidden nodes yielded better solutions when trained for 25 iterations than when they were trained for 50 iterations, whereas networks with 20 and 30 hidden nodes yielded better solutions when trained for 50 iterations than when they were trained for 25 iterations.

5.3.4 Effect on efficiency

Figure 5.4 shows the effect of varying the numbers of hidden nodes and training iterations on the efficiency of the $cp/\sim cp$ classifier. The patterns in Figure 5.4 portray a positive relationship between the time required during training and the number of hidden nodes. Networks with larger number of hidden nodes seem to have required more training time compared to those that have less. However, the positive relationship between the number of hidden nodes and training time was not that pronounced. The patterns in Figure 5.4 show that there was an increase in the training time as the number of hidden nodes was increased. But the increase in the training time was rather small, especially in the range between 5 and 30 hidden nodes, for networks trained for 25 to 100 iterations. A more pronounced increase in the training time happened when the number of hidden nodes was increased from 30 to 40. For the effect on efficiency, based on the various numbers of training iterations, the patterns in Figure 5.4 suggest that the training time increased as the number of iterations was increased. For instance, Figure 5.4 shows that the training times of networks trained with 25 iterations were less than 50 seconds, whereas the training times of networks trained with 200 iterations were greater than 600 seconds.

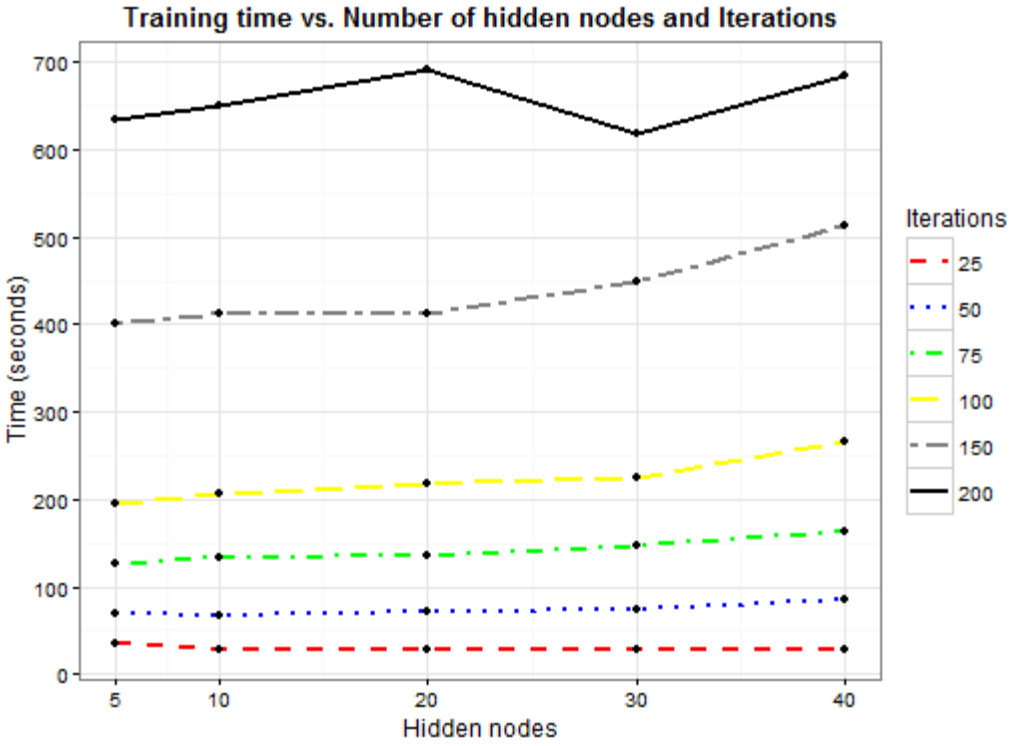

Figure 5.4: Training time (in seconds) for varying number of hidden nodes and training iterations for the *cp/~cp* binary classifier

5.4 Comparing Rprop and CGP using the Yeast Proteins

For the E.coli data, the results showed that the differences in accuracy between Rprop and CGP were rarely greater than 2% for some classifiers, even if the superiority of Rprop was evident when both accuracy and efficiency were used simultaneously in the comparison of the two training algorithms. In this section, we used the Yeast data, which is of different composition from the E.coli data, to see if there would be more differences in accuracy between Rprop and CGP, when using data with different class structures. The Yeast data consists of 10 different classes. However, due to the very small cases (i.e. 5) contained in the *ERL* class, and given that the 10-fold cross validation was utilised to generalise the various NN classifiers, which would have made the training of the $ERL/\sim ERL$ classifier practically impossible, 9 Yeast binary classifiers, instead of 10 Yeast binary classifiers, were considered in our experiments. The same training scenarios as the ones described in

Section 4.5 were also applied here. The performance results for the 9 Yeast binary classifiers are presented in the following sections. The network configurations that produced the best results for each Yeast binary classifier are provided in Table 5.5. For the $CYT/\sim CYT$ for instance, the best network configuration was 8-40-2.

Table 5.5: The network configurations that produced the best Yeast binary classifiers

Binary classifiers	Network configurations (I-H-O)
$CYT/\sim CYT$	8-40-2
$NUC/\sim NUC$	8-30-2
$MIT/\sim MIT$	8-10-2
$ME3/\sim ME3$	8-10-2
$ME2/\sim ME2$	8-20-2
$ME1/\sim ME1$	8-20-2
$EXC/\sim EXC$	8-20-2
$VAC/\sim VAC$	8-5-2
$POX/\sim POX$	8-10-2

5.4.1 Accuracy comparison

Table 5.6 provides the results of the best OAtrain and OAtest for the 9 Yeast binary classifiers, and for both Rprop and CGP. Generally, the results in Table 5.6 suggest that Rprop outperformed CGP for both accuracies on training and test sets. On average, Rprop achieved an OAtrain of 93.54% with a standard deviation of 7.89%, while CGP produced an OAtrain of 91.75% with a standard deviation of 9.19%. The highest OAtrain (i.e. 99.47%) for Rprop was achieved with the $ME1/\sim ME1$ classifier, while the highest OAtrain (i.e. 99.26%) for CGP was achieved with the $POX/\sim POX$ classifier. The smallest OAtrain for both Rprop (i.e. 78.55%) and CGP (i.e. 74.22%) was produced by $CYT/\sim CYT$. For the OAtest, Table 5.6 shows that

Rprop outperformed CGP. On average, the OAtest for Rprop was 91.91% with a standard deviation of 8.87%, while the OAtest for CGP was 90.8% with a standard deviation of 9.65%. The highest OAtest for both Rprop (i.e. 99.04%) and CGP (i.e. 99.04%) were achieved with the $POX/\sim POX$ classifier, while the smallest OAtest for both Rprop (i.e. 74.53%) and CGP (i.e. 72.41%) were achieved by the $CYT/\sim CYT$ classifier. The differences between Rprop and CGP based on training accuracy and test accuracy are respectively shown in the columns labelled DOAtrain and DOAtest in Table 5.6. These differences were smaller on test sets (i.e. in the range between 0% and 2.52%) than they were on training sets (i.e. in the range between 0.14% and 4.33%). On average, the accuracy difference between Rprop and CGP was 1.79% on training set and 1.12% on test set.

Table 5.6: Yeast Best OAtrain and Best OAtest

Binary classifiers	OAtrain (%)			OAtest (%)		
	Rprop	CGP	DOAtrain	Rprop	CGP	DOAtest
$CYT/\sim CYT$	78.55	74.22	4.33	74.53	72.41	2.12
$NUC/\sim NUC$	80.32	77.06	3.26	77.95	75.7	2.25
$MIT/\sim MIT$	91.53	88.6	2.93	89.12	86.93	2.19
$ME3/\sim ME3$	96.8	93.85	2.95	95.27	92.75	2.52
$ME2/\sim ME2$	98.34	97.63	0.71	96.86	96.76	0.1
$ME1/\sim ME1$	99.47	98.63	0.84	98.3	97.74	0.56
$EXC/\sim EXC$	99.03	98.29	0.74	98.17	97.86	0.31
$VAC/\sim VAC$	98.42	98.17	0.25	97.98	97.98	0
$POX/\sim POX$	99.4	99.26	0.14	99.04	99.04	0
Mean	93.54	91.75	1.79	91.91	90.8	1.12
Stdv	7.89	9.19	1.47	8.87	9.65	1.05

In summary, the comparison of accuracies between the Rprop and CGP algorithms using the 9 Yeast binary classifiers considered in our experiments has revealed that

Rprop outperformed CGP both on training and test sets. However, for some Yeast binary classifiers, the difference between the two training algorithms was very small, whereas for some others the difference was relatively high. For instance, the differences between the two algorithms were amongst the smallest (i.e. less than 1%) for classifiers such as $POX/\sim POX$, $VAC/\sim VAC$, $EXC/\sim EXC$, $ME1/\sim ME1$ and $ME2/\sim ME2$, whereas they were amongst the largest (i.e. more than 2.5%) for the classifiers such as $CYT/\sim CYT$, $NUC/\sim NUC$, $MIT/\sim MIT$ and $MIT/\sim MIT$.

As was done for the E.coli binary classifiers, the reason of the differences in accuracy between the various Yeast binary classifiers may have been due to the fact that the various binary classifiers may have been dealing with data having different class structures. Some class structures may have been more complex to train than some others. In this regard, the classifiers dealing with more complex class structures would be less accurate than the classifiers dealing with less complex class structure. Besides, when compared to the E.coli binary classifiers in Table 5.2, the Yeast binary classifiers appeared to have lower classification accuracies in general. The accuracies of the E.coli binary classifiers started from 90%, while those for the Yeast binary classifiers started from less than 75%. This implies that the Yeast binary classifiers may have been more complex than the E.coli binary classifiers. Further analysis of the difference in class structures and effect on the various classifiers is presented in Section 5.6 and 5.7.

5.4.2 Convergence comparison

Table 5.7 gives the best MSEs reached by the 9 Yeast binary classifiers considered in our experiments for both Rprop and CGP. The results in Table 5.7 show that Rprop outperformed CGP for all the 9 Yeast binary classifiers. The MSEs achieved by Rprop were smaller than those achieved by CGP. On average, Rprop achieved a MSE of 0.0479 with a standard deviation of 0.0541, whereas CGP achieved a MSE of 0.063 with a standard deviation of 0.0659. The best performance (or the smallest

MSE) achieved for Rprop was 0.0054, whereas the best performance achieved for CGP was 0.0072. For Rprop, the best performance was achieved when training the $ME1/\sim ME1$ classifier, while the best performance for CGP was achieved when training the $POX/\sim POX$ classifier. The poorest convergence performance for Rprop was 0.1471, while that for CGP was 0.1845. Both were achieved by the $CYT/\sim CYT$ classifier.

Table 5.7: Yeast Best MSE

Binary classifiers	MSE	
	Rprop	CGP
$CYT/\sim CYT$	0.1471	0.1845
$NUC/\sim NUC$	0.1391	0.1685
$MIT/\sim MIT$	0.0684	0.0937
$ME3/\sim ME3$	0.0266	0.0488
$ME2/\sim ME2$	0.015	0.0214
$ME1/\sim ME1$	0.0054	0.0107
$EXC/\sim EXC$	0.0085	0.0148
$VAC/\sim VAC$	0.0152	0.0172
$POX/\sim POX$	0.0058	0.0072
Mean	0.0479	0.063
Stdv	0.0541	0.0659

5.4.3 Efficiency comparison

As for the E.coli binary classifiers (see Section 5.2.3), we utilised the same error targets for both Rprop and CGP, for fair comparison of their efficiency in the training of the Yeast binary classifiers. Since CGP produced the largest MSEs (or the farthest from the minimum of the network error functions), we assumed that these would be the easiest targets to reach for both algorithms. Therefore, we measured efficiency by the time taken to reach the MSEs of CGP. Table 5.8 gives the time taken to reach the

MSEs of CGP provided in Table 5.7, for the 9 Yeast binary classifiers. The results in Table 5.8 suggest that Rprop was more efficient than CGP. Rprop took much shorter times than CGP to reach convergence for the 9 Yeast binary classifiers trained in this study. On average, Rprop reached the target in 1072 seconds, whereas CGP did so in 3741 seconds. The longest time for Rprop (i.e. 1981 seconds) was obtained in the training of the $VAC/\sim VAC$ classifier, while that for CGP (i.e. 7726 seconds) was obtained in the training of the $CYT/\sim CYT$ classifier. The shortest time for Rprop was 278 seconds, while that for CGP was 729 seconds. Both shortest times were achieved with the $POX/\sim POX$ classifier

Table 5.8: Yeast Training time to best MSE

Binary classifiers	Time (seconds)	
	Rprop	CGP
$CYT/\sim CYT$	1414	7726
$NUC/\sim NUC$	888	2797
$MIT/\sim MIT$	1219	3605
$ME3/\sim ME3$	838	2310
$ME2/\sim ME2$	1117	2606
$ME1/\sim ME1$	899	4471
$EXC/\sim EXC$	1014	3382
$VAC/\sim VAC$	1981	6046
$POX/\sim POX$	278	729
Mean	1072	3741
Stdv	437	1977

5.4.4 Concluding remarks

The results for the Yeast data have revealed that Rprop algorithm performed better than CGP. The two algorithms were compared based on four measures of performance, i.e. OAtrain, OAtest, convergence (or MSE achieved), and efficiency.

Based on these 4 measures of performance, Rprop outperformed CGP for all the 9 Yeast binary classifiers analysed in our experiments. We can therefore suggest based on the Yeast data that: 1) Rprop was more accurate than CGP, 2) Rprop had better convergence capabilities than CGP, and 3) Rprop was more efficient and converged faster than CGP.

We emphasise once more that the difference in accuracies between Rprop and CGP may seem not big enough to decide on the best algorithm between the two. But when accuracy and efficiency were simultaneously considered in the choice of the best training algorithms, Rprop appeared to be the best training algorithm. On average, Rprop was 3.5 (3741 seconds/1072 seconds) times more efficient than CGP. This means that to achieve comparable accuracies, Rprop required 3.5 times less training time than CGP. In this regard, we can suggest that Rprop was much better than CGP. Based on these findings, Rprop was hence selected for further investigations on the trade-off between the performance measures, and the effect of varying the numbers of hidden nodes and training iterations on the performance of the Yeast binary classifiers. Results of these investigations are presented in the following sections.

5.5 Effect of hidden nodes and training iterations on Yeast classifiers using Rprop

As for the E.coli data, the Rprop was selected for further investigation since it yielded better results compared to CGP for the Yeast data. In this section, we investigated the effect of the number of hidden nodes and the maximum number of training updates (iterations) on a classifier using the Yeast data set. Since the interest was to analyse the behaviour of a classifier based on the change in the number of hidden nodes and number of training iterations, the use of one single binary classifier was deemed good enough to illustrate this behaviour. We hence restricted our experiments to only one Yeast binary classifier, i.e. $CYT/{\sim}CYT$ instead of all the 9 Yeast binary classifiers analysed in the previous sections, though each Yeast binary classifier seemed to be of different complexity. It is worth emphasising that there

was no objective reason for choosing the $CYT/\sim CYT$ classifier rather than another Yeast binary classifier, to evaluate the behaviour of a NN classifier based on the change in the numbers of hidden nodes and training iterations. A different Yeast binary classifier could have been utilised for illustrative purpose of the behaviour of a NN classifier based on varying the numbers of hidden nodes and training iterations; and the results, though different, would still have been as informative as the ones presented in this section. In choosing only the $CYT/\sim CYT$ classifier, we do not imply that all the 9 Yeast binary classifiers will exhibit the same behaviour. These binary classifiers would probably behave differently given that they are of different complexity. However, we assumed that varying the numbers of hidden nodes and training iterations would impact the behaviour of these classifiers in one way or another. For the experiments in this section, we applied the same training scenarios as the ones applied for the E.coli binary classifiers in Section 5.3. The performance results are presented in the following sections.

5.5.1 Effect on convergence

The patterns in Figure 5.5 display the effect of changing the numbers of hidden nodes and the pre-specified training iterations on the convergence of the $CYT/\sim CYT$ binary classifier. Figure 5.5 portrays a negative relationship between the MSEs and the number of hidden nodes. The MSEs decreased as the number of hidden nodes was increased. For instance, the MSEs decreased as the hidden nodes were increased from 5 to 10, 10 to 20, 20 to 30, and 30 to 40. Figure 5.5 also exhibits a negative relationship between the MSEs and the various numbers of training iterations. The MSEs decreased as the training iterations were increased. This can be observed on Figure 5.5 where the lines corresponding to higher training iterations were situated at the smaller MSEs than the lines corresponding to smaller training iterations. For instance, the line corresponding to the highest (i.e. 200t) training iterations was situated at the bottom of all the lines, where the MSE's values were the smallest, while the line corresponding to the smallest (i.e. 25t) training iterations was situated at the top of all the lines, where the MSE's values were the highest. Based on the

behaviour of the $CYT/{\sim}CYT$ classifier portrayed in Figure 5.5, we can suggest that the convergence of a NN classifier can be improved with an increase in the hidden nodes and training iterations.

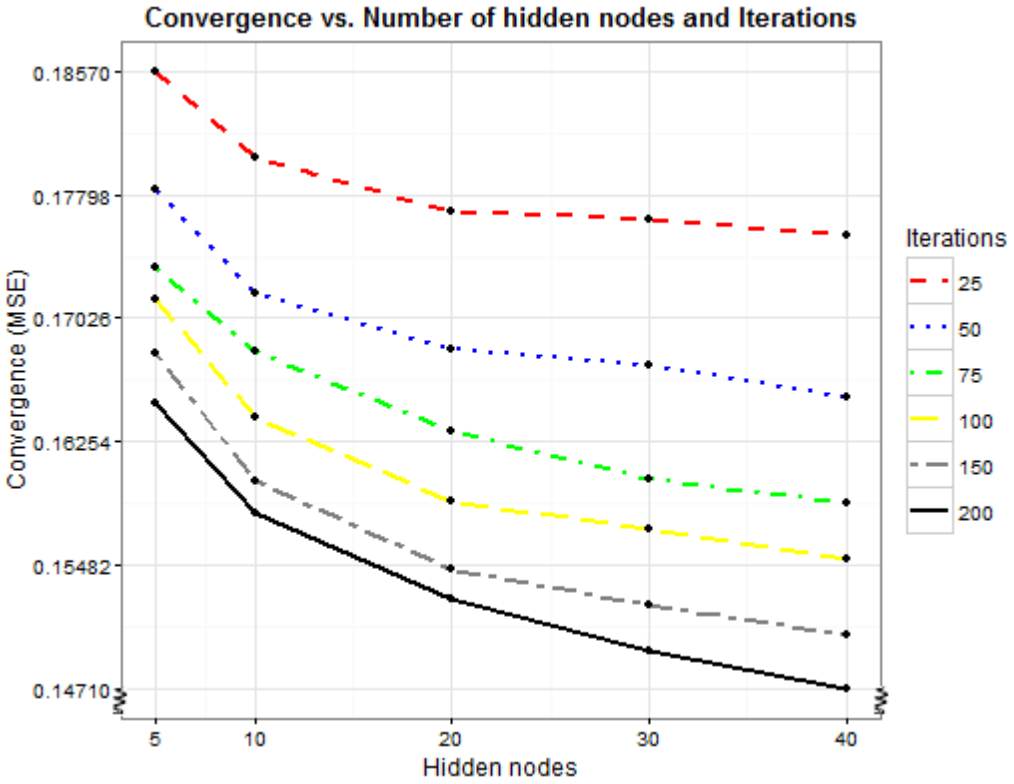

Figure 5.5: The MSE for varying number of hidden nodes and training iterations for the CYT/~CYT binary classifier trained with Rprop

5.5.2 Effect on the accuracy on training set

Figure 5.6 portrays the effect on the accuracy on training (i.e. OAtrain) set as the numbers of hidden nodes and training iterations were varied for the $CYT/{\sim}CYT$ binary classifier. The patterns in Figure 5.6 suggest a positive relationship between the numbers of hidden nodes and training iterations with the OAtrain, which increased as the hidden nodes and training iterations were increased. For the hidden

nodes for instance, Figure 5.6 shows that the OAtrain constantly increased as the hidden nodes were increased from 5 to 10, 10 to 20, 20 to 30, and 30 to 40. Figure 5.6 also shows that the lines corresponding to higher training iteration were situated at the higher OAtrain than the lines corresponding to the smaller training iterations, suggesting that the accuracy on training set increased as the number of training iterations was increased. For instance, the OAtrain for 200 training iterations were the highest, while the OAtrain for 25 training iterations were the smallest as shown in Figure 5.6.

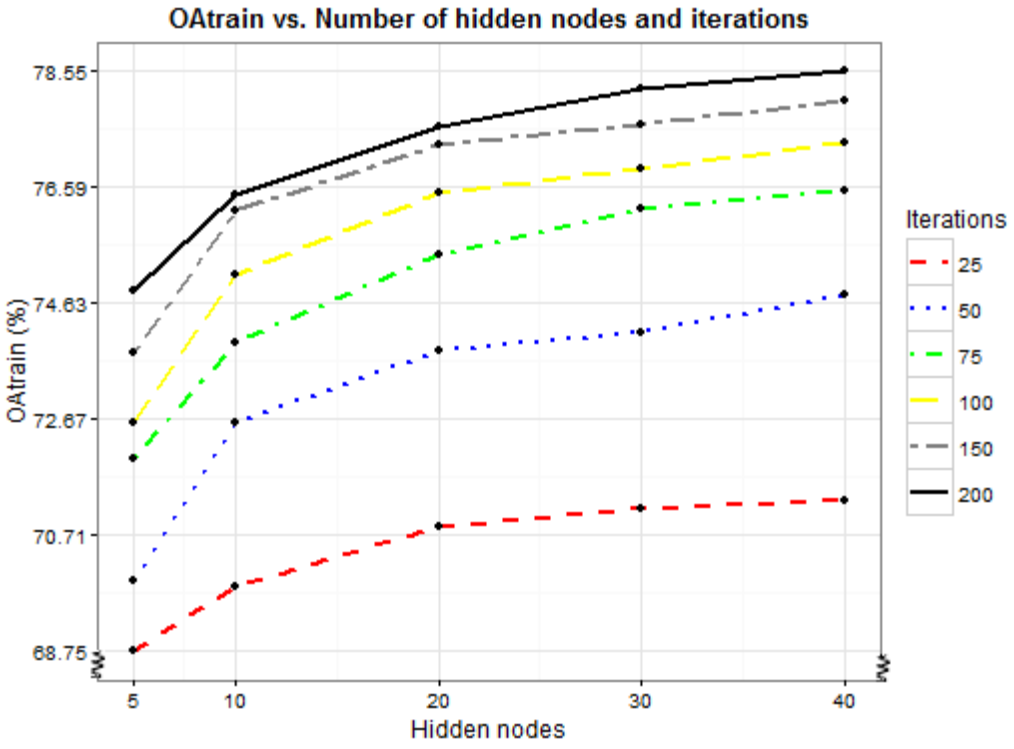

Figure 5.6: The OAtrain for varying number of hidden nodes and training iterations for the CYT/~CYT binary classifier trained with Rprop

5.5.3 Effect on the accuracy on test set

Figure 5.7 displays the effect on the accuracy on test set (i.e. OAtest) as the numbers of hidden nodes and training iterations were varied for the $CYT/\sim CYT$ binary classifier. The patterns in Figure 5.7 do not portray a clear and constant behaviour of

the OAtest with respect to the change in the numbers of hidden nodes and training iterations. When the hidden nodes were increased from 5 to 10 and 10 to 20 for instance, Figure 5.7 shows that the OAtest increased, except for when the number of training iterations was 200, where the OAtest decreased as the hidden nodes were increased from 10 to 20. When the hidden nodes were increased from 20 to 30 and 30 to 40, Figure 5.7 shows that the OAtest remained constant for the case where 25 training iterations was used, while the OAtest increased for the cases of 50 and 200 iterations. For the cases of 100 and 150 iterations, the OAtest decreased as the hidden nodes were increased from 20 to 30, and then increased as the hidden nodes were increased from 30 to 40. Figure 5.7 also shows that for the number of hidden nodes in the range between 20 and 40, the NN classifiers trained for 150 iterations produced the best OAtest, suggesting that beyond 150 iterations, the networks started to over-fit the training set; and as a result, the accuracy on the test set started to decrease. But for the number of hidden nodes between 5 and 10, the best solution was produced by the networks trained for 200 iterations; suggesting that for network configurations with numbers of hidden nodes 5 and 10, there might still be room for improving the OAtest for training iterations beyond 200. This is because Figure 5.7 shows that the accuracy on the test set was still increasing for networks with 5 and 10 hidden nodes, when the maximum number of training iterations (i.e. 200) was reached. The observations based on Figure 5.7 are consistent with the conclusion reached when analysing the E.coli binary classifiers (see Section 5.3.3): determining the optimal number of hidden nodes should be coupled with the finding of the optimal number of training iterations; which is the basis of the trial and error approach.

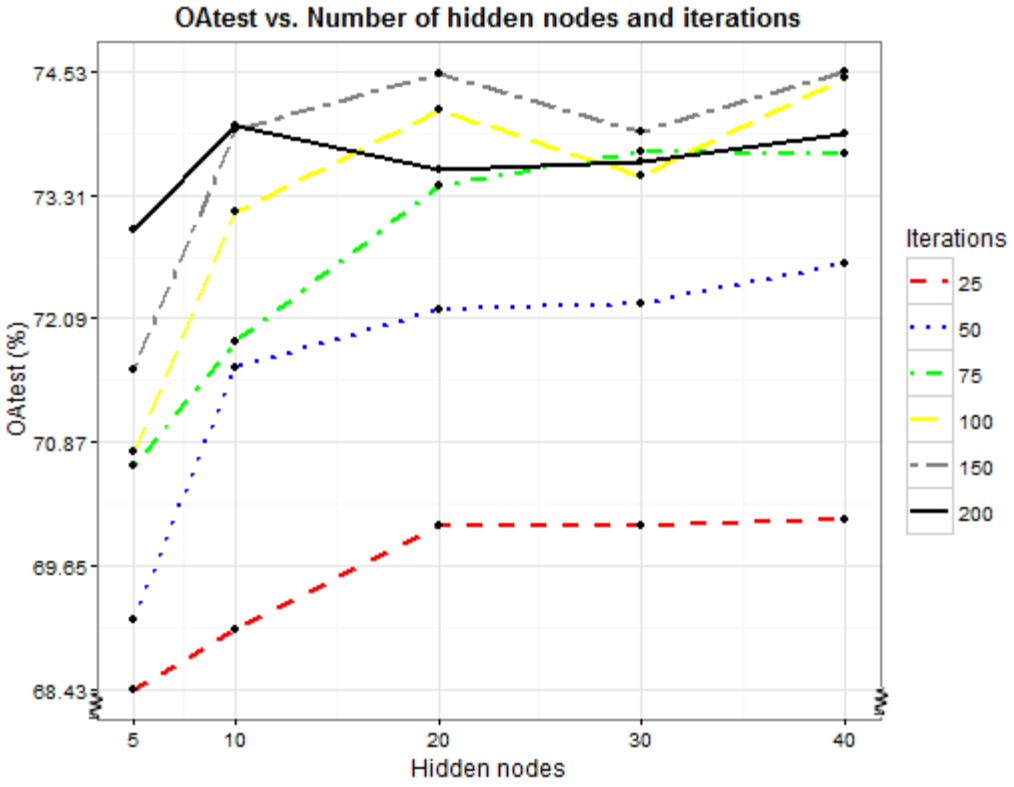

Figure 5.7: The OAtest for varying number of hidden nodes and training iterations for the CYT/~CYT binary classifier trained with Rprop

5.5.4 Effect on efficiency

Figure 5.8 portrays the effect of varying the numbers of hidden nodes and training iterations on the efficiency for the $CYT/{\sim}CYT$ binary classifier. In general, the patterns in Figure 5.8 suggest a positive relationship between the training times and the number of hidden nodes. The training times increased as the hidden nodes were increased. This suggests that the increase in the training times may have been due to the fact that the complexity of the networks increased as more hidden nodes were added. It is worth highlighting that when the hidden nodes were increased in the interval between 5 and 30, Figure 5.8 shows that the increase in the training times was very small; especially for networks trained for less than 150 iterations. The training times increased significantly when the hidden nodes were increased from 30 to 40 hidden nodes, especially for networks trained for 100, 150 and 200 iterations.

The positive relationship between the number of training iterations and the training time is evident in Figure 5.8. The results show that the more the maximum number of iterations, the more time was required to finish training. This is shown in Figure 5.8 by the fact that the lines corresponding to the higher numbers of training iterations were situated at the higher values of training times.

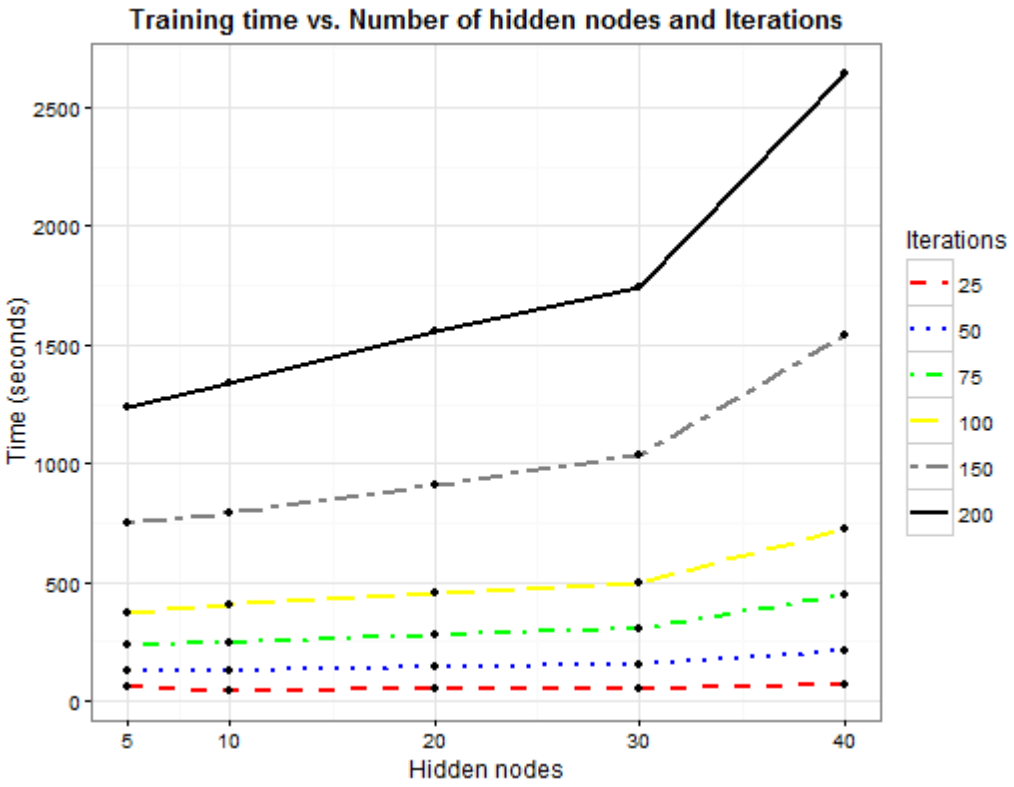

Figure 5.8: Training time (in seconds) for varying number of hidden nodes and training iterations for the CYT/~CYT binary classifier trained with Rprop

5.6 Comparing the E.coli classifiers and the Yeast classifiers

The results have thus far shown that the E.coli classifiers were more accurate than the Yeast classifiers. The assumed reason for this difference was that the Yeast classifiers may have been more complex than the E.coli classifiers. In other words, the structure of the Yeast data may have been more complex than the structure of E.coli data. When we refer complexity in this context, we are referring to the

interconnections and overlaps between classes in the data. If there are too many overlaps between classes, the decision boundary between different classes would be difficult to estimate. The data points that are in the region where classes overlap with other classes may be incorrectly classified. If there are a large number of data points in this region, the misclassification rate will be higher. This will mean that in these situations the accuracies are going to be lower than when there are less overlaps (Ho and Basu, 2002; Attoor and Dougherty, 2004). In this regard, we implicitly evaluated the complexity of data through the performance of a classifier in classifying that data. Poor performance for a particular classifier was an indication that maybe the data was intrinsically very complex for the classifier to correctly classify it.

The results have also shown differences in accuracy between the E.coli classifiers and between the Yeast classifiers. The difference of complexity between E.coli classifiers and between the Yeast classifiers was assumed to be the reason. Therefore, in this section and Section 5.7 following, we analysed the problem of complexity of classifiers (note that the complexity of classifiers is related to the complexity of class structures of the dataset that they are classifying). This section focuses on a general performance and complexity comparison of the E.coli and the Yeast classifiers. Section 5.7 delves into performance of particular classifiers.

Table 5.9: Best Test performances for the E.coli classifiers using Rprop

Binary classifiers	Total samples in class1 and class2	Times (seconds)	OAtest (%)	Atest1 (%)	Atest2 (%)
$cp/\sim cp$	173/~163	36	96.85	97.52	96.34
$im/\sim im$	77/~259	69	91.2	87.41	92.33
$pp/\sim pp$	52/~284	33	95.01	93.16	95.35
$imU/\sim imU$	35/~301	217	92.75	83.3	93.86
$om/\sim om$	20/~316	29	98.4	97.05	98.49
Mean	**71/~265**	77	94.84	91.69	95.27
Stdv	**54/~54**	72	2.62	5.54	2.1

We used the results from our experiments to compare the performance of the E.coli and the Yeast classifiers. The experiments were conducted using the same parameters for Rprop and training scenarios as specified in Section 3.2.3 and Section 4.5, respectively. The results obtained involve the best OAtest for each classifier, with their corresponding test accuracy of class 1 (Atest1), test accuracy of class 2 (Atest2), and training times to yield the specified accuracies on test set. Moreover, we used the distribution of samples in class 1 and class 2 as another indication of the complexity of a classifier. In brief, Atest1 and Atest2 can be understood as follows. Take for instance the $cp/\sim cp$ binary classifier; it contains two classes, i.e. cp (class 1) and no cp or ~cp (class 2). Atest1 is the test classification accuracy of cp (the ability of the classifier $cp/\sim cp$ to correctly classify cp on the test set), and Atest2 is the test classification accuracy of no cp (the ability of the classifier $cp/\sim cp$ to correctly classify no cp on the test set).

Table 5.10: Best Test performances for the Yeast classifiers using Rprop

Binary classifiers	Total samples in class1 and class2	Times (seconds)	OAtest (%)	Atest1 (%)	Atest2 (%)
$CYT/\sim CYT$	463/~1021	1537	74.53	53.99	83.85
$NUC/\sim NUC$	429/~1055	306	77.95	50.55	89.08
$MIT/\sim MIT$	244/~1240	249	89.12	57.35	95.16
$ME3/\sim ME3$	163/~1321	245	95.27	82.34	96.86
$ME2/\sim ME2$	51/~1433	272	96.86	54.96	98.36
$ME1/\sim ME1$	44/~1440	1554	98.3	91.62	98.51
$EXC/\sim EXC$	37/~1447	1554	98.17	66.56	98.92
$VAC/\sim VAC$	30/~1454	45	97.98	22.6	99.54
$POX/\sim POX$	20/~1464	233	99.03	56.31	99.61
Mean	**165/~1319**	666	91.91	59.59	95.54
Stdv	**166/~166**	628	8.87	18.59	5.18

Table 5.9 gives the best test results for the E.coli classifiers, and Table 5.10 gives the best test results for the Yeast classifiers. The results in Tables 5.9 and 5.10 show that,

in general, the E.coli classifiers performed better than the Yeast classifiers. On average, the OAtest for the E.coli classifier was 94.84% with a standard deviation of 2.62%, while that for the Yeast classifier was 91.91% with a standard deviation of 8.87%. The E.coli classifier was hence 4% more accurate than the Yeast classifier, on average. The superiority of the E.coli classifiers over the Yeast classifiers was also reflected in the ability of both classifiers to accurately classify class 1 and class 2. On average, the Atest1 and Atest2 for the E.coli classifier were 91.69% and 95.27% respectively, whereas those for the Yeast classifier were 59.59% and 95.54% respectively. The results suggest that the E.coli classifiers were able to classify both class 1 and class 2 very well, whereas the Yeast classifiers performed poorly on class 1 and very well on class 2. As said before, this difference in performance between the E.coli and Yeast classifiers may be attributed to some extent, to the difference of complexity between the class structures of the E.coli and Yeast data. The class structure for the Yeast data may have been more complex than the class structure for the E.coli data, which may have led to the E.coli classifiers performing better than the Yeast classifiers.

In an attempt to better evaluate the complexity of the two datasets, we also included the distribution of samples in class 1 and class 2 for every classifier, and the time required to train every one of them. Table 5.9 and Table 5.10 provide the results for the E.coli classifiers and for the Yeast classifiers, respectively. The results in both tables show that the training times for the Yeast classifiers were longer than those for the E.coli classifiers. On average, a Yeast classifier needed 666 seconds to finish training, whereas an E.coli classifier needed 77 seconds to finish training. In other words, a Yeast classifier took about 8.6 times longer than an E.coli classifier for training. This suggests to some degree that there seemed to be a positive relationship between the training time and the number of samples for a classifier. For the E.coli classifier, the sample size was 336 with on average, 71 and 265 samples in class 1 and class 2, respectively (i.e. 71/~265); whereas for the Yeast classifier, the sample size was 1484 with on average, 165 and 1319 samples in class 1 and class 2, respectively (165/~1319). These distributions of samples seem to have influenced the

training times of the classifiers. The larger the samples, the longer the classifiers took for training.

Another aspect we analysed was the imbalanced nature (i.e. the high unequal distribution of samples in class 1 and class 2, with in general, class 2 containing far more samples than class 1) of the E.coli and Yeast classifiers. Table 5.9 and Table 5.10 show that, in general, the Yeast classifiers were more imbalanced than the E.coli classifiers. It is worth emphasising that, in this context, a classifier is said to be imbalanced if it is classifying an imbalanced dataset. The general impression that may have arisen from analysing the training times of the different classifiers in Tables 5.9 and 5.10 is that the more imbalanced classifiers required lesser training times than the less imbalanced ones. This was only true to some extent, because there were more imbalanced classifiers that required more training time than the less imbalanced ones. For E.coli for instance, $im/{\sim}im$ (77/~259) was more imbalanced than $cp/{\sim}cp$ (173/~163); but it required more training time (i.e. 69 seconds) than $cp/{\sim}cp$ which required 36 seconds for training. For Yeast, $ME1/{\sim}ME1$ (44/~1440) was more imbalanced than $MIT/{\sim}MIT$ (244/~1240), but required more training time (i.e. 1554 seconds) than $MIT/{\sim}MIT$ which required 249 seconds for training. Moreover, while the Yeast classifiers were the more imbalanced, they required more training time than the E.coli classifiers.

To summarise, results in Tables 5.9 and 5.10 suggest that, in general, the Yeast classifiers were less accurate and took longer to train than the E.coli classifiers. This situation may be explained by the number of samples that needed to be classified by each classifier. The Yeast classifiers were dealing with more samples than the E.coli classifiers. They thus required more training times than the E.coli classifiers. The more samples a classifier had to classify, the more time was required to classify them. But the number of samples did not account for all the differences in accuracies and times between the Yeast and E.coli classifiers. The structure of the data itself and the distribution of samples in different classes were very important features which could have made the task of distinguishing between classes difficult. It can be assumed that, because the Yeast classifiers were less accurate than the E.coli ones, the structure of the Yeast data was more complex than that of the E.coli data, which

may have led to the better performance of the E.coli classifiers compared to the Yeast classifiers.

5.7 Further Experiments - Evaluating the complexity of classifiers

5.7.1 Introduction

We have observed based on the above experiments that the performances of the various classifiers were different. For the same experimental design, some classifiers achieved very good performance, whereas others performed poorly. This has been observed for both the E.coli and Yeast classifiers. For the E.coli data for instance, the $om/\sim om$ classifier performed better than $im/\sim im$ classifier. The OAtrain and OAtest for $om/\sim om$ were respectively 99.91% and 98.4%, while those for $im/\sim im$ were respectively 97.59% and 91.2% (see results produced by Rprop in Table 5.2). For the Yeast data for instance, the $POX/\sim POX$ classifier performed better than the $CYT/\sim CYT$ classifier. The OAtrain and OAtest for $POX/\sim POX$ were respectively 99.4% and 99.04%, while those for $CYT/\sim CYT$ were respectively 78.55% and 74.53% (see results produced by Rprop in Table 5.6). Based on these results, we argued that the difference in accuracies between these classifiers may have been due to the difference of complexity between the class structures of the data that had to be classified by these classifiers. The classifiers that dealt with more complex class structures produced better accuracies than the classifiers that dealt with less complex class structures. Thus, we can suggest for the E.coli data that the $im/\sim im$ classifier dealt with the more complex class structure since it was the less accurate classifier, and the $om/\sim om$ classifier dealt with the less complex class structure since it was the more accurate classifier. For the Yeast data, the $CYT/\sim CYT$ classifier dealt with the more complex class structure since it was the less accurate classifier, and the $POX/\sim POX$ dealt with the less complex class structure since it was the more accurate classifier.

We can also suggest based on the above results that the training scenarios as described in Section 4.5, which were (numbers of hidden nodes and iterations) utilised for the training of the classifiers that produced the less accuracies, may not have been appropriate. The numbers of hidden nodes and training iterations applied may have been unnecessarily very high or insufficient for the less performing E.coli and Yeast classifiers. If that was the case, there might be room for improvement of the less performing E.coli and Yeast classifiers, provided that the appropriate training scenarios are found. To test this assumption, it was worthwhile doing further experiments and to see if we could eventually improve the performance of the classifiers that performed poorly. For further experiments, we chose two binary classifiers; one binary classifier with the lowest accuracies among the E.coli binary classifiers and another one with the lowest accuracies among the Yeast binary classifiers. The E.coli binary classifier chosen was the $im/\sim im$ classifier, and the Yeast binary classifier chosen was the $CYT/\sim CYT$ classifier. This gave us a possibility of testing if we could improve their performance under different training scenarios from the ones used in the previous experiments as specified in section 4.5. Also, utilising these two above specified worst performing classifiers could highlight the complexity of the class structures of the data that had to be classified by these two E.coli and Yeast classifiers. The performance results for the $im/\sim im$ classifier and $CYT/\sim CYT$ classifier under the further training scenarios applied are presented in the following sections. The training was done using Rprop since it was the better performing algorithm.

5.7.2 Comparing the performance of the $im/\sim im$ E.coli classifier and $CYT/\sim CYT$ Yeast classifier

In this section, we used different training scenarios to compare the performance results of the $im/\sim im$ E.coli classifier and $CYT/\sim CYT$ Yeast classifier for the reasons provided in section 5.7.1. The experimental designs applied in this section are as follows: 1) networks with 1, 2, 3, 4 and 5 hidden nodes trained for 25, 50, 75, 100, 150 and 200 iterations; 2) networks with 5, 10, 20, 30 and 40 hidden nodes

trained for 2, 5, 10, 15, 20 and 24 iterations. Both the first and second experimental designs had 30 training scenarios each. The choice of these training scenarios was based on the trial and error method for determining the appropriate numbers of hidden nodes and training iterations (see Section 4.3.2).

Also, results presented so far have shown that training scenarios that used networks with hidden nodes in the range between 5 and 40, trained for iterations in the range between 25 and 200, could produce acceptable results for the E.coli classifiers (see Table 5.2) and Yeast classifiers (see Table 5.6). It was therefore deemed appropriate for the experiments in this section, to use training scenarios with either reduced numbers of hidden nodes (as in experimental design 1) or reduced numbers of training iterations (as in experimental design 2), to see whether the performance of the above specified classifiers would improve. Furthermore, the results from the training scenarios applied in this section would establish whether the values for hidden nodes and iterations previously used (see Section 4.5) where not unnecessarily large or insufficient.

5.7.2.1 Performance based on scenarios of 1-5 hidden nodes with 25-200 iterations for $im/\sim im$ and $CYT/\sim CYT$

The training scenarios that utilised the network configurations of 1, 2, 3, 4 and 5 hidden nodes, trained for 25, 50, 75, 100, 150 and 200 iterations, were implemented and the performance results for the $im/\sim im$ and $CYT/\sim CYT$ are presented in this section. Figure 5.9 portrays the overall accuracies on test sets (or OAtest) produced by all the scenarios, for the $im/\sim im$ classifier; and Figure 5.10 does so for the $CYT/\sim CYT$ classifier. The results in Figure 5.9 suggest that the best OAtest for the $im/\sim im$ classifier (i.e. 90.65%) was produced by the network configuration of 5 hidden nodes, trained for 50 iterations (or 5hn_50t), while the results in Figure 5.10 suggest that the best OAtest for the $CYT/\sim CYT$ classifier (i.e. 72.75%) was produced by the network configuration of 5 hidden nodes, trained for 200 iterations (or 5hn_200t). Also, the patterns in Figure 5.9 show that for network configuration of 5 hidden nodes, the OAtest of the $im/\sim im$ classifier started to decrease when the

number of iterations was beyond 50. This suggests that for network configuration of 5 hidden nodes, the $im/\sim im$ classifier was over-trained when the number of iterations exceeded 50. The patterns in Figure 5.10 show that for the network configuration of 5 hidden nodes, the OAtest for the $CYT/\sim CYT$ classifier was apparently still increasing when the number of iterations reached 200. This suggests that maybe there was still room for increase of the OAtest of the $CYT/\sim CYT$ classifier, beyond 200 iterations.

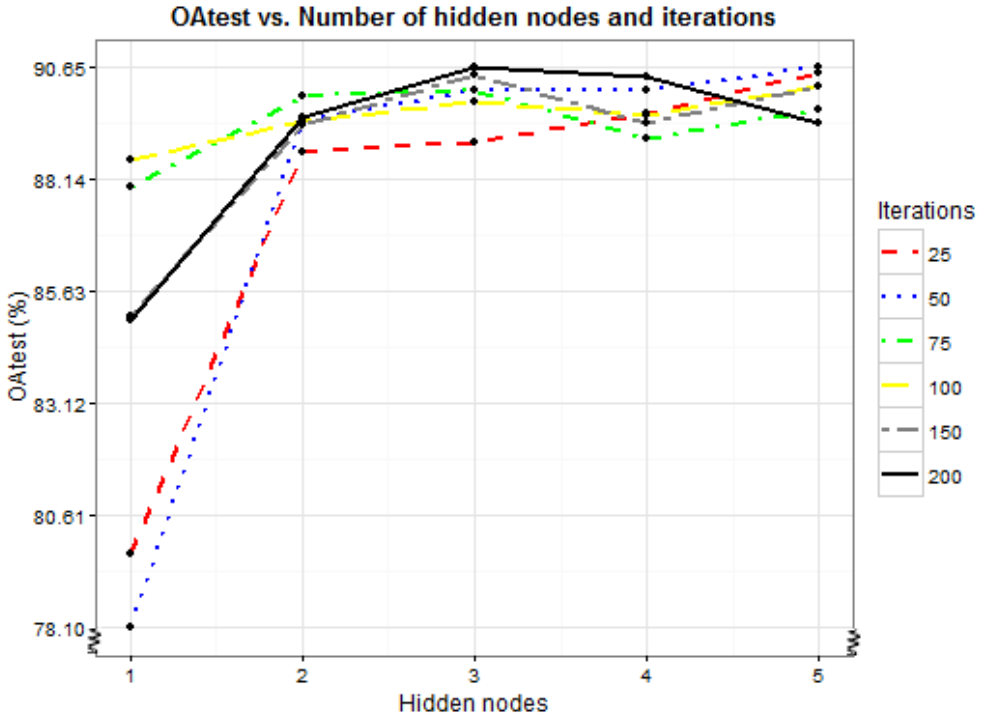

Figure 5.9: OAtest for 1-5 hidden nodes with 25-200 training iterations for the *im/~im* binary classifier

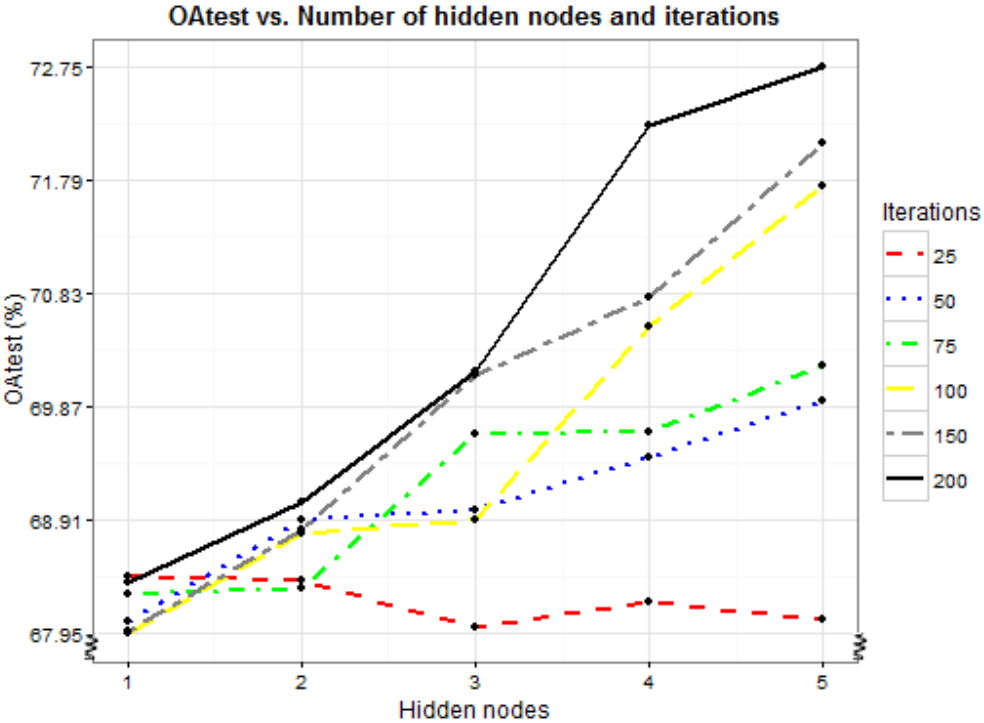

Figure 5.10: OAtest for 1-5 hidden nodes with 25-200 training iterations for the *CYT/~CYT* binary classifier

Table 5.11 summarises the performance results based on the best training scenario for the $im/\sim im$ classifier and the best training scenario for the $CYT/\sim CYT$ classifier. These results were extracted from the 30 training scenarios as described previously. The results in the top part of Table 5.11 are related to the training scenario (i.e. 5hn_50t which is described as network with 5 hidden nodes (hn) trained for 50 iterations (t)) that produced the best accuracy on test set (OAtest), for the $im/\sim im$ classifier. Based on the same training scenario (i.e. 5hn_50t), the results produced for the $CYT/\sim CYT$ classifier are also provided in the top part of Table 5.11. The results in the bottom part of Table 5.11 are related to the training scenario (i.e. 5hn_200t) that produced the best OAtest, for the $CYT/\sim CYT$ classifier. Based on the same training scenario (i.e. 5hn_200t), the results produced for the $im/\sim im$ classifier are also provided in the bottom part of Table 5.11. In a nutshell, the results in Table 5.11 can be interpreted as follows: if the $im/\sim im$ classifier is trained by utilising a network configuration of 5 hidden nodes trained for 50 iterations (i.e. 5hn_50t), the expected performance results would be 93.21% as OAtrain, 90.65% as OAtest, 0.0479 as MSE and 39 seconds as training time. For the same training scenario (5hn_50t), the corresponding results for the $CYT/\sim CYT$ classifier would be 72.15% as OAtrain, 69.9% as OAtest, 0.1986 as MSE and 346 seconds as training time.

The results in Table 5.11 clearly show that the $im/\sim im$ classifier outperformed the $CYT/\sim CYT$ classifier in terms of accuracy, efficiency and convergence for both training scenarios 5hn_50t and 5hn_200t. Since these two training scenarios, out of the 30 training scenarios implemented, are the ones that produced the best accuracies on test set for both classifiers, we can suggest that the $im/\sim im$ classifier outperformed the $CYT/\sim CYT$ for all training scenarios. Based on the results in Table 5.11, we can also suggest that the $CYT/\sim CYT$ classifier was more complex to train than the $im/\sim im$ classifier, because the $CYT/\sim CYT$ classifier was less accurate and less efficient than the $im/\sim im$ classifier. The complexity of the $CYT/\sim CYT$ classifier compared to that of the $im/\sim im$ classifier may be explained by the fact that the $CYT/\sim CYT$ classifier had to classify data with more complex class structure than the data that had to be classified by the $im/\sim im$ classifier. Since $CYT/\sim CYT$ is

a Yeast classifier and $im/\sim im$ is an E.coli classifier, we can also suggest based on the results in Table 5.11 that the Yeast classifiers were more complex to train than the E.coli classifiers.

Table 5.11: Best *im/~im* and *CYT/~CYT* based on OAtest for 1-5 hidden nodes with 25-200 training iterations

Performance based on the best $im/\sim im$					
Binary classifiers	OAtrain (%)	OAtest (%)	MSE	Time (seconds)	Scenario
$im/\sim im$	93.21	90.65	0.0479	39	5hn_50t
$CYT/\sim CYT$	72.15	69.9	0.1986	346	5hn_50t
Performance based on the best $CYT/\sim CYT$					
Binary classifiers	OAtrain (%)	OAtest (%)	MSE	Time (seconds)	Scenario
$im/\sim im$	95.35	88.62	0.0458	144	5hn_200t
$CYT/\sim CYT$	74.23	72.75	0.1663	1242	5hn_200t

5.7.2.2 Performance based on scenarios of 5-40 hidden nodes with 2-24 iterations for $im/\sim im$ and $CYT/\sim CYT$

The training scenarios that applied the network configurations of 5, 10, 20, 30 and 40 hidden nodes, trained for 2, 5, 10, 15, 20 and 24 iterations, were implemented and the performance results for the $im/\sim im$ and $CYT/\sim CYT$ are presented in this section. Figure 5.11 depicts the OAtest produced by all the scenarios for the $im/\sim im$ classifier and Figure 5.12 does the same for the $CYT/\sim CYT$ classifier. The patterns in Figure 5.11 suggest that the best OAtest for the $im/\sim im$ classifier (i.e. 91.37%) was produced by the network configuration of 20 hidden nodes trained for 24 iterations (or 20hn_24t), while the patterns in Figure 5.12 suggest that the best OAtest for the $CYT/\sim CYT$ classifier (i.e. 70.34%) was produced by the network

configuration of 40 hidden nodes trained for 24 iterations (or 40hn_24t). Also, Figure 5.11 and Figure 5.12 show that for all the training scenarios, the OAtests for the $im/{\sim}im$ classifier were in the range between 72% and 91.5%, while those for the $CYT/{\sim}CYT$ classifier were in the range between 58% and 70.5%. Overall, the $im/{\sim}im$ classifier produced much better accuracies than the $CYT/{\sim}CYT$ classifier for all the training scenarios, suggesting that the $im/{\sim}im$ classifier was less complicated to train than the $CYT/{\sim}CYT$ classifier.

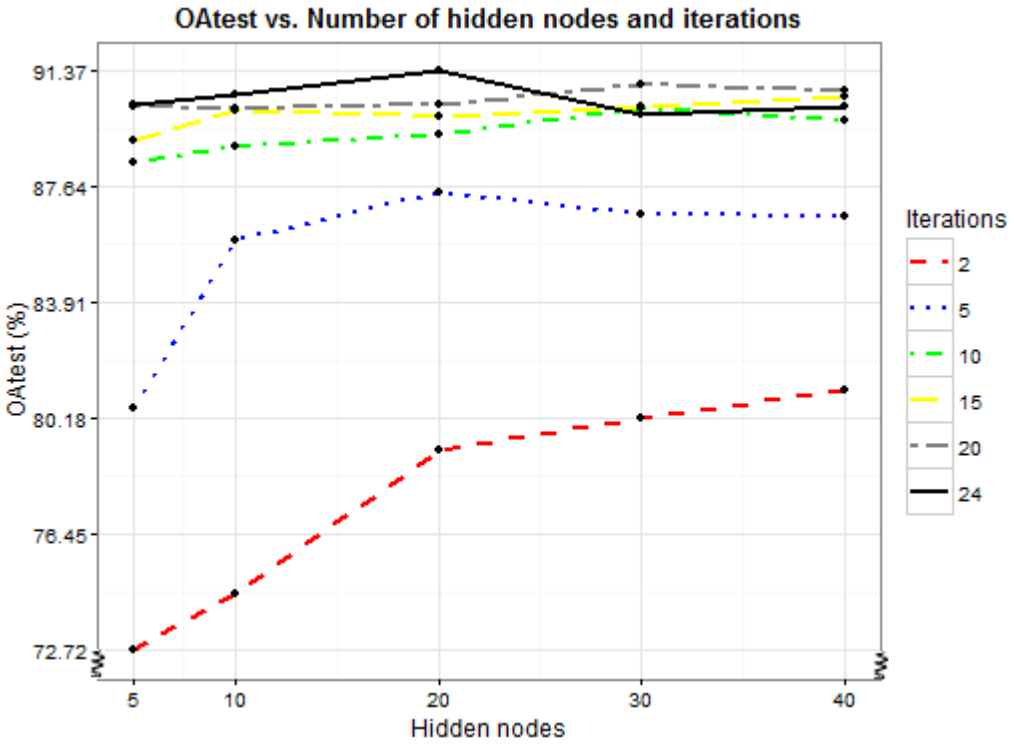

Figure 5.11: OAtest for 5-40 hidden nodes with 2-24 training iterations for the *im/~im* binary classifier

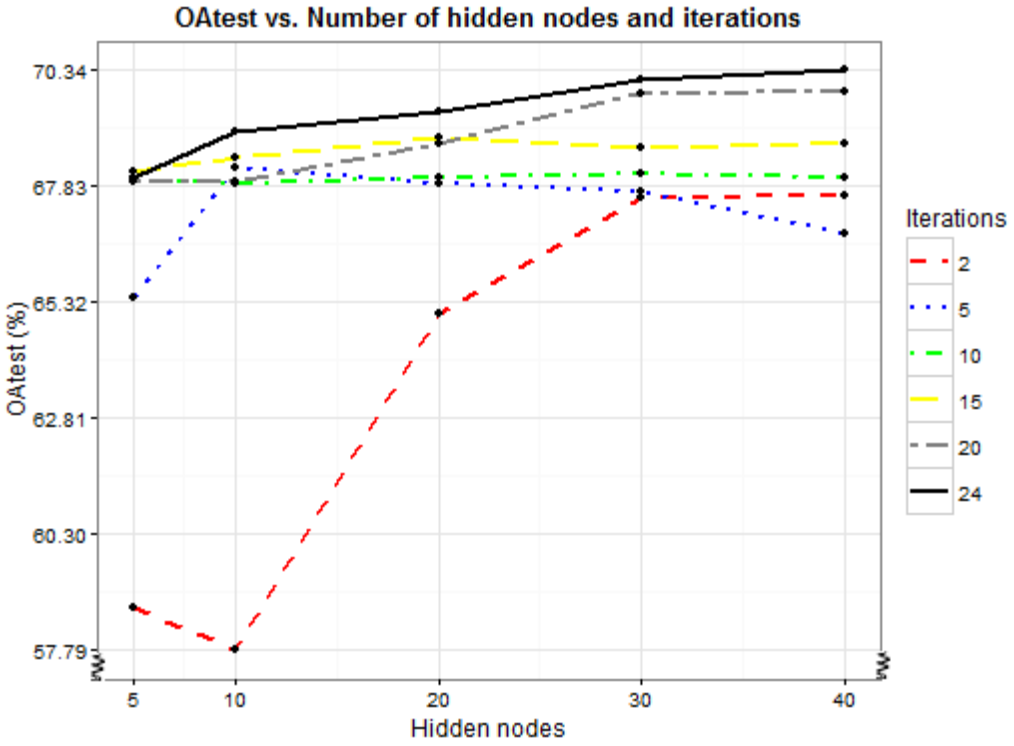

Figure 5.12: OAtest for 5-40 hidden nodes with 2-24 training iterations for the *CYT/~CYT* binary classifier

Table 5.12 provides the performance results based on the best training scenario for the $im/\sim im$ classifier and the best training scenario for the $CYT/\sim CYT$ classifier obtained from the 30 training scenarios as described above. The top part of Table 5.12 provides the results based on the training scenario (i.e. 20hn_24t) that produced the best $im/\sim im$ classifier, while the results in the bottom part are based on the training scenario (i.e. 40hn_24t) that produced the best $CYT/\sim CYT$ classifier (see Section 5.7.2.1 for interpretation of table). The results in Table 5.12 indicate that the $im/\sim im$ classifier outperformed the $CYT/\sim CYT$ classifier in terms of accuracy, efficiency and convergence for both training scenarios 20hn_24t and 40hn_24t. Based on scenario 40hn_24t for instance, the OAtest for $im/\sim im$ was 90.12%, while that for $CYT/\sim CYT$ was 70.34%. The results in Table 5.12 also suggest that the $CYT/\sim CYT$ classifier was more complex than the $im/\sim im$ classifier, because the $CYT/\sim CYT$ classifier required many hidden nodes (40) to achieve its best

performance, while the $im/\sim im$ classifier required fewer hidden nodes (20) to achieve its best performance.

Table 5.12: Best *im/~im* and *CYT/~CYT* based on OAtest for 5-40 hidden nodes with 2-24 training iterations

Performance based on the best $im/\sim im$					
Binary classifiers	OAtrain (%)	OAtest (%)	MSE	Time (seconds)	Scenario
$im/\sim im$	93.37	91.37	0.0497	15	20hn_24t
$CYT/\sim CYT$	70.85	69.21	0.1944	52	20hn_24t
Performance based on the best $CYT/\sim CYT$					
Binary classifiers	OAtrain (%)	OAtest (%)	MSE	Time (%)	Scenario
$im/\sim im$	92.98	90.12	0.0525	18	40hn_24t
$CYT/\sim CYT$	71.07	70.34	0.1765	63	40hn_24t

5.7.2.3 Concluding remarks

We suspected that the differences in performance between the E.coli and Yeast classifiers may have been due to the difference of complexity between the class structures of the data that were to be classified by the different classifiers. Also, given that some classifiers achieved very high accuracies while others performed poorly, we assumed that, the first proposed experimental design (5-40hn with 25-200t) for training may not have been appropriate for some of the classifiers. We thought that perhaps the numbers of hidden nodes and training iterations were unnecessarily very high for some binary classifiers, especially for those with very high accuracies, or insufficient (very small), especially for those with lower accuracies (i.e. having more misclassifications).

To test the above assumptions, we utilised extra training scenarios to evaluate the performance of the E.coli and Yeast classifiers. The classifiers chosen for the extra training scenarios were the $im/\sim im$ classifier and the $CYT/\sim CYT$ classifier. These two classifiers were chosen because they were the two classifiers with the lowest accuracies. The $im/\sim im$ classifier and the $CYT/\sim CYT$ classifier produced the lowest accuracy among the E.coli classifiers and the Yeast classifiers respectively. The interest of using these two worst performing classifiers was to highlight the complexity of the data structures that these two classifiers were dealing with, and see if the extra training scenarios could improve their performance.

The results from the extra experiments have shown little improvements in accuracy (from 91.2% to 91.37%) for the $im/\sim im$ classifier and no improvement at all for the $CYT/\sim CYT$ classifier. Actually, the accuracy for the $CYT/\sim CYT$ classifier decreased under the extra experiments described in Sections 5.7.2.1 and 5.7.2.2, compared to the accuracy obtained under the original experiments described in Section 4.5. The accuracies on test set obtained under the various experimental designs are summarised in Table 5.13 for $im/\sim im$ and $CYT/\sim CYT$ classifiers. We also observed based on the results obtained under the extra experiments that the $CYT/\sim CYT$ classifier was more complex to train than the $im/\sim im$ classifier, because the $CYT/\sim CYT$ yielded very low accuracies when compared to the accuracies yielded by the $im/\sim im$ classifier under the various experimental designs (see Table 5.13). Finally, since $CYT/\sim CYT$ is a Yeast classifier and performed poorly than $im/\sim im$ which is an E.coli classifier, we hence suggested that the Yeast classifiers we more complex to train than the E.coli classifiers. This also indicated that the decision boundaries between the Yeast class structures were more complex to estimate than the decision boundaries between the E.coli class structures.

Table 5.13: Best *im/~im* and *CYT/~CYT* based on OAtest for the various experimental designs

Experimental designs	$im/\sim im$	$CYT/\sim CYT$
	OAtest (%)	OAtest (%)
5-40hn with 25-200t	91.2	74.53
1-5hn with 25-200t	90.65	72.75
5-40hn with 2-24t	91.37	70.34

5.7.3 Attempts to improve the performance of the $CYT/\sim CYT$ classifier by increasing the number of iterations

The performance results presented in Section 5.7.2 have indicated that the $CYT/\sim CYT$ classifier was the more complex of all the E.coli and Yeast binary classifiers analysed in our experiments, because the $CYT/\sim CYT$ classifier produced the lowest accuracies among all the classifiers based on the training scenarios applied so far. The largest number of training iterations utilised so far in the various training scenarios has been 200. Given that with this number of iterations the classification accuracy for the $CYT/\sim CYT$ classifier has shown no improvement, we deemed worthwhile increasing the number of iterations well beyond 200, in an attempt to increase the accuracy for the $CYT/\sim CYT$ classifier. The following training scenarios were therefore used to improve the performance of the $CYT/\sim CYT$ classifier: 1) network configurations of 1, 2, 3, 4 and 5 hidden nodes, trained for 500, 1000, 2000, 3000, 3500 and 4000 iterations; 2) network configurations of 15, 20, 25, 30 and 35 hidden nodes, trained for 500, 1000, 2000, 3000, 3500 and 4000 iterations. These training scenarios were defined based on the trial and error method coupled with the motivations of testing the performance of the $CYT/\sim CYT$ classifier under different training scenarios from those that have been applied so far. The performance results are presented in the following sections.

5.7.3.1 Results based on scenarios of 1-10 hidden nodes with 500-4000 iterations for $CYT/\sim CYT$

Figure 5.13 portrays the performance results obtained under the training scenarios that utilised the network configurations of 1-10 hidden nodes trained for 500-4000 iterations for the $CYT/\sim CYT$ classifier. The patterns in Figure 5.13 suggest that the highest OAtest was produced for the network configuration of 5 hidden nodes trained for 2000 iterations (i.e. 5hn_2000t). The accuracy decreased as the number of iterations was increased beyond 2000. Table 5.14 present the performance results of the best scenario (5hn_2000t) out of the 30 training scenarios used in this section. The results in Table 5.14 indicate that the best OAtest was 73.79% and the training time was 110590 seconds.

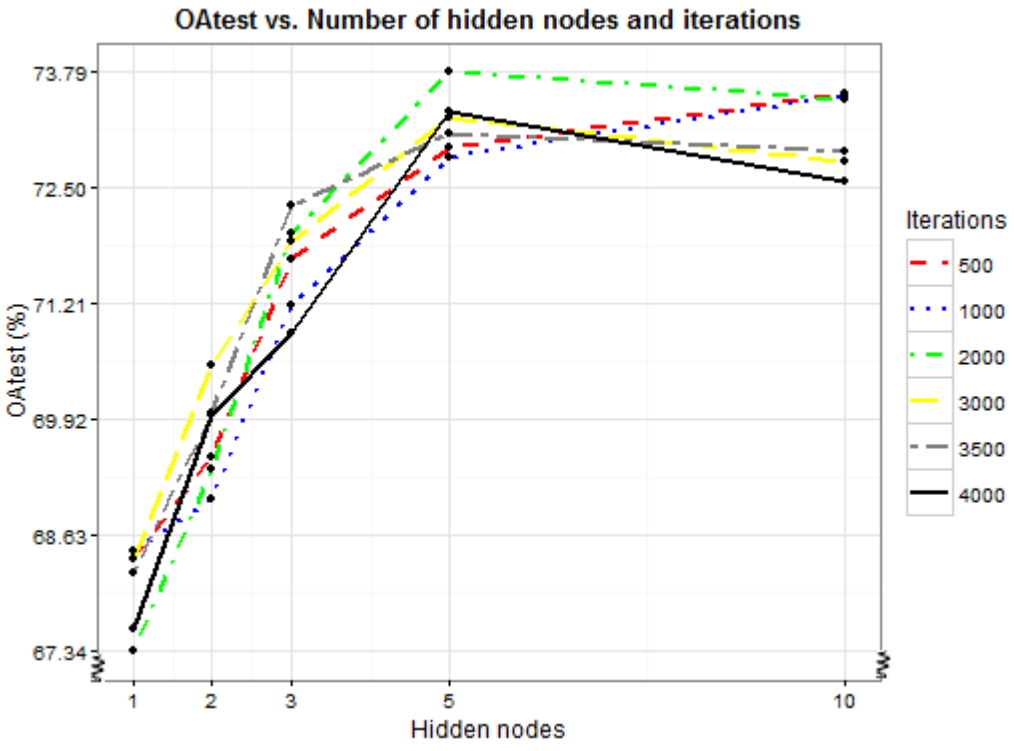

Figure 5.13: OAtest for 1-10 hidden nodes with 500-4000 training iterations for *CYT/~CYT* binary classifier

Table 5.14: Best *CYT/~CYT* based on OAtest for 1-10 hidden nodes with 500-4000 training iterations

Binary classifiers	OAtrain (%)	OAtest (%)	MSE	Time (seconds)	Architecture
$CYT/\sim CYT$	76.64	73.79	0.1568	110590	5hn_2000t

5.7.3.2 Results based on scenarios of 15-35 hidden nodes with 500-4000 iterations for $CYT/\sim CYT$

Figure 5.14 portrays the performance results obtained under the training scenarios that utilised the network configurations of 15-35 hidden nodes trained for 500-4000 iterations for the $CYT/\sim CYT$ classifier. The patterns in Figure 5.14 indicate that the highest OAtest was produced for the network configuration of 25 hidden nodes trained for 500 iterations (i.e. 25hn_500t). The accuracy decreased as the number of iterations was increased beyond 500. Table 5.15 shows the performance results of the best scenario (25hn_500t) out of the 30 training scenarios used in this section. The results in Table 5.15 indicate that the best OAtest was 73.72% with the corresponding training time of 9248 seconds.

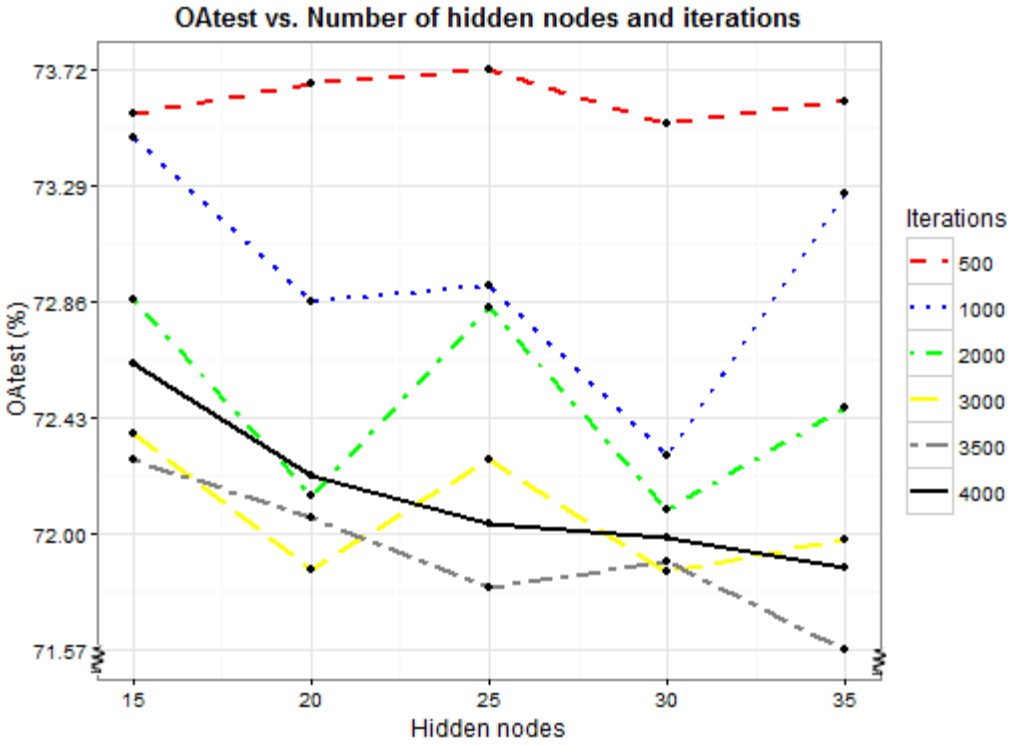

Figure 5.14: OAtest for 15-35 hidden nodes with 500-4000 training iterations for the *CYT/~CYT* binary classifier

Table 5.15: Best *CYT/~CYT* based on OAtest for 15-35 hidden nodes with 500-4000 training iterations

Binary classifiers	OAtrain (%)	OAtest (%)	MSE	Time (seconds)	Architecture
$CYT/{\sim}CYT$	79.33	73.72	0.1416	9248	25hn_500t

5.7.3.3 Concluding remarks

We applied training scenarios that used increased numbers of training iterations (beyond 200) to improve the performance of the $CYT/{\sim}CYT$ classifier. The results of these training scenarios, as presented in Sections 5.7.3.1 and 5.7.3.2, have shown that the best training scenarios were 5hn_2000t with the corresponding accuracy on test

set of 73.79%, for the design in Section 5.7.3.1; and 25hn_500t with the corresponding accuracy on test set of 73.72%, for the design in Section 5.7.3.2. We can therefore suggest based on these results that increasing the training iterations, well beyond 200, could not improve the accuracy of the $CYT/\sim CYT$ classifier. This suggestion is based on the fact that the accuracy of the $CYT/\sim CYT$ classifier did not improve under the training scenarios used in Sections 5.7.3.1 and 5.7.3.2 when compared to the accuracy (74.53%) obtained under the previous training scenarios (see Table 5.13).

CHAPTER 6: CONCLUSIONS

6.1 Introduction

Chapter 5 presented the analysis and results obtained from our experimental designs. This chapter provides the discussion and conclusions of the whole research investigation. Section 6.2 discusses the findings of this study, while Section 6.3 gives recommendations for possible future work based on these findings.

6.2 Discussion and conclusions

The process of NNs training has been proven to be very challenging. Complex networks are more likely to be characterised by features such as saddle-points, local minima, flat-spots and plateaus, which result in poor performance of NNs in terms of their efficiency, accuracy and convergence ability. To alleviate these issues, methods based on unconstrained and global optimisation theory have been proposed in the development of NNs training algorithms.

This study proposed the investigation of the performance of two NNs training algorithms derived from unconstrained and global optimisation theory, i.e. the Rprop and CGP algorithms. It further presented an empirical optimisation scheme of NNs training that involved using simultaneously the proposed training algorithms and the trial and error method. This method tried to find the optimal size of parameters, i.e. adequate numbers of hidden nodes and training iterations that were more likely to improve the performance of NNs based on the classification problems under consideration.

In order to reach the objectives of this study, two biological problems, i.e. the E.coli and Yeast problems, were used. These problems involved the classification of protein localisation patterns into different known classes, which are particularly useful in the

post-genomic era. Furthermore, these multiclass classification problems were transformed into multiple binary classification problems using the One-Against-All approach. Finally, different training scenarios were used to train these binary classifiers. Hence, the following conclusions were derived.

The Rprop algorithm performed better than the CGP algorithm. Using the OAtrain, OAtest, training times, and MSE, the Rprop algorithm was proven to be more accurate and efficient, and had better convergence ability than the CGP algorithm. Moreover, the superiority of Rprop was observed for all the E.coli and Yeast binary classifiers. However, the difference between the two training algorithms was more pronounced for some classifiers than it was for some others. This was believed to be the consequence of the difference of complexity between different classifiers. For more complex classifiers, the difference between Rprop and CGP was high, while for less complex classifiers the difference between the two algorithms was less.

We also argued that the small differences in accuracy between Rprop and CGP that were observed for some classifiers should not question the superiority of Rprop over CGP. The reason being as follows: although the accuracy differences were small in some cases, but CGP took much longer to achieve comparable results with Rprop. This means that for equal training times, the Rprop accuracy would be much better than that of CGP. Therefore, when training efficiency was taken into consideration, Rprop outperformed CGP unequivocally.

Conflicting claims are found in the literature about the performance of CG based methods. Some argue that CG based methods are devised to converge faster than GD based methods, because they update the weights in the conjugate directions of the gradient (Sharma and Venugopalan, 2014). Moreover, it is claimed that "CG based methods are probably the most famous iterative methods for efficiently training NNs due to their simplicity, numerical efficiency, and their very low memory requirements" (Ioannis and Panagiotis, 2012). With regard to efficiency, obviously these claims were not verified in our experiments. Instead, Rprop which is a GD based method, performed much better than CGP which is a CG based method, especially with regard to efficiency. However, others argue that despite their

theoretical and practical advantages in solving large scale unconstrained optimisation problems such as minimising NNs error functions, CG methods are characterised by a major drawback which is the use of restarting techniques in order to guarantee convergence. The restart may be activated too often, which could affect the overall efficiency of CG methods (Livieris and Pintelas, 2009; Andrei, 2011). The very poor efficiency of CGP observed in our experiments seemed to agree with this concern. CGP has been shown to be 3 to 4 times less efficient than Rprop.

Concerning the effect of varying the number of hidden nodes and number of training iterations, it appeared that this had an impact on the performance of the classifiers. This impact was shown and consistent in the classification accuracy, convergence and efficiency of the classifiers. This indicated that the trial and error method could assist in the finding of optimal parameters that in returns can help optimise the performance of NNs training.

Also, it was shown that depending on their complexity, some classifiers required more hidden nodes and training iterations to perform well, whereas some required less hidden nodes and training iterations to do so. In fact, the more complex a classifier was, the more hidden nodes and training iterations were needed; and the less complex a classifier was, the less hidden nodes and training iterations were needed. This proved that the performance of NNs training is dependent on the complexity of the application problems.

One interesting observation was that, increasing the number of hidden nodes could improve the training time for some classifiers. This was observed with the $cp/{\sim}cp$ classifier trained for 25 iterations; the training time decreased as we increased the hidden nodes in the range between 5 and 40 hidden nodes. This may seem strange because increasing the nodes amounts to increasing the number of weight parameters to be updated, which should increase the training time. However, this result (i.e. decrease in time as the number of hidden nodes was increased) was consistent with findings in some previous studies (Lawrence *et al.*, 1996; Livni *et al.*, 2014); in which it is claimed that, "Sufficiently Over-Specified Networks Are Easy to Train" (Livni *et al.*, 2014). In other words, it can be easy to train networks that are oversized

(i.e. networks that are larger than needed). This is true in the sense that the optimisation problem related to the training of sufficiently over-specified networks are easy to solve because in such networks, the presence of a global minimum is more likely than that of many local (non-global) minima. Hence, maybe it was easier to find the global minimum of $cp/\sim cp$ as we increased the hidden nodes, which resulted in less training time when the 25 iterations were completed.

Another finding was that the performance of NNs classifiers, especially in terms of efficiency, had a positive relationship with the number of samples to be trained. The more the number of samples to be trained, the more the training time and iterations were needed. The less the number of samples to be trained, the less the training time and iterations were needed. This was observed by comparing the performance of the E.coli classifiers to the performance of the Yeast classifiers. Since the Yeast data had more samples, it also required more time and iterations to be trained. This suggests to some extent that, the size of the data added to the complexity of the training process. However, it appeared that the structure of the data itself played the biggest role in its complexity. This was shown by the fact that the Yeast classifiers differed in their performances though they trained the same number of samples. This was also true for the E.coli classifiers.

Finally, it was observed that the best convergence during training did not necessarily result into the best generalisation of a classifier. This means that, during training, a network can reach the global minimum of the error function, and still does not produce the best performance on unseen data. Or, a network can reach a local minimum instead of the global minimum during training, and still perform better on unseen data. Therefore, achieving global convergence (reaching the global minimum) on training set should not be the main focus of NN training. Instead, the main focus should be to achieve a convergence (not necessarily the best one in terms of minimum) during training that would eventually allow the network to accurately perform its task when presented with new data.

6.3 Future work

One major characteristic of the protein data used for training the NNs classifiers in this study was their very imbalanced class sizes nature. Moreover, transforming these multiclass classification problems into multiple binary classification problems had even accentuated their imbalanced class sizes. We believe that this might be the main cause of the very small differences in classification accuracy between the Rprop and CGP algorithms for some binary classifiers. To test this assumption, one could consider for future work, using a more balanced and originally binary classification problem such as the cancer, diabetes problems (Murphy and Aha, 1994), and see if the difference between the two algorithms would not significantly increase.

Another future development could be to compare the performance of multiclass classifiers to that of binary ones in correctly classifying every class of the E.coli and Yeast proteins. Also, instead of the One-Against-All binary classifiers which accentuate the imbalance of these data, one could use the One-Against-One binary classifiers to assess the performance of the Rprop and CGP algorithms. However, it is worth highlighting that in this case, the number of binary classifiers to be trained will significantly increase, i.e. 28 (instead of 8) for the E.coli data, and 45 (instead of 10) for the Yeast data.

In this study, we implicitly assessed the classifiers complexity through their performances. The classifier with good performance was said to be less complex (i.e. it was easy to classify the data because there were less overlaps between the classes) and the classifier with poor performance was said to be more complex (i.e. it was difficult to classify the data because of much overlaps between the classes). We did not calculate a direct metric of data complexity as proposed by Ho and Basu (2002) and Attoor and Dougherty (2004), because it was not the main objective of our study. Moreover, we could not improve the accuracy of $CYT/\sim CYT$ even after trying various scenarios with very large numbers of hidden nodes and iterations. For future wok, one could focus on a detailed study of the complexity of the $CYT/\sim CYT$ classifier by following the steps proposed in the two above mentioned studies, and find an optimal way of significantly improving its accuracy.

REFERENCES

Akarachai, A. and Daricha, S., "Avoiding Local Minima in Feedforward Neural Networks by Simultaneous Learning", *Advances in Artificial Intelligence*, 4830, 100-109, 2007.

Anastasiadis, A.D., "Neural network training and applications using biological data", Doctoral Thesis, School of Computer Science and Information Systems, University of London, London, 2005.

Anastasiadis, A.D., Magoulas, G.D., and Vrahatis, M.N., "An efficient improvement of the Rprop algorithm", In *Proceedings of the First International Workshop on Artificial Neural Networks in Pattern Recognition*, Florence, Italy, *IAPR2003*, 197-201, 2003.

Anastasiadis, A.D., Magoulas, G.D., and Vrahatis, M.N., "Sign-based Learning Schemes for Pattern Classification", *Pattern recognition Letters*, 26, 1926-1936, 2005.

Andrei, N., "40 Conjugate Gradient Algorithms for Unconstrained Optimization", *ICI Technical Report*, 2008.

Andrei, N., "Open Problems in Nonlinear Conjugate Gradient Algorithms for Unconstrained Optimization", *ICI Technical, Report*, 2011.

Arai, M., "Bounds on the number of hidden units in binary valued three-layer neural networks", *Neural Networks*, 6, 855‑860, 1993.

Attoor, S.N., and Dougherty, E.R., "Classifier performance as a function of distributional complexity", *Pattern Recognition* 37:8, 1641-1651, 2004.

Basheer, I.A., and Hajmeer, M., "Artificial neural networks: fundamentals, computing, design, and application", *Journal of Microbiological Methods*, 43, 3-31, 2000.

Battiti, R., "First- and second-order methods for learning: between steepest descent and Newton's method", *Neural Computation*, 4, 141-166, 1992.

Berry, M.J.A., and Linoff, G., *Data Mining Techniques*, John Wiley and Sons, New York, 1997.

Bishop, C.M., *Neural Networks for Pattern Recognition*, Oxford University Press, 1995.

Blum, A., *Neural Networks in C++*, John Wiley and Sons, New York, 1992.

Boger, Z., and Guterman, H., "Knowledge extraction from artificial neural network models", *IEEE Systems, Man, and Cybernetics Conference*, Orlando, FL, USA, 1997.

Breiman, L., Friedman, J.H., Olshen, R.A., and Stone, C.J., *Classification and Regression Trees*, Wadsworth, Belmont, 1984.

Burton, R.M., and Mpitsos, G.J., "Event dependent control of noise enhances learning in neural networks", *Neural Networks*, 5, 627-637, 1992.

Canu, S., "Empirical criteria to compare the performance of neuro algorithms", In *Proceedings of the International Conference on Artificial Neural Networks*, Springer, London, 764-767, 1993.

Crammer, K., and Singer, Y., "On the algorithmic implementation of multiclass kernel-based vector machines", *Journal of Machine Learning Research*, 2, 265-292, 2001.

Dauphin, Y., Pascanu, R., Gulcehre, C., Cho, K., Ganguli, S., and Bengio, Y., "Identifying and attacking the saddle point problem in high-dimensional non-

convex optimization", In *Advances in Neural Information Processing Systems*, 2933–2941, 2014.

Duda, R.O., Hart, P.E., and Stork, D.E., *Pattern Classification*, John Wiley and Sons, Canada, 2000.

Efron, B., "Estimating the error rate of a prediction rule: improvement on cross-validation", *Journal of the American Statistical Association*, 78, 316-330, 1983.

Erin, A., Robert, S., and Yoram, S., "Reducing multiclass to binary: A unifying approach for margin classifiers", *Journal of Machine Learning Research*, 113–141, 2000.

Ferrari, S., and Stengel, R.F., "Smooth Function Approximation Using Neural Networks", In *IEEE Transactions on Neural Networks*, 16, 24-38, 2005.

Fletcher, R. and Reeves, C. M., "Function minimization by conjugate gradients", *Computer Journal*, 7, 149–154, 1964.

Fletcher, R., *Practical Methods of Optimization*, John Wiley and Sons, New York, 1981.

Ford, J.A., Narushima, Y. and Yabe, H., "Multi-step nonlinear conjugate gradient methods for unconstrained minimization", *Computational Optimization and Applications*, 40, 191–216, 2008.

Frimpong, E.A., and Okyere, P.Y., "Forecasting the Daily Peak Load of Ghana Using Radial Basis Function Neural Network and Wavelet Transform", *Journal of Electrical Engineering*, 10, 15-18, 2010.

Fukumizu, K. and Amari, S., "Local Minima and Plateaus in Multilayer Neural Networks", In *Ninth International Conference on Artificial Neural Networks*, 2, 597—602, 1999.

Gershenson, C., "Artificial Neural Networks for Beginners", In *Formal Computational Skills Teaching Package*, COGS, University of Sussex, 2001.

Gilbert, J.C. and Nocedal, J., "Global convergence properties of conjugate gradient methods for optimization," *SIAM Journal of Optimization*, 2, 21–42, 1992.

Gill, P. E., Murray, W., and Wright, M. H., *Practical Optimization*, Academic Press, New York, 1981.

Gori, M. and Tesi A., "On the problem of local minima in backpropagation", *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 14, 76-85, 1992.

Hagan, M.T., and Menhaj, M.B., "Training feedforward networks with the Marquardt algorithm", *IEEE Transactions on Neural Networks*, 5, 989-993, 1994.

Hager, W. W. and Zhang, H. "A survey of nonlinear conjugate gradient methods", Pacific Journal of Optimization, 2:35, 35–58, 2006.

Hagiwara, M., "A simple and effective method for removal of hidden units and weights", *Neuro-computing*, 6, 207‑218, 1994.

Harney, L.G.C., "Benchmarking feed-forward neural networks: models and measures", In *Advances in Neural Information Processing Systems 4*, San Mateo, CA: Morgan Kaufmann, 1167-1174, 1992.

Haykin, S., *Neural Networks: A Comprehensive Foundation*, Macmillan College Publishing Company, New York, 1994.

Hestenes, M. R. and Stiefel, E., "Methods for conjugate gradients for solving linear systems", *Journal of Research of the National Bureau of Standards*, 49, 409–436, 1952.

Ho, T.K., and Basu, M., "Complexity Measures of Supervised Classification Problems", *IEEE Transactions on Pattern Analysis and Machine Intelligence* 24, 289-300, 2002.

Horton, P. and Nakai, K., "Better prediction of protein cellular localization sites with the k nearest neighbors classifier", *Intelligent Systems for Molecular Biology*, 4, 368–383, 1996.

Hornik, K., Stinchcombe, M. and White, H., "Multilayer Feedforward Networks Are Universal Approximators", *Neural Networks*, 2, 359-366, 1989.

Hsu, C., and Lin, C., "A comparison of methods for multi-class support vector machines", *IEEE Transactions on Neural Networks*, 13, 415-425, 2002.

Igel, C. and Husken, M., "Empirical evaluation of the improved Rprop learning algorithms", *Neurocomputing*, 50, 105-123, 2003.

Ioannis, E.L. and Panagiotis, P., "An Advanced Conjugate Gradient Training Algorithm Based on a Modified Secant Equation", *ISRN, Artificial Intelligence*, 2012.

Jacobs, R., "Increased rates of convergence through learning rate adaptation", *Neural Networks*, 1 (4), 295-307, 1988.

Jonathan, R.S., "An introduction to the conjugate gradient method without the agonizing pain", *Technical Report* CMU-CS-94-125, School of Computer Science, Carnegie Mellon University, 1994.

Kazuhiro, S., "A Two Phase Method for Determining the Number of Neurons in the Hidden Layer of a 3-Layer Neural Network", *SICE Annual Conference*, 2010.

Kohavi, R., "A study of cross-validation and bootstrap for accuracy estimation and model selection," In *Proceedings of the IEEE International Joint Conference on Artificial Intelligence*, 223–228, 1995.

Lawrence, S., Giles, C. L., and Tsoi, A. C., "What size neural network gives optimal generalization? Convergence properties of backpropagation", *Technical Report*, UMIACS-TR-96-22 and CS-TR-3617, Institute for Advanced Computer Studies, Univ. of Maryland, 1996.

Li, D.H. and Fukushima, M., "A modified BFGS method and its global convergence in non-convex minimization", *Journal of Computational and Applied Mathematics*, 129, 15–35, 2001.

Li, G., Tang, C., and Wei, Z., "New conjugacy condition and related new conjugate gradient methods for unconstrained optimization", *Journal of Computational and Applied Mathematics*, 202, 523–539, 2007.

Livieris, I. E. and Pintelas, P., "Performance evaluation of descent CG methods for neural network training", In *Proceedings of the Ninth Hellenic European Research on Computer Mathematics & its Applications Conference*, 40–46, 2009.

Livni, R., Shalev-Shwartz, S., and Shamir, O., "On the computational efficiency of training neural networks", In *Advances in Neural Information Processing Systems*, 855–863, 2014.

Lodish, H., Berk, A., Zipursky, S. L., Matsudaira, P., Baltimore, D. and Darnell, J., *Molecular Cell Biology*, Freeman, New York, 2003.

Looney, C. G., *Pattern Recognition Using Neural Networks: Theory and Algorithms for Engineers and Scientists*, Oxford University Press, New York, 171-172, 1997.

Magoulas, G.D. and Vrahatis M.N., "A Class of Adaptive Learning Rate Algorithms Derived by One-Dimensional Subminimization Methods", *Neural, Parallel and Scientific Computations*, 8, 147-168, 2000.

Magoulas, G.D., Vrahatis, M.N., and Androulakis, G.S., "On the alleviation of the problem of local minima in back-propagation", *Nonlinear Analysis: Theory, Methods and Applications*, 30, 4545-4550, 1997a.

Magoulas, G.D., Vrahatis M.N., and Androulakis G.S., "Effective back-propagation with variables stepsize", *Neural Networks*, 10, 69-82, 1997b.

Magoulas, G.D., Vrahatis, M.N., and Androulakis, G.S., "Improving the Convergence of the Backpropagation Algorithm Using Learning Rate Adaptation Methods", *Neural Computation*, 11, 1769-1796, 1999.

MATLAB, R., "Version 8.1. 0.604 (R2013a)", *Natrick, Massachusetts: The MathWorks Inc*, 2013.

McCulloch, W.S. and Pitts, W., "A logical calculus of the ideas immanent in nervous activity", *Bulletin of Mathematical Biophysics*, 5, 115-133, 1943.

Moller, M.F., "A scaled conjugate gradient algorithm for fast supervised learning", *Neural Networks*, 6, 525-533, 1993.

Murray, W., "Newton-type methods", *Technical report*, Department of Management Science and Engineering, Stanford University, 2010.

Murphy, P. M. and Aha, D. W., UCI Repository of machine learning databases, Irvine, CA: University of California, Department of Information and Computer Science, 1994, http://www.ics.uci.edu/mlearn/MLRepository.html.

Myint, M.Y., Khin, S.L., and Marlar, K., "Implementation of Neural Network Based Electricity Load Forecasting", In *World Academy of Science, Engineering and Technology*, 42, 381-386, 2008.

Nguyen, D. and Widrow, B., "Improving the learning speed of 2-layer neural network by choosing initial values of adaptive weights", *Biological Cybernetics*, 59, 71–113, 1990.

Nocedal, J., "Theory of algorithms for unconstrained optimization", *Acta Numerica*, 1, 199-242, 1992.

Patnaik L.M. and Rajan, K., "Target detection through image processing and resilient propagation algorithms", *Neurocomputing*, 35, 123-135, 2000.

Plagianakos, V.P., Magoulas, G.D., and Vrahatis, M.N., "Learning in multilayer perceptrons using global optimization strategies", *Nonlinear Analysis: Theory, Methods and Applications*, 47, 3431-3436, 2001a.

Plagianakos, V.P., Magoulas, G.D., and Vrahatis, M.N., "Supervised training using global search methods", N. Hadjisavvas and P. Pardalos (eds.), *Advances in Convex Analysis and Global Optimization,, Non-convex Optimization and its Applications*, Kluwer Academic Publishers, Dordrecht, The Netherlands, 54, 421-432, 2001b.

Polak, E. and Ribi`ere, G., "Note sur la convergence de methods de directions conjuguees", *Revue Francais d'Informatique et de Recherche Operationnelle*, 16, 35–43, 1969.

Powell, M.J.D., "Restart procedures for the conjugate gradient method", *Mathematical Programming*, 12, 241–254, 1977.

Powell, M.J.D., "Non-convex minimization calculations and the conjugate gradient method", *Numerical Analysis, Lecture notes in mathematics*, Springer, Berlin, Germany, 1066, 122-141, 1984.

Powell, M.J.D., "Convergence properties of algorithms for nonlinear optimization", *Siam Review*, 28, 487–500, 1986.

Prasad, N., Singh, R., and Lal, S.P., "Comparison of Back Propagation and Resilient Propagation Algorithm for Spam Classification", In *Fifth International Conference on Computation Intelligence, Modelling and Simulation*, 29-34, 2013.

Reed, R.D., and Marks II, R.J., *Neural Smithing*, MIT Press, Cambridge, Massachusetts, 1999.

Richard, M.D., and Lippmann, R.P., "Neural network classifiers estimate Bayesian a posteriori probabilities", *Neural computation*, 3, 461-483, 1991.

Riedmiller, M., "Rprop-Description and Implementation Details", *Technical Report*, University of Karlsruhe, January, 1994.

Riedmiller, M., and Braun, H., "A direct adaptive method for faster back-propagation learning: The Rprop algorithm", *International Conference on Neural Networks*, San Francisco, CA, 586-591, 1993.

Rifkin, R., and Klautau, A., "In Defense of One-Vs-All Classification", *Journal of Machine Learning Research*, 5, 101–141, 2004.

Ripley, B. D., "Statistical aspects of neural networks", In *Networks and Chaos: Statistical and Probabilistic Aspects*, Chapman and Hall, London, 1993.

Rojas, R., "Oscillating iteration paths in neural networks learning", *Computers and Graphics*, 4, 593-597, 1994.

Rojas, R., *Neural Networks: A Systematic Introduction*, Springer, Germany, 1996.

Rosenblatt, F., "The perceptron: a probabilistic model of information storage and organisation in the brain", *Psychological Review*, 65, 386-408, 1958.

Rudner, L.M., "The Classification Accuracy of Measurement Decision Theory", *Paper presented at the annual meeting of the National Council on Measurement in Education*, Chicago, 23–25, 2003.

Rumelhart, D.E., Hinton, G.E., and Williams, R.J., "Learning internal representations by error propagation", D.E. Rumelhart, J.L. McClelland (eds.), *Parallel Distributed Processing: Explorations in the Microstructure of Cognition 1*, MIT Press, Cambridge, Massachusetts, 318-362, 1986.

Sartori, M. A. and Antsaklis, P., "A simple method to derive bounds on the size and to train multilayer neural networks", *IEEE Transactions on Neural Networks* 2, 467–471, 1991.

Saurabh, K., "Approximating number of hidden layer neurons in multiple hidden layer BPNN architecture", *International Journal of Engineering Trends and Technology*, 3, 714–717, 2012.

Scales, L.E., *Introduction to non-linear optimization*, Springer-Verlag, New York, 34-35, 1985.

Sharma, B. and Venugopalan, K., "Comparison of Neural Network Training Functions for Hematoma Classification in Brain CT Images", *IOSR Journal of Computer Engineering*, 16, 31-35, 2014.

Sheela, K.G., and Deepa S.N., "Review on methods to fix number of hidden neurons in neural networks", *Mathematical Problems in Engineering*, 11, 2013.

Stuti, A., and Rakesh, K.B., "Handwritten Multi-script Pin Code Recognition System having Multiple hidden layers using Back Propagation Neural Network", *International Journal of Electronics Communication and Computer Engineering*, 2, 1, 2011.

Taguchi, I. and Sugai, Y., "Oscillation Behaviour for the Layerd Neural Networks Based on the Selection of Training Data by Using Rastrigin Function", *International Journal of Emerging Technology and Advanced Engineering*, 2013.

Treadgold, N.K. and Gedeon, T.D., "Simulated Annealing and Weight Decay in Adaptive Learning: The SARPROP Algorithm", *IEEE Transactions on Neural Networks*, 9, 4, 662-668, 1998.

Van der Smagt, P.P., "Minimization Methods for training feed-forward neural networks", *Neural Networks*, 7, 1-11, 1994.

Veitch, A.C., and Holmes, G., "Benchmarking and fast learning in neural networks: Results for back-propagation", In *Proceedings of the Second Australian Conference on Neural Networks*, 167–171, 1991.

Wickham, H., ggplot2, *Wiley Interdisciplinary Reviews: Computational Statistics* 3, 2, 180-185, 2011.

Yabe, H. and Takano, M., "Global convergence properties of nonlinear conjugate gradient methods with modified secant condition", *Computational Optimization and Applications*, 28, 203–225, 2004.

Yuan, G. "Modified nonlinear conjugate gradient methods with sufficient descent property for large-scale optimization problems," *Optimization Letters*, 3, 11–21, 2009.

Zhang, G.P., "Neural networks for classification: a survey", *IEEE Transactions on Systems, Man, and Cybernetics, Part C (Applications and Reviews)*, 4, 451-462, 2000.

Zhang, L., "Two modified Dai-Yuan nonlinear conjugate gradient methods", *Numerical Algorithms*, 50, 1–16, 2009.

Zhang, L., Zhou, W. and D. Li., "Some descent three-term conjugate gradient methods and their global convergence", *Optimization Methods and Software*, 22, 697–711, 2007.

Zhang, L. and Zhou, W., "Two descent hybrid conjugate gradient methods for optimization", *Journal of Computational and Applied Mathematics*, 216, 251–264, 2008.

Zhou, W. and Zhang, L., "A nonlinear conjugate gradient method based on the MBFGS secant condition", *Optimization Methods and Software*, 21, 707–714, 2006.

APPENDIX A: Matlab CODE – Transforming the E.coli multiclass problem into multiple binary problems

Summary

The E.coli problem is a multiclass classification problem. For the purpose of this study, we proposed the One-Against-All approach to transform it into multiple binary classification problems. Hence, this section presents the Matlab code used to prepare the E.coli data for the binary classification tasks. As said before, the E.coli data is constituted of 8 classes. Therefore, the resulting One-Against-All binary classifiers are also 8. The transformation process is as follows.

```
%First step: Importing E.coli_data in matlab as a matrix. %Not that
prior to this step a process has been followed %for putting all the
patterns of each class next to each %other (Grouping data by class).

%Removing columns 1 and 9 from E.coli data. This is done %because
column 1 gives the Sequence Name (Accession %number for the SWISS-
PROT database), and column 9 gives %the name of the proteins for a
pattern (group of %attribute). So, these two columns are not
important for %the input data matrix. But the last one is important
for %creating the target matrix for each E.coli binary %classifier.

test_1 = ecoli(:,[2,3,4,5,6,7,8]);

%E_coli inputs matrix.

E_coli_Inputs = test_1.';  %transpose test_1



%Creating E_coli targets Matrix

targets_1 = [ones(1,143), zeros(1,193)];

targets_2 = [zeros(1,143), ones(1,77), zeros(1,116)];

targets_3 = [zeros(1,220), ones(1,52), zeros(1,64)];
```

```
targets_4 = [zeros(1,272), ones(1,35), zeros(1,29)];

targets_5 = [zeros(1,307), ones(1,20), zeros(1,9)];

targets_6 = [zeros(1,327), ones(1,5), zeros(1,4)];

targets_7 = [zeros(1,332), ones(1,2), zeros(1,2)];

targets_8 = [zeros(1,334), ones(1,2)];


%E_coli targets matrix (Concatenate all the targets_i).

%This target matrix can be used for a multiclass %classifier.

E_coli_Targets = [targets_1; targets_2; targets_3; targets_4;
targets_5; targets_6; targets_7; targets_8];


%Creating inputs and targets matrices for each binary %classifier.

%E_coli inputs matrix for each binary classifier

E_coli_Inputs;


% E_coli_Targets_cp (Targets for the cp/~cp binary %classifier)

targets_1a = [ones(1, 143), zeros(1, 193)];

targets_1b = [zeros(1, 143), ones(1, 193)];

E_coli_Targets_cp = [targets_1a; targets_1b];


%E_coli_Targets_im (Targets for the im/~im binary %classifier)

targets_2a = [zeros(1, 143), ones(1, 77), zeros(1, 116)];

targets_2b = [ones(1, 143), zeros(1, 77), ones(1, 116)];

E_coli_Targets_im = [targets_2a; targets_2b];
```

```
%E_coli_Targets_pp (Targets for the pp/~pp binary %classifier)

targets_3a = [zeros(1, 220), ones(1, 52), zeros(1, 64)];

targets_3b = [ones(1, 220), zeros(1, 52), ones(1, 64)];

E_coli_Targets_pp = [targets_3a; targets_3b];


%E_coli_Targets_imU (Targets for the imU/~imU binary %classifier)

targets_4a = [zeros(1, 272), ones(1, 35), zeros(1, 29)];

targets_4b = [ones(1, 272), zeros(1, 35), ones(1, 29)];

E_coli_Targets_imU = [targets_4a; targets_4b];


%E_coli_Targets_om (Targets for the om/~om binary %classifier)

targets_5a = [zeros(1, 307), ones(1, 20), zeros(1, 9)];

targets_5b = [ones(1, 307), zeros(1, 20), ones(1, 9)];

E_coli_Targets_om = [targets_5a; targets_5b];


%E_coli_Targets_omL (Targets for the omL/~omL binary %classifier)

targets_6a = [zeros(1, 327), ones(1, 5), zeros(1, 4)];

targets_6b = [ones(1, 327), zeros(1, 5), ones(1, 4)];

E_coli_Targets_omL = [targets_6a; targets_6b];


%E_coli_Targets_imL (Targets for the imL/~imL binary %classifier)

targets_7a = [zeros(1, 332), ones(1, 2), zeros(1, 2)];

targets_7b = [ones(1, 332), zeros(1, 2), ones(1, 2)];

E_coli_Targets_imL = [targets_7a; targets_7b];
```

```
%E_coli_Targets_imS (Targets for the imS/~imS binary %classifier)

targets_8a = [zeros(1, 334), ones(1, 2)];

targets_8b = [ones(1, 334), zeros(1, 2)];

E_coli_Targets_imS = [targets_8a; targets_8b];
```

APPENDIX B: Matlab CODE – Transforming the Yeast multiclass problem into multiple binary problems

Summary

As for E.coli, the Yeast problem is also a multiclass classification problem. We applied the same approach to create the binary classification problems. This section gives the Matlab code used for that. The Yeast data is constituted of 10 classes. Therefore, the resulting One-Against-All binary classifiers are also 10. The transformation process is as follows.

```
%First step: Importing Yeast_data in matlab as a matrix. %As for the
E.coli, not that prior to this step, a process has been done of
putting all the patterns of each class %next to each other (Grouping
data by class).

%Remove columns 1 and 10 from Yeast data. This is done because
column 1 gives the Sequence Name (Accession %number for the SWISS-
PROT database), and column 10 gives the name of the proteins for a
pattern (group of %attribute). So these two columns are not
important for %the input data matrix. But the last one is important
%for creating the target matrix for every Yeast binary.

test_1 = Yeast(:,[2,3,4,5,6,7,8,9]);


%Yeast inputs matrix.

Yeast_Inputs = test_1.'; %transpose test_1


%Creating Yeast_targets matrix

targets_1 = [ones(1, 463), zeros(1, 1021)];

targets_2 = [zeros(1, 463), ones(1, 429), zeros(1, 592)];
```

```
targets_3 = [zeros(1, 892), ones(1, 244), zeros(1, 348)];

targets_4 = [zeros(1, 1136), ones(1, 163), zeros(1, 185)];

targets_5 = [zeros(1, 1299), ones(1, 51), zeros(1, 134)];

targets_6 = [zeros(1, 1350), ones(1, 44), zeros(1, 90)];

targets_7 = [zeros(1, 1394), ones(1, 35), zeros(1, 55)];

targets_8 = [zeros(1, 1429), ones(1, 30), zeros(1, 25)];

targets_9 = [zeros(1, 1459), ones(1, 20), zeros(1, 5)];

targets_10 = [zeros(1, 1479), ones(1, 5)];


%Yeast targets matrix (Concatenate all the targets_i)

%This target matrix can be used for a multiclass classifier.

Yeast_Targets = [targets_1; targets_2; targets_3; targets_4;
targets_5; targets_6; targets_7; targets_8; targets_9; targets_10];


%Creating inputs and targets matrices for each binary classifier.

%Yeast inputs matrix for each binary classifier

Yeast_Inputs;


%Yeast_Targets_CYT (Targets for the CYT/~CYT binary classifier)

targets_1a = [ones(1, 463), zeros(1, 1021)];

targets_1b = [zeros(1, 463), ones(1, 1021)];

Yeast_Targets_CYT = [targets_1a; targets_1b];


%Yeast_Targets_NUC (Targets for the NUC/~NUC binary classifier)

targets_2a = [zeros(1, 463), ones(1, 429), zeros(1,592)];
```

```
targets_2b = [ones(1, 463), zeros(1, 429), ones(1, 592)];

Yeast_Targets_NUC = [targets_2a; targets_2b];


%Yeast_Targets_MIT (Targets for the MIT/~MIT binary classifier)

targets_3a = [zeros(1, 892), ones(1, 244), zeros(1,348)];

targets_3b = [ones(1, 892), zeros(1, 244), ones(1, 348)];

Yeast_Targets_MIT = [targets_3a; targets_3b];


%Yeast_Targets_ME3 (Targets for the ME3/~ME3 binary classifier)

targets_4a = [zeros(1, 1136), ones(1, 163), zeros(1,… 185)];

targets_4b = [ones(1, 1136), zeros(1, 163), ones(1,185)];

Yeast_Targets_ME3 = [targets_4a; targets_4b];


%Yeast_Targets_ME2 (Targets for the ME2/~ME2 binary classifier)

targets_5a = [zeros(1, 1299), ones(1, 51), zeros(1,134)];

targets_5b = [ones(1, 1299), zeros(1, 51), ones(1, 134)];

Yeast_Targets_ME2 = [targets_5a; targets_5b];


%Yeast_Targets_ME1 (Targets for the ME1/~ME1 binary classifier)

targets_6a = [zeros(1, 1350), ones(1, 44), zeros(1, 90)];

targets_6b = [ones(1, 1350), zeros(1, 44), ones(1, 90)];

Yeast_Targets_ME1 = [targets_6a; targets_6b];


%Yeast_Targets_EXC (Targets for the EXC/~EXC binary classifier)

targets_7a = [zeros(1, 1394), ones(1, 35), zeros(1, 55)];
```

```
targets_7b = [ones(1, 1394), zeros(1, 35), ones(1, 55)];

Yeast_Targets_EXC = [targets_7a; targets_7b];


%Yeast_Targets_VAC (Targets for the VAC/~VAC binary classifier)

targets_8a = [zeros(1, 1429), ones(1, 30), zeros(1, 25)];

targets_8b = [ones(1, 1429), zeros(1, 30), ones(1, 25)];

Yeast_Targets_VAC = [targets_8a; targets_8b];


%Yeast_Targets_POX (Targets for the POX/~POX binary classifier)

targets_9a = [zeros(1, 1459), ones(1, 20), zeros(1, 5)];

targets_9b = [ones(1, 1459), zeros(1, 20), ones(1, 5)];

Yeast_Targets_POX = [targets_9a; targets_9b];


%Yeast_Targets_ERL (Targets for the ERL/~ERL binary classifier)

targets_10a = [zeros(1, 1479), ones(1, 5)];

targets_10b = [ones(1, 1479), zeros(1, 5)];

Yeast_Targets_ERL = [targets_10a; targets_10b];
```

APPENDIX C: Matlab CODE – Training process of classifiers

Summary

This section presents the Matlab code implemented for the training of the binary classifiers. Given that the steps involved in the training of every binary classifier are similar, the following code is an example for training one binary classifier only. For training all the binary classifiers, this process must be repeated 18 times. Furthermore, this process is for only one experimental design. Appropriate changes must be made to the various hidden nodes and training iterations to accommodate it for every experiment. The CYT/~CYT binary classifier is used for the purpose of this presentation. Also, the training design used is as follows: combinations of 15, 20, 25, 30, and 35 hidden nodes, with 500, 1000, 2000, 3000, 3500, and 4000 training iterations.

Notations used

The following notations are used in the presentation of the code:

N: The total number of repetitions for every training design

H: Vector of various numbers of hidden nodes h

T: Vector of various numbers of training iterations t

OAtrain: Overall Accuracy on training set

OAtest: Overall Accuracy on test set

Time: Overall Total Training time. In brief, this stands for the training time used in the presentation of the results in chapter 5.

MSE: Mean Squared Error. In brief, this stands for the convergence used in the presentation of the results in chapter 5.

Atrain1: Training Accuracy of class 1 or Accuracy of class 1 on the training set

Atrain2: Training Accuracy of class 2 or Accuracy of class 1 on training set

Atest1: Test Accuracy of class 1 or Accuracy of class 1 on the test set

Atest2: Test Accuracy of class 1 or Accuracy of class 2 on the test set

N_ht_Performance: Array containing results for the N training repetitions from ht training scenario. For instance, N_ht_OAtrain stands for OAtrain for training scenario ht, and for N training tasks. The same applies for every performance measure.

The whole process implementation can be summarised in the following steps:

Step 1: Process to get the performance measure estimates.

Set 50 as the number of trials for each training task of classifiers;

Set 10 as the number of folds for applying the 10-fold CV;

Initialise 4 three dimensional arrays for storing the 4 performance measures (Time, MSE, OAtrain, OAtest). The three dimensions of an array are $5 \times 6 \times 50$; where 5, 6 and 50 are the first, second and third dimensions respectively.

The first and second dimensions of an array constitute a 5x6 matrix, where 5 is the dimension of the various numbers hidden nodes (hn) and 6 is the dimension of the various maximum numbers of training iterations. The last (third) dimension is of size 50, representing the 50 different trials. This means that for instance, the first three dimensional array is for storing the 50 different 5x6 matrices of times. The 50 different 5x6 matrices of the MSE, OAtrain and OAtest are to be stored in the second, third and last three dimensional arrays, respectively.

For $n = 1{:}\,50$ {

$H = [5, 10, 20, 30, 40]$; This is a vector of various numbers of hidden nodes

For $h = H$ {

$T = [25,50,75,100,150,200]$; This is a vector of various numbers iterations;

Read the input matrix (i.e. data matrix of features to be classified);

Read the target matrix (i.e. data matrix of the desired output classes for the input matrix features);

Specify the neural network topology to be used. For our experiments, the topology is the feedforward network. The corresponding notation in matlab is as follows:

$net = feedforwardnet(h)$;

Where h is the number of hidden nodes to be used.

Specify the remaining parameters of the network;

For $t = T$ {

Split the data in 10 different folds to apply cross validation. This is done as follows:

Read $K = 10$ as the total number of folds;

Assign an index to each case in the input matrix. The total number of indices corresponds to the size of the input matrix. For instance, the number of indices is 336 for the E.coli data.

Permute these indices;

Split the indices in $K = 10$ folds;

For $k = 1{:}K$ {

Do training and test on the K folds as follows:

Select subsequently each training set (fold);

Select subsequently each test set (fold);

Specify the training functions and parameters;

Specify the performance measures;

Train and test a classifier by subsequently using the *K* training and *K* test sets (folds);

Extract the confusion matrices from the *K* training and *K* test results;

Store the performance results for the *K* trainings and tests in four different vectors of size *K* each. For a particular performance measure, this can be done as follows:

$$Performance(k) = Performance;$$

}

Calculate the average performance over the *K* folds for each performance measure (i.e. arithmetic mean of each of the four vectors obtained above). This will give four averages (i.e., arithmetic means), each for one performance measure. Notation for a particular average performance can be as follows:

$$Mean\_Performance = mean(Performance(k));$$

Store the performance measures for every training scenario in a two dimensional array (or matrix) of dimensions $h \times t$. This step will give four matrices, each for one performance measure. For example, storage of a performance in a matrix can be done as follows:

$$Matrix(h,t) = Mean\_Performance;$$

}

}

Subset the four matrices of performance to make sure that their rows and columns correspond to the specified hidden nodes and training iterations. For instance, for one matrix this is done as follows:

$Matrix = Matrix([5,10,20,30,40],[25,50,75,100,150,200]);$

Notice that if a matrix is not subset as showed above, it will have 40 rows and 200 columns instead of 5 and 6 respectively.

Store the subset matrices of performance in the four 3 dimensional arrays initialised at the beginning of our process. For example, a matrix can be stored in a three dimensional array as follows:

$Array(:,:,n) = Matrix;$

Array and Matrix should be named after a particular performance measure they represent.

}

Step 2: Process of aggregating the performance measure estimates stored in the four 3 dimensional arrays.

This is done by computing the matrix average (mean of 50 matrices stored in every 3 dimensional array) for each performance measure. This process is described as follows:

Initialise four arrays of matrix sums. Four one array this can be done as follows:

$Array\_of\_Sums = 0;$

Compute the arrays of sums using for loop as follows:

For $n = 1{:}50$ {

The four arrays of sums are calculated here. For instance, the process for one array can be represented as follows:

$Array\_of\_Sums = Array\_of\_Sums + Array(:,:,n);$

}

Compute the matrix averages from each array of matrix sums produced by the above for loop. Four one matrix of averages the notation can be as follows:

$Matrix\_of\_Averages = Array\_of\_Sums\ /50;$

Step 3: Process of finding the best binary classifier based on a particular performance measure.

This process involves extracting the best performance measure from each matrix average above. Depending on whether we want to find the minimum or the maximum for a particular performance measure, this can be done as follows:

$Best\_Performance = \min(\min(Matrix\_of\_Averages));$

Or,

$Best\_Performance = \max(\max(Matrix\_of\_Averages));$

Step 4: Process of finding the best training scenario (combination of number of hidden nodes and number of training iterations) for each performance measure. Also, this process involves finding the trade-off between the four performance measures.

The best training scenario is the one that produced the best performance. This process is described for the four performance measures as follows:

- Find row (number of hidden nodes) and column (number of training iterations) from $Matrix\_of\_Averages$ of Times, corresponding to the best Time. The scenario (hn_t) found is the one that produced the shortest training time. For this scenario, find the corresponding MSE, OAtrain, and OAtest.
- Find row and column from $Matrix\_of\_Averages$ of MSEs, corresponding to the best MSE. For this scenario, find the corresponding Time, OAtrain, and OAtest.
- Find row and column from $Matrix\_of\_Averages$ of OAtrains, corresponding to the best OAtrain. For this scenario, find the corresponding Time, MSE, OAtest.

- Find row and column from $Matrix\_of\_Averages$ of OAtests, corresponding to the best OAtest. For this scenario, find the corresponding Time, MSE, and OAtrain.

Step 5: Write results onto output file

This step involves sending the results to an external file to save them. It is worth emphasising that training of the binary classifiers is done in one run and subsequently one of another. So if this step is not implemented, results of the previous classifiers will be lost while those of the subsequent ones are being generated.

Step 6: Repeat step 1 to step 5 for all the binary classifiers

```
%The following codes are given as an example of how to
traingaclassifier
%We use CYT/~CYT binary classifier to illustrate the process. For
other
%binary classifiers, one will need to change the input and target
matrices
%accordingly

%Set:
N = 50; %Number of trials for the all training process of
classifiers
K = 10; %Number of folds for applying the k-fold CV

%Set 4 three dimensional arrays for storing the Time, MSE, OAtrain
and
%OAtest as follows:
N_ht_M_Time_subset(5,6,N) = 0;
N_ht_M_MSE_subset(5,6,N) = 0;
N_ht_M_OAtrain_subset(5,6,N) = 0;
N_ht_M_OAtest_subset(5,6,N) = 0;

%Set 4 three dimensional arrays for storing the Atrain1, Atrain2,
Atest1
```

```
%and Atest2
N_ht_M_Atrain1_subset(5,6,N) = 0;
N_ht_M_Atrain2_subset(5,6,N) = 0;
N_ht_M_Atest1_subset(5,6,N) = 0;
N_ht_M_Atest2_subset(5,6,N) = 0;

%Step 1: Implementation of the four nested for loops with the k-fold
cross
%validation as the innermost for loop.

%Starting the outer for loop for the total number of repetitions N
for n = 1:N
    %Vector of the 5 various numbers of hidden nodes.
    %Vector of the 6 various numbers training iterations.
H = [5,10,20,30,40];
T = [25,50,75,100,150,200];

%Starting the second for loop for the various hidden nodes h.
for h = H
    inputs = Yeast_Inputs; %Assigning the input matrix to the
process.
    targets = Yeast_Targets_CYT; %Assign the target matrix to the
process.
    net = feedforwardnet(h);
    net.inputs{1}.processFcns = {'removeconstantrows','mapminmax'};
    net.outputs{2}.processFcns = {'removeconstantrows','mapminmax'};
    net.layers{1}.transferFcn = 'tansig';
    net.layers{2}.transferFcn = 'tansig';
    net.layers{1}.initFcn = 'initnw';
    net.layers{2}.initFcn = 'initnw';

    %Starting the third for loop for the various training iterations
t.
    for t = T

        %Creating the indices Q for the training and test sets to
use in
        %the Cross Validation process.
        Q = size(inputs,2);
```

```
        ind = randperm(Q); %Random permutation Q, which gives
indices ind.
        cvFolds = crossvalind('Kfold',ind,K);

        %Starting the most inner for loop for 10-fold Cross
Validation.
        for k = 1:K
            testIdx = (cvFolds == k);
            trainIdx = ~testIdx;
            trInd = find(trainIdx);
            tstInd = find(testIdx);

            net.initFcn = 'initlay';
            net.trainFcn = 'trainrp';
            net.trainParam.epochs = t;
            net.trainParam.goal = 0;
            net.trainParam.max_fail = 5000; %5000 is the stopping
criteria,
            %which will never be reached.
            net.divideFcn = 'divideind';
            net.divideParam.trainInd = trInd;
            net.divideParam.testInd = tstInd;

            net.performFcn = 'mse';

            %Initializing and Training the Network.
            net = init(net);
            [net,tr] = train(net,inputs,targets);
            outputs = net(inputs);
            errors = gsubtract(targets,outputs);

            %---------------------Time------------------------------
            Time(k).time = tr.time;
            %Train_time(k) = mean(Time(k).time);
            MeTime(k) = mean(Time(k).time); %Mean training time
(training
            %time for one iteration (epochs) in an entire training
process.
            Time(k) = sum(Time(k).time);  %Total training time
```

```
            %(training time for all iterations (epochs) in an entire
            %training process).

            %----------------------MSE-----------------------------
            MSE(k) = tr.best_perf; %Minimun error reached during
training.


            %-------------------------Train-------------------------
            %Training inputs for each partition of CV.
            %Training targets for each partition of CV.
            %Training outputs for each partition of CV.
            train_inputs = inputs(:,trInd);
            train_targets = targets(:,trInd);
            train_outputs = net(train_inputs);

            %Misclassification rate (c_tr) and confusion matrix
(cm_tr) on
            %training.
            [c_tr,cm_tr] = confusion(train_targets,train_outputs);

            %Classification accuracy on training.
            OAtrain(k) = 100*sum(diag(cm_tr))/sum(sum(cm_tr));

            %Classification accuracy on training for class1 and
class2
            diag_cm_tr = diag(cm_tr);
            Atrain1(k) = 100*diag_cm_tr([1])/sum(cm_tr(1,:));
            Atrain2(k) = 100*diag_cm_tr([2])/sum(cm_tr(2,:));

            %----------------------Test----------------------------
            %Test inputs for each partition of CV.
            %Test targets for each partition of CV.
            %Test outputs for each partition of CV.
            test_inputs = inputs(:,tstInd);
            test_targets = targets(:,tstInd);
            test_outputs = net(test_inputs);
```

```
            %Misclassification rate (c_tst) and confusion matrix
(cm_tst)
            %on testing.
            [c_tst,cm_tst] = confusion(test_targets,test_outputs);

            %Classification accuracy on testing.
            OAtest(k) = 100*sum(diag(cm_tst))/sum(sum(cm_tst));

            %Classification accuracy on testing for class1 and
class2
            diag_cm_tst = diag(cm_tst);
            Atest1(k) = 100*diag_cm_tst([1])/sum(cm_tst(1,:));
            Atest2(k) = 100*diag_cm_tst([2])/sum(cm_tst(2,:));
        end

        %----------------------Training time-----------------------
        OMeTrain_time_1 = mean(MeTime);  %Overall mean training
time.
        Mean_Time = sum(Time);  %Overall total training time
        %for the 10 subsets of cross validation.

        %Overall total training time for all network
architectures(Asso-
        %ciations of number of nodes (h) with number of iterations
(t).
        ht_M_Time(h,t) = Mean_Time;

        %--------------------------Training accuracies--------------
        %Overall classification accuracy on training.
        Mean_OAtrain = mean(OAtrain);

        %Overall classification accuracy on training for all
scenarios.
        ht_M_OAtrain(h, t) = Mean_OAtrain;

        %Accuracies for class1 and class2
        Mean_Atrain1 = mean(Atrain1);
        Mean_Atrain2 = mean(Atrain2);
        ht_M_Atrain1(h, t) = Mean_Atrain1;
```

```
        ht_M_Atrain2(h, t) = Mean_Atrain2;

        %---------------------------Test accuracies-----------------
        %Overall classification accuracy on testing set
        Mean_OAtest = mean(OAtest);

        %Overall classification accuracy on testing for all
scenarios
        ht_M_OAtest(h, t) = Mean_OAtest;

        %Accuracies for class1 and class2
        Mean_Atest1 = mean(Atest1);
        Mean_Atest2 = mean(Atest2);
        ht_M_Atest1(h, t) = Mean_Atest1;
        ht_M_Atest2(h, t) = Mean_Atest2;

        %------------------------MSE--------------------------------
        %Overall minimun error reached during training.
        Mean_MSE = mean(MSE);
        %storing OBTrainPerf for all training scenarios.
        ht_M_MSE(h,t) = Mean_MSE;
    end
end

%------------Subsetting the matrices of the performance measures----
%This process makes sure that the matrices to be stored are of the
correct
%dimensions, i.e. 5 rows (various h) and 6 columns (various t).
ht_M_Time_subset = ht_M_Time([5,10,20,30,40],...
    [25,50,75,100,150,200]);
ht_M_MSE_subset = ht_M_MSE([5,10,20,30,40],...
    [25,50,75,100,150,200]);
ht_M_OAtrain_subset = ht_M_OAtrain([5,10,20,30,40],...
    [25,50,75,100,150,200]);
ht_M_OAtest_subset = ht_M_OAtest([5,10,20,30,40],...
    [25,50,75,100,150,200]);

ht_M_Atrain1_subset = ht_M_Atrain1([5,10,20,30,40],...
    [25,50,75,100,150,200]);
```

```
ht_M_Atrain2_subset = ht_M_Atrain2([5,10,20,30,40],...
    [25,50,75,100,150,200]);
ht_M_Atest1_subset = ht_M_Atest1([5,10,20,30,40],...
    [25,50,75,100,150,200]);
ht_M_Atest2_subset = ht_M_Atest2([5,10,20,30,40],...
    [25,50,75,100,150,200]);

%------------Storing the matrices of performance measures-----------
%Here, these matrices are stored in the three-dimensional arrays in
order
%to keep record of the results for all the N=50 repetitions of the
training
%tasks.
N_ht_M_Time_subset(:,:,n) = ht_M_Time_subset;
N_ht_M_MSE_subset(:,:,n) = ht_M_MSE_subset;
N_ht_M_OAtrain_subset(:,:,n) = ht_M_OAtrain_subset;
N_ht_M_OAtest_subset(:,:,n) = ht_M_OAtest_subset;

N_ht_M_Atrain1_subset(:,:,n) = ht_M_Atrain1_subset;
N_ht_M_Atrain2_subset(:,:,n) = ht_M_Atrain2_subset;
N_ht_M_Atest1_subset(:,:,n) = ht_M_Atest1_subset;
N_ht_M_Atest2_subset(:,:,n) = ht_M_Atest2_subset;
end

%-------------------Process of aggregating the results--------------
%Step 2: implementation of the process of aggregating the
performances
%contained in the four 3 dimensional arrays.

%All the N=50 training process are done with all the results stored
in the
%different three-dimensiol arrays. Now the process of aggregating
the
%results can start, i.e. computing the matrix sum and mean for each
%performance measure.

%Initialising the sum matrices.
Sum_N_ht_M_Time_subset = 0;
Sum_N_ht_M_MSE_subset = 0;
```

```
Sum_N_ht_M_OAtrain_subset = 0;
Sum_N_ht_M_OAtest_subset = 0;

Sum_N_ht_M_Atrain1_subset = 0;
Sum_N_ht_M_Atrain2_subset = 0;
Sum_N_ht_M_Atest1_subset = 0;
Sum_N_ht_M_Atest2_subset = 0;

%Applying the for loop to compute the sum and mean matrices for the
%performance measures.

%Computing the Sum
for n =1:N;
Sum_N_ht_M_Time_subset = ...
    Sum_N_ht_M_Time_subset + N_ht_M_Time_subset(:,:,n);
Sum_N_ht_M_MSE_subset = ...
    Sum_N_ht_M_MSE_subset + N_ht_M_MSE_subset(:,:,n);
Sum_N_ht_M_OAtrain_subset = ...
    Sum_N_ht_M_OAtrain_subset + N_ht_M_OAtrain_subset(:,:,n);
Sum_N_ht_M_OAtest_subset = ...
    Sum_N_ht_M_OAtest_subset + N_ht_M_OAtest_subset(:,:,n);

Sum_N_ht_M_Atrain1_subset = ...
    Sum_N_ht_M_Atrain1_subset+N_ht_M_Atrain1_subset(:,:,n);
Sum_N_ht_M_Atrain2_subset = ...
    Sum_N_ht_M_Atrain2_subset+N_ht_M_Atrain2_subset(:,:,n);
Sum_N_ht_M_Atest1_subset = ...
    Sum_N_ht_M_Atest1_subset+N_ht_M_Atest1_subset(:,:,n);
Sum_N_ht_M_Atest2_subset = ...
    Sum_N_ht_M_Atest2_subset+N_ht_M_Atest2_subset(:,:,n);
end

%Computing the Mean
Mean_N_ht_M_Time_subset = Sum_N_ht_M_Time_subset/N;
Mean_N_ht_M_MSE_subset = Sum_N_ht_M_MSE_subset/N;
Mean_N_ht_M_OAtrain_subset = Sum_N_ht_M_OAtrain_subset/N;
Mean_N_ht_M_OAtest_subset = Sum_N_ht_M_OAtest_subset/N;

Mean_N_ht_M_Atrain1_subset = Sum_N_ht_M_Atrain1_subset/N;
```

```
Mean_N_ht_M_Atrain2_subset = Sum_N_ht_M_Atrain2_subset/N;
Mean_N_ht_M_Atest1_subset = Sum_N_ht_M_Atest1_subset/N;
Mean_N_ht_M_Atest2_subset = Sum_N_ht_M_Atest2_subset/N;

%--------Rounding off the results to specified number of decimals---
%The mean matrices have been calculated; they are 8 in tolal,
corresponding
%to the 8 performance measures. Now we round them off to the number
of
%decimals as follows: 0 for training time, 4 for MSE, 2 for
%classification accuracies.
Roundn_Mean_N_ht_M_Time_subset = roundn(Mean_N_ht_M_Time_subset, 0);
Roundn_Mean_N_ht_M_MSE_subset = roundn(Mean_N_ht_M_MSE_subset, -4);
Roundn_Mean_N_ht_M_OAtrain_subset =
roundn(Mean_N_ht_M_OAtrain_subset, -2);
Roundn_Mean_N_ht_M_OAtest_subset = roundn(Mean_N_ht_M_OAtest_subset,
-2);

Roundn_Mean_N_ht_M_Atrain1_subset =
roundn(Mean_N_ht_M_Atrain1_subset, -2);
Roundn_Mean_N_ht_M_Atrain2_subset =
roundn(Mean_N_ht_M_Atrain2_subset, -2);
Roundn_Mean_N_ht_M_Atest1_subset = roundn(Mean_N_ht_M_Atest1_subset,
-2);
Roundn_Mean_N_ht_M_Atest2_subset = roundn(Mean_N_ht_M_Atest2_subset,
-2);

%---Finding the optimal (best) value for each performance measure---

%Step 3: the process of finding the best binary classifier based on
a
%particular performance measure.

%Obviously the optimal values are, the minimum for training time and
MSE,
%and the maximum for the accuracies. These are found from the round
off
%matrices.
Min_Roundn_Mean_N_ht_M_Time_subset = ...
```

```
    min(min(Roundn_Mean_N_ht_M_Time_subset));
Min_Roundn_Mean_N_ht_M_MSE_subset = ...
    min(min(Roundn_Mean_N_ht_M_MSE_subset));
Max_Roundn_Mean_N_ht_M_OAtrain_subset = ...
    max(max(Roundn_Mean_N_ht_M_OAtrain_subset));
Max_Roundn_Mean_N_ht_M_OAtest_subset = ...
    max(max(Roundn_Mean_N_ht_M_OAtest_subset));

Max_Roundn_Mean_N_ht_M_Atrain1_subset = ...
    max(max(Roundn_Mean_N_ht_M_Atrain1_subset));
Max_Roundn_Mean_N_ht_M_Atrain2_subset = ...
    max(max(Roundn_Mean_N_ht_M_Atrain2_subset));
Max_Roundn_Mean_N_ht_M_Atest1_subset = ...
    max(max(Roundn_Mean_N_ht_M_Atest1_subset));
Max_Roundn_Mean_N_ht_M_Atest2_subset = ...
    max(max(Roundn_Mean_N_ht_M_Atest2_subset));

%--------Saving the performance Matrices for a binary classifier----
%Since the training of the binary classiers is done subsequently one
after
%onther, results for each classifier should be directly saved in an
%external file in other not to lose them. Here we show how to save
results
%in Excel file for the CYT/~CYT binary classifier.

%Open Directory
Directory = 'C:\Users\1SavingCYTrp5to40H25to200T.xls'; %Specify your
Directory
%Saving matrices
xlswrite(Directory, Roundn_Mean_N_ht_M_Time_subset, 'Sheet1', 'A1')
xlswrite(Directory, Roundn_Mean_N_ht_M_MSE_subset, 'Sheet1', 'A7')
xlswrite(Directory, Roundn_Mean_N_ht_M_OAtrain_subset, 'Sheet1',
'A13')
xlswrite(Directory, Roundn_Mean_N_ht_M_OAtest_subset, 'Sheet1',
'A19')

xlswrite(Directory, Roundn_Mean_N_ht_M_Atrain1_subset, 'Sheet1',
'A25')
```

```
xlswrite(Directory, Roundn_Mean_N_ht_M_Atrain2_subset, 'Sheet1',
'A31')
xlswrite(Directory, Roundn_Mean_N_ht_M_Atest1_subset, 'Sheet1',
'A37')
xlswrite(Directory, Roundn_Mean_N_ht_M_Atest2_subset, 'Sheet1',
'A43')
%Saving best performance from each matrix
xlswrite(Directory, Min_Roundn_Mean_N_ht_M_Time_subset, 'Sheet1',
'A45')
xlswrite(Directory, Min_Roundn_Mean_N_ht_M_MSE_subset, 'Sheet1',
'A46')
xlswrite(Directory, Max_Roundn_Mean_N_ht_M_OAtrain_subset, 'Sheet1',
'A47')
xlswrite(Directory, Max_Roundn_Mean_N_ht_M_OAtest_subset, 'Sheet1',
'A48')

xlswrite(Directory, Max_Roundn_Mean_N_ht_M_Atrain1_subset, 'Sheet1',
'A50')
xlswrite(Directory, Max_Roundn_Mean_N_ht_M_Atrain2_subset, 'Sheet1',
'A51')
xlswrite(Directory, Max_Roundn_Mean_N_ht_M_Atest1_subset, 'Sheet1',
'A52')
xlswrite(Directory, Max_Roundn_Mean_N_ht_M_Atest2_subset, 'Sheet1',
'A53')


%-Finding of best classifier based on a particular performance
%measure

%Step 4: the process of finding the best training scenario
(combination of
%number of hidden nodes and number of training iterations) for each
%performance measure. Also, this process involves find the trade-off
%between the four performance measures.

%The idea here is to find a classifier with the best performance on
a
%particular performance measure, the corresponding training scenario
and
```

```
%other measures. For instance, if the best classifier is based on
OAtest,
%then its corresponding scenario is found. For this scenario, the
OAtrain,
%time, MSE, and so forth are found.

%-------Optimal classifier based on Atrain1 (train_Acc_class1)------
%find scenario h_t (row and column) of Max Atrain1
[row,col] = find(Roundn_Mean_N_ht_M_Atrain1_subset == ...
    max(max(Roundn_Mean_N_ht_M_Atrain1_subset)));
index = [row,col];   %save row and column as index
%create a linear index
Linear_idx = sub2ind(size(Roundn_Mean_N_ht_M_Atrain1_subset),
row,col);
%for that index, find the corresponding Atest1,
Roundn_Mean_N_ht_M_Atest1_subset(Linear_idx);

%create a cell array of 11 times 9 dimension. This is store results
for
%every classifier. This example is for the Yeast binary classifiers.
The 11
%correspond to 10 Yeast classifiers + 1 row for heading. The 9
correspond
%to the 8 performance measures + 1 column for scenarion.

%Notice that we use smaller arrays to illustrate the process. But
for
%results of the 10 Yeast classifiers and all the performance
measures use
%arrays as specified above. For the E.coli classifier, specify the
%dimension of arrays accordingly

%The following is an example of one can implement the process
Optimal_train_Acc_class1 = cell(4, 4);
Optimal_train_Acc_class1{1,1} = 'Binary classifiers';
Optimal_train_Acc_class1{2,1} = 'CYT';
Optimal_train_Acc_class1{3,1} = 'ERL';

Optimal_train_Acc_class1{1,2} = 'Class1Atrain';
```

```
Optimal_train_Acc_class1{1,3} = 'Class1Atest';
Optimal_train_Acc_class1{1,4} = 'Architecture';

Optimal_train_Acc_class1{2,2} =
Max_Roundn_Mean_N_ht_M_Atrain1_subset;
Optimal_train_Acc_class1{2,3} =
Roundn_Mean_N_ht_M_Atest1_subset(Linear_idx);
Optimal_train_Acc_class1{2,4} = Linear_idx;

%--------------From same process as above is repeted---------------
%------------Optimal classifier based on Atest1--------------------
%find row h and column t of the maximum of matrix of Atest1
[row,col] = find(Roundn_Mean_N_ht_M_Atest1_subset ==...
    max(max(Roundn_Mean_N_ht_M_Atest1_subset)));
index = [row,col];   %save row and column as index
Linear_idx = sub2ind(size(Roundn_Mean_N_ht_M_Atest1_subset),
row,col);
Roundn_Mean_N_ht_M_Atrain1_subset(Linear_idx);

Optimal_test_Acc_class1 = cell(4, 4);
Optimal_test_Acc_class1{1,1} = 'Binary classifiers';
Optimal_test_Acc_class1{2,1} = 'CYT';
Optimal_test_Acc_class1{3,1} = 'ERL';

Optimal_test_Acc_class1{1,2} = 'Class1Atrain';
Optimal_test_Acc_class1{1,3} = 'Class1Atest';
Optimal_test_Acc_class1{1,4} = 'Architecture';

Optimal_test_Acc_class1{2,2} =
Roundn_Mean_N_ht_M_Atrain1_subset(Linear_idx);
Optimal_test_Acc_class1{2,3} = Max_Roundn_Mean_N_ht_M_Atest1_subset;
Optimal_test_Acc_class1{2,4} = Linear_idx;

%-----------------Optimal classifier based on OAtest---------------
Optimal_OAtest = cell(4, 6); %create a cell array of 11 times 6
dimension
Optimal_OAtest{1,1} = 'Binary classifiers';
Optimal_OAtest{2,1} = 'CYT';
Optimal_OAtest{3,1} = 'ERL';
```

```
Optimal_OAtest{1,2} = 'OAtrain';
Optimal_OAtest{1,3} = 'OAtest';
Optimal_OAtest{1,4} = 'MSE';
Optimal_OAtest{1,5} = 'Time';
Optimal_OAtest{1,6} = 'Scenario';

[row,col] = find(Roundn_Mean_N_ht_M_OAtest_subset == ...
    max(max(Roundn_Mean_N_ht_M_OAtest_subset)));
index = [row,col];   %save row and column as index
Linear_idx = sub2ind(size(Roundn_Mean_N_ht_M_OAtest_subset),
row,col);
Roundn_Mean_N_ht_M_OAtrain_subset(Linear_idx);
Roundn_Mean_N_ht_M_MSE_subset(Linear_idx);
Roundn_Mean_N_ht_M_Time_subset(Linear_idx);

Optimal_OAtest{2,2} = Roundn_Mean_N_ht_M_OAtrain_subset(Linear_idx);
Optimal_OAtest{2,3} = Max_Roundn_Mean_N_ht_M_OAtest_subset;
Optimal_OAtest{2,4} = Roundn_Mean_N_ht_M_MSE_subset(Linear_idx);
Optimal_OAtest{2,5} = Roundn_Mean_N_ht_M_Time_subset(Linear_idx);
Optimal_OAtest{2,6} = Linear_idx;
%-------Optimal classifier based on OAtrain-------------------------
Optimal_OAtrain = cell(4, 6); %create a cell array of 11 times 6
dimension
Optimal_OAtrain{1,1} = 'Binary classifiers';
Optimal_OAtrain{2,1} = 'CYT';
Optimal_OAtrain{3,1} = 'ERL';

Optimal_OAtrain{1,2} = 'OAtrain';
Optimal_OAtrain{1,3} = 'OAtest';
Optimal_OAtrain{1,4} = 'MSE';
Optimal_OAtrain{1,5} = 'Time';
Optimal_OAtrain{1,6} = 'Scenario';

[row,col] = find(Roundn_Mean_N_ht_M_OAtrain_subset == ...
    max(max(Roundn_Mean_N_ht_M_OAtrain_subset)));
index = [row,col];   %save row and column as index
Linear_idx = sub2ind(size(Roundn_Mean_N_ht_M_OAtrain_subset),
row,col);
```

```
Roundn_Mean_N_ht_M_OAtest_subset(Linear_idx);
Roundn_Mean_N_ht_M_MSE_subset(Linear_idx);
Roundn_Mean_N_ht_M_Time_subset(Linear_idx);

Optimal_OAtrain{2,2} = Max_Roundn_Mean_N_ht_M_OAtrain_subset;
Optimal_OAtrain{2,3} = Roundn_Mean_N_ht_M_OAtest_subset(Linear_idx);
Optimal_OAtrain{2,4} = Roundn_Mean_N_ht_M_MSE_subset(Linear_idx);
Optimal_OAtrain{2,5} = Roundn_Mean_N_ht_M_Time_subset(Linear_idx);
Optimal_OAtrain{2,6} = Linear_idx;


%Optimal_for_all
Optimal_for_all = cell(4, 9); %create a cell array of 11 times 6
dimension
Optimal_for_all{1,1} = 'Binary classifiers';
Optimal_for_all{2,1} = 'CYT';
Optimal_for_all{3,1} = 'ERL';

Optimal_for_all{1,2} = 'OAtrain';
Optimal_for_all{1,3} = 'OAtest';
Optimal_for_all{1,4} = 'Class1Atrain';
Optimal_for_all{1,5} = 'Class1Atest';
Optimal_for_all{1,6} = 'Class2Atrain';
Optimal_for_all{1,7} = 'Class2Atest';
Optimal_for_all{1,8} = 'MSE';
Optimal_for_all{1,9} = 'Time';

Optimal_for_all{2,2} = Max_Roundn_Mean_N_ht_M_OAtrain_subset;
Optimal_for_all{2,3} = Max_Roundn_Mean_N_ht_M_OAtest_subset;
Optimal_for_all{2,4} = Max_Roundn_Mean_N_ht_M_Atrain1_subset;
Optimal_for_all{2,5} = Max_Roundn_Mean_N_ht_M_Atest1_subset;
Optimal_for_all{2,6} = Max_Roundn_Mean_N_ht_M_Atrain2_subset;
Optimal_for_all{2,7} = Max_Roundn_Mean_N_ht_M_Atest2_subset;
Optimal_for_all{2,8} = Min_Roundn_Mean_N_ht_M_MSE_subset;
Optimal_for_all{2,9} = Min_Roundn_Mean_N_ht_M_Time_subset;

%---------------Training the remaining binary classifiers-----------
%The above process has shown how to train one binary classifier. To
train
```

```
%all the binary classifiers, the above process must be repeated as
many
%times as the number of classifiers to train. Each time, the
appropriate
%target matrix must be fed to the network. The input matrix remains
the
%same, i.e. one for the E.coli, and one for the Yeast classifiers.
```