DOCTORAL THESIS 2016

Large Scale Data Mining via Parallel and Distributed Neural Networks and Machine Learning Schemes

by
Yiannis Kokkinos

A dissertation submitted to the Graduate Faculty of the
Department of Applied Informatics,
University of Macedonia, Thessaloniki, Greece,
in partial fulfillment of the
requirements for the Degree of
Doctor of Philosophy

Thessaloniki, Greece
2016

Supervisor: Professor Konstantinos G. Margaritis

Devoted to my family and friends

ABSTRACT (IN GREEK)

Για μία περίοδο τριών δεκαετιών τα τεχνητά νευρωνικά δίκτυα, η παράλληλη και κατανεμημένη επεξεργασία και η μηχανική μάθηση με την εξόρυξη γνώσης από δεδομένα συνεχίζουν να αναπτύσσονται και να εξελίσσονται. Όλο και περισσότερο συχνότερα στις μέρες μας συνεργάζονται και συνδυάζονται μεταξύ τους λόγω των καθημερινών και αυξημένων αναγκών για ανάλυση μεγάλων όγκων δεδομένων. Στη διατριβή μελετάμε νέες μεθόδους για το θέμα: "Μεγάλης κλίμακας εξόρυξη γνώσης με Παράλληλα και Κατανεμημένα Νευρωνικά Σχήματα Μηχανικής Μάθησης". Η διατριβή εστιάζει σε τέσσερις τομείς: νέους παράλληλους αλγόριθμους για νευρωνικά δίκτυα, νέες μεθόδους κατανεμημένης εκπαίδευσης επιτροπών νευρωνικών δικτύων για εξόρυξη γνώσης από κατανεμημένα δεδομένα, ένα ιεραρχικό νευρωνικό δίκτυο κατάλληλο για ιεραρχίες, και έναν αλγόριθμο τοπικής μάθησης για νευρωνικά δίκτυα.

Παράλληλοι αλγόριθμοι για νευρωνικά δίκτυα Extreme Learning Machines (ELM) διερευνώνται στα κεφ. 3, 4. Πρώτα δείχνουμε ότι για την κλιμακούμενη εκπαίδευσής τους, η δημιουργία πολλών μοντέλων ELM και η επιλογή του καταλληλότερου μοντέλου αντιμετωπίζονται ως μία φάση, την οποία μπορούμε έτσι να επιταχύνουμε μέσω γνωστών μεθόδων παραγοντοποιήσεων πινάκων όπως Cholesky, SVD, QR και Eigen Value decomposition. Τα θεωρητικά και πειραματικά αποτελέσματα έδειξαν ότι ο συνδυασμός του προτεινομένου γρήγορου 10-fold cross validation με Eigen Value decomposition υπερτερεί έναντι όλων των άλλων συνδυασμών σε ταχύτητα. Έπειτα προτείνουμε το Enhanced Convex Incremental Extreme Learning Machine (ECI-ELM) που συνδυάζει τα δύο γνωστά Enhanced I-ELM και Convex I-ELM και έτσι λειτουργεί καλύτερο από αυτά ως προς την ακρίβεια και δείχνουμε ότι το παράλληλο ECI-ELM στην αρχιτεκτονική master-worker προσφέρει επιταχύνσεις πολύ κοντά στην ιδεατή γραμμική επιτάχυνση καθώς αυξάνουμε τον αριθμό των επεξεργαστών.

Παράλληλοι αλγόριθμοι εκπαίδευσης ταξινομητών για Probabilistic Neural Network (PNN) διερευνώνται στο κεφ. 5 στο πλαίσιο μίας αρχιτεκτονικής παράλληλης σωληνωτής επεξεργασίας σε δακτύλιο, όπου απεικονίζεται το σχήμα εκπαίδευσης του προτεινόμενου αλγόριθμου kernel averaged gradient descent Subtractive Clustering με τον αλγόριθμο Expectation Maximization. Αντίθετα από άλλες μεθόδους εδώ δεν απαιτείται καμία παράμετρος από τον χρήστη και η εκπαίδευση του δικτύου γίνεται αυτόματα. Ο δακτύλιος σωληνωτής επεξεργασίας επιτρέπει τον διαμερισμό τόσο των δεδομένων όσο και τον διαμερισμό των νευρώνων του δικτύου στους επεξεργαστές, και κλιμακώνεται τόσο σε πολλά δεδομένα όσο και σε μεγάλα μοντέλα νευρώνων. Ο δακτύλιος επιτρέπει επίσης την χρήση της ακολουθία {Send-Compute-Receive} που επικαλύπτει τις καθυστερήσεις επικοινωνίας με τις καθυστερήσεις υπολογισμών, επιτυγχάνοντας σχεδόν γραμμικές επιταχύνσεις και κλιμάκωση.

Παράλληλοι αλγόριθμοι εκπαίδευσης για Radial Basis Function (RBF) Neural Networks μελετούνται στο κεφ. 6 πάλι μέσω της αρχιτεκτονικής παράλληλης σωληνωτής επεξεργασίας σε δακτύλιο. Ο συνδυασμός του προτεινόμενου αλγόριθμου kernel averaged gradient descent Subtractive Clustering επιλέγει αυτόματα τα κέντρα των RBF νευρώνων και η εκπαίδευση συνεχίζεται με τον αλγόριθμο mini-batch gradient descent που βρίσκει τις παραμέτρους του RBF δικτύου. Πάλι χρησιμοποιείται η ακολουθία {Send-Compute-Receive} για την παράλληλη υλοποίηση των παραπάνω αλγορίθμων. Η προτεινόμενη προσέγγιση κλιμακώνεται σε πολλά δεδομένα και είναι κατάλληλη τόσο για συστήματα μοιραζόμενης μνήμης όσο και για συστάδες επεξεργαστών κατανεμημένης μνήμης.

Η κατανεμημένη μάθηση μελετάται στο κεφ. 7 στο πλαίσιο μιας επιτροπής κατανεμημένων νευρωνικών δικτύων ταξινομητών κατάλληλων για συστήματα P2P. Εδώ για την μηχανή επιτροπής και για όλα τα υπόλοιπα δίκτυα χρησιμοποιούνται τα Regularization Networks. Καθώς τα δεδομένα είναι επιμερισμένα σε πολλές τοποθεσίες, που δεν μπορούν να συγχρονιστούν μεταξύ τους και δεν εμπιστεύονται η μία την άλλη, εγείρεται η ανάγκη ασύγχρονων και κατανεμημένων υπολογισμών που διατηρούν την εμπιστευτικότητα των κατανεμημένων δεδομένων. Προτείνουμε μία απλή στρατηγική που δημιουργεί έναν πίνακα χαρτογράφησης των νευρωνικών δικτύων με βάση την ακρίβεια του κάθε ενός στα δεδομένα του κάθε άλλου. Με αυτό τον πίνακα εκπαιδεύονται τα βάρη της μηχανής επιτροπής. Η προτεινόμενη μέθοδος υπερτερεί έναντι γνωστών μεθόδων majority voting, weighted average και stacked generalization.

Για την κατανεμημένη μάθηση το κεφ. 8 εξετάζει το πρόβλημα της περικοπής του συνόλου νευρωνικών δικτύων ταξινομητών και επιλογής των καλύτερων. Προτείνουμε τον αλγόριθμο Confidence Ratio Affinity Propagation, που εκτελεί πρώτα έναν κύκλο ασύγχρονων και κατανεμημένων υπολογισμών που διατηρούν την εμπιστευτικότητα των κατανεμημένων δεδομένων, και φτιάχνουν ένα πίνακα αμοιβαίων επικυρώσεων με όλα τα ζεύγη των δικτύων. Το κάθε ζεύγος παράγει ένα μέτρο για την μεταξύ τους απόσταση βασισμένο στο κλάσμα εμπιστοσύνης (Confidence Ratio). Έπειτα ο γνωστός αλγόριθμος Affinity Propagation επιλέγει τα καλύτερα δίκτυα. Συγκρίσεις με άλλα γνωστά μέτρα και άλλες μεθόδους περικοπής του συνόλου, όπως reduce-error pruning, Kappa pruning, orientation pruning, JAM's diversity pruning δείχνουν ότι η προτεινόμενη μέθοδος μπορεί να επιλέξει τα καλύτερα δίκτυα, με λιγότερους υπολογισμούς και δίχως ο αριθμός τους να δίνεται ως παράμετρος.

Για την κατανεμημένη μάθηση προτείνεται στο κεφ. 9 ένα probabilistic neural network (PNN) Committee Machine όπου στο στρώμα προτύπων του PNN το κάθε νευρωνικό δίκτυο ειδικεύεται σε ξεχωριστή κλάση κατανεμημένων δεδομένων. Έπειτα για κάθε τοποθεσία που διατηρεί έχει το δικό της νευρωνικό δίκτυο εκτελείται ένας κύκλος ασύγχρονων και κατανεμημένων υπολογισμών που διατηρούν ωστόσο την εμπιστευτικότητα των κατανεμημένων δεδομένων, για να κατασκευαστεί έναν πίνακας αμοιβαίων επικυρώσεων, όπου το κάθε δίκτυο επικυρώνει τα υπόλοιπα. Από αυτό τον πίνακα δείχνουμε ότι είναι δυνατό να εφαρμόσουμε επιλογή των καλυτέρων ταξινομητών μέσω εύρεσης των μη-αρνητικών βαρών τους.

Η ιεραρχική μάθηση αντιμετωπίζεται με ένα νέο ιεραρχικό Μαρκοβιανό Radial Basis Function neural network (HiMarkovRBFNN) στο κεφ. 10. Το προτεινόμενο μοντέλο βασίζεται στην κλασική στρατηγική "διαίρει και βασίλευε" που επιτρέπει την αναδρομική λειτουργία του δικτύου μέσω πρόσβασης των δεδομένων από κάτω προς τα πάνω. Σε κάθε επίπεδο οι κρυφοί νευρώνες του ιεραρχικού νευρωνικού δικτυού αποτελούνται από άλλα πλήρως εμφωλευμένα νευρωνικά δίκτυα. Έτσι η λειτουργία του είναι μία συνάρτηση αναδρομής ίδια σε όλα τα επίπεδα.

Για τους αλγόριθμους τοπικής μάθησης στο κεφ. 11, που δημιουργούν για διαφορετικά σημεία και διαφορετικά τοπικά μοντέλα νευρωνικού δικτύου, βασιζόμενοι στα κοντινότερα δεδομένα εκπαίδευσης, εξετάζουμε τα Regularization Networks. Μελετούμε τέσσερεις περιπτώσεις για την βελτίωση τόσο της ταχύτητάς τους όσο και της ακρίβειας πρόβλεψης. Δείχνεται ότι για την ελάττωση των χρόνων υπολογισμού και βελτιστοποίησης των παραμέτρων τους, η καθολική βελτιστοποίηση, με ένα σύνολο παραμέτρων κοινό για όλα τα δίκτυα, παράγει καλύτερα αποτελέσματα ταχύτητας και ακρίβειας από την βελτιστοποίηση, με ένα σύνολο παραμέτρων ξεχωριστό για κάθε δίκτυο.

ABSTRACT

For a period of more than three decades the fields of artificial neural networks (ANNs), parallel and distributed computing as well as data mining and machine learning are continuously evolving. More and more often in our days, driven by the ever increasing demands of analyzing large scale data, they cooperate with each other and combined. In this thesis we study Parallel and Distributed Neural Networks and Machine Learning Schemes, which during the last years became the golden standard in many fields of real life applications. The thesis focuses on four important factors: new parallelizable algorithms for neural networks, new distributed privacy-preserving neural network ensembles and committees for data mining, a new hierarchical markovian RBF Neural Network algorithm and a local learning algorithm for Regularization Networks.

Parallelizable algorithms for regression with Extreme Learning Machines (ELM) are investigated in chapters 3, 4. First we show that for scalable ELM training the model construction and the model selection phase should not be treated separately but as one phase, in which we can speedup the computations by employing Cholesky, SVD, QR and Eigen decompositions. The theoretical and experimental results reveal that the combination of the proposed fast 10-fold cross validation with Eigen Value decomposition outperforms all the other combinations in terms of scalability and speed. In chapter 4 we propose the Parallel Enhanced and Convex Incremental Extreme Learning Machine (ECI-ELM) which combines two popular existing incremental versions of ELM, namely the Enhanced I-ELM with the Convex I-ELM, and thus it outperforms them in accuracy. Chapter 4 also demonstrates that the master-worker parallel ECI-ELM brings speedups very close to the ideal case.

Parallelizable algorithms for training parsimonious Probabilistic Neural Network (PNN) classifiers are investigated in chapter 5, in the context of a Ring pipeline parallel scheme, where we map the proposed leave-one-out kernel averaged gradient descent algorithm together with Subtractive Clustering and Expectation Maximization algorithm that refines all PNN network parameters. Both neurons and data are held distributed across the processors in the Ring pipeline were we specifically use the {Send-Compute-Receive} pattern that permits overlapping the computation delays with the communication delays. Unlike other methods, in the proposed scheme no user-defined parameters are required and the PNN model is created automatically. In addition the PNN training has much lower computational complexity and memory requirements that the other methods in the comparisons.

Parallel algorithms for Radial Basis Function (RBF) Neural Networks for regression tasks are studied in chapter 6 by using the Ring pipeline architecture. We use the recently proposed leave-one-out Kernel Averaged Gradient descent Subtractive Clustering (KG-SC) for automatically selecting appropriate RBF centers and then the training continues with a mini-batch gradient descent for refining all the network parameters {centers, widths, output weights}. In contrast to master/worker or pipeline where either data or neurons are partitioned, in the ring pipeline both neurons and data are distributed across the processors. The {Send-Compute-Receive} pattern that overlaps computation delays with communication delays is also employed. The proposed approach is suitable for distributed clusters of workstations as well as shared memory machines and scales well on increasing the number of processors in the ring.

Distributed learning in chapter 7 is studied in terms of a Regularization Network committee machine composed of many distributed Regularization networks suitable for Peer-to-Peer systems. While the data are distributed across the locations we consider the challenging case of asynchronous training and with no local data exchange among the classifiers and propose a simple strategy that maps all peers in an asymmetric mutual validation matrix. The training set of one peer becomes the test set of the other and vice versa. The method favors the distributed asynchronous nature in P2P systems. Experimental results show that the proposed RN committee outperforms majority voting, weighted average and stacked generalization in most of the cases.

For distributed learning in chapter 8 another classical problem is ensemble pruning. An ensemble of distributed neural network models can split the dataset into several data partitions of limited size and learns a model separately for each partition. Then the pruning phase must select the best of the ensemble members. A fully distributed and privacy-preserving computation of two mutual validation matrices is the basis for the proposed ensemble selection approach that can finally constructs a two-by-two mapping of all the classifier pairs by using simple one directional point-to-point communication messages. We finally propose Confidence Ratio Affinity Propagation that selects the best neural networks which form the ensemble. The proposed model showed promising results as compared with other pruning methods, such as reduce-error pruning, Kappa pruning, orientation pruning and JAM's diversity pruning.

Distributed learning in chapter 9 deals with class specialized regularization network Peer classifiers in the framework of a probabilistic neural network (PNN) committee machine. The PNN pattern neurons are composed of locally trained class-specialized regularization networks. Then an asynchronous distributed computing P2P cycle is executed to construct a mutual validation matrix. From this matrix, based on regularized non-negative weighting, it is possible to perform weight based ensemble selection of best members for every class. The scheme manages to use fewer Peers thus increasing classification speed and improves significantly the PNN committee accuracy.

Hierarchical learning in chapter 10 is dealt with a new Hierarchical Markovian Radial Basis Function neural network (HiMarkovRBFNN) topology. The proposed model depends in the classical divide-and-conquer parallel programming strategy and allows data access in a recursive fashion that enables recursive operations. The hierarchical structure of this network is composed of nested RBF Neural Networks with arbitrary levels of hierarchy. All hidden neurons in the hierarchy levels are composed of truly RBF Neural Networks with two weight matrices. The neural network operation is exactly the same at all levels in contrast to the simple summation neurons with only linear weighted combinations which are usually encountered in ensemble models and cascading networks. Comparisons with Committee Machines and Cascaded Machines reveal that the proposed HiMarkovRBFNN serves well as a hierarchical combiner.

For Local Learning algorithms that use a different model for a different testing point, we consider Regularization Networks in chapter 11 for creating each model. We explore four different cases for improving the training accuracy and speed by exploiting the interplay between locally optimized and globally optimized parameters, in order to reduce the optimization time of the hyper-parameters needed at runtime. Global parameters perform better. We also examine different matrix decompositions and found that Cholesky decomposition should be preferred when validating several neighbourhood sizes as well as when the local network operates online.

PREFACE

In essence we are studying the rapidly growing theme of Parallel and Distributed Neural Networks and Machine Learning Schemes which continues to make evolutionary jumps year-by-year. The thesis is based on a number of papers, each presented in a different chapter. Scalable cross-validation algorithms for model selection are presented in chapter 3. ***Parallelized*** algorithms for Neural Networks are described in chapters 4, 5 and 6 for Extreme Learning Machines (ELM), Probabilistic Neural Networks (PNN) and RBF Neural Networks respectively. ***Distributed*** Neural Network Ensembles for privacy-preserving data mining are given in chapters 7, 8 and 9. ***Hierarchical*** markovian RBF Neural Network algorithms are discussed in chapter 10. ***Local*** Learning algorithms for Regularization Networks are presented in chapter 11.

JOURNAL PUBLICATIONS

	Reference	Related Chapter
1	Kokkinos Y. and Margaritis K.G. (2014) "A distributed privacy-preserving regularization network committee machine of isolated Peer classifiers for P2P data mining". Artificial Intelligence Review, 42(3), pp. 385–402.	7
2	Kokkinos Y. and Margaritis K.G. (2015). "Confidence ratio Affinity Propagation in ensemble selection of neural network classifiers for distributed privacy-preserving data mining". Neurocomputing, 150, pp. 513–528.	8
3	Kokkinos Y. and Margaritis K.G. (2015) "Topology and simulations of a Hierarchical Markovian Radial Basis Function Neural Network classifier". Information Sciences, vol. 294, pp. 612–624.	10
4	Kokkinos Y. and Margaritis K.G. (2016) "Local Learning Regularization Networks for localized regression". Neural Computing and Applications. In press. DOI :10.1007/s00521-016-2569-0	11
5	Kokkinos Y. and Margaritis K.G. (2016) "Big data regression with Parallel Enhanced and Convex Incremental Extreme Learning Machines", accepted	4
6	Kokkinos Y. and Margaritis K.G. (2016) "Managing the computational cost of model selection and cross-validation in Extreme Learning Machines via Cholesky, SVD, QR and Eigen decompositions", under review	3
7	Kokkinos Y. and Margaritis K.G. (2016) "Ring Pipeline Parallel algorithms for scalable parsimonious Probabilistic Neural Networks", under review	5
8	Kokkinos Y. and Margaritis K.G. (2016) "Parallel Radial Basis Function Neural Networks for scalable regression with mini-batch gradient learning in a ring pipeline architecture", submitted	6
9	Kokkinos Y. and Margaritis K.G. (2016) "P2P data mining via Asynchronous Distributed and privacy-preserving weight-based ensemble pruning of class-specialized Neural Network Peers", submitted	9

CONFERENCE PUBLICATIONS

	Reference	Related Chapter
1	Kokkinos Y. and Margaritis K.G. (2012). "Parallelism, localization and chain gradient tuning combinations for fast scalable Probabilistic Neural Networks in data mining applications". In 7th Hellenic Conference on Artificial Intelligence (SETN 2012), Springer Lecture Notes in Artificial Intelligence 7297, pp. 41–48.	5
2	Kokkinos Y. and Margaritis K.G. (2012). "A distributed asynchronous and privacy preserving neural network ensemble selection approach for peer-to-peer data mining". In 5th Balkan Conference in Informatics, BCI 2012.	8
3	Kokkinos Y. and Margaritis K.G. (2012). "A Parallel Radial Basis Probabilistic Neural Network for scalable data mining in distributed memory machines". In 24th IEEE International Conference on Tools with Artificial Intelligence, (ICTAI 2012), pp. 1094–1099.	5
4	Kokkinos Y. and Margaritis K.G. (2012). "A Regularization Network committee machine of isolated Regularization Networks for distributed privacy preserving data mining". In 8th International Conference on Artificial Intelligence Applications and Innovations, (AIAI 2012), Springer/ Lecture Notes in Computer Science, IFIP AICT 381, pp. 97–106.	7
5	Kokkinos Y. and Margaritis K.G. (2013). "A Parallel and Hierarchical Markovian RBF Neural Network: preliminary performance evaluation". In 14th Conference on Engineering Applications of Neural Networks (EANN 2013), Springer Lecture Notes in Computer Science, CCIS 383, pp. 340–349.	10
6	Kokkinos Y. and Margaritis K.G. (2013). "Parallel and local learning for fast Probabilistic Neural Networks in scalable data mining". In ACM Proceedings of 6th Balkan Conference in Informatics, (BCI 2013), pp. 47-52.	-
7	Kokkinos Y. and Margaritis K.G. (2013). "Distributed privacy-preserving P2P data mining via Probabilistic Neural Network Committee Machines". In IEEE Proceedings of 4th International Conference on Information, Intelligence, Systems, and Applications (IISA 2013), pp. 151-154.	9
8	Kokkinos Y. and Margaritis K.G. (2014). "Breaking ties of plurality voting in ensembles of distributed neural network classifiers using soft max accumulations". In 10th International Conference on Artificial Intelligence Applications and Innovations, (AIAI 2014), Springer Lecture Notes in Computer Science, IFIP AICT 436, pp. 20–28.	8
9	Kokkinos Y. and Margaritis K.G. (2015). "Multithreaded Local Learning Regularization Neural Networks for regression tasks". In 16th International Conference on Engineering Applications of Neural Networks, (EANN 2015), Springer Communications in Computer and Information Science, CCIS 517, pp. 129–138	11
10	Kokkinos Y. and Margaritis K.G. (2015). "A fast progressive Local Learning regression ensemble of Generalized Regression Neural Networks". In 19th Panhellenic Conference on Informatics (PCI 2015).	-
11	Kokkinos Y. and Margaritis K.G. (2016). "Exemplar selection via leave-one-out kernel averaged gradient descent and Subtractive Clustering", In 12th International Conference on Artificial Intelligence Applications and Innovations (AIAI 2016), Springer Lecture Notes in Computer Science, IFIP AICT 475.	5, 6

ACKNOWLEDGEMENTS

There are three people whose teaching had a stronge influence on the subject of this Ph.D Thesis. First and most I wish to express my gratitude to my supervisor Prof. Konstantinos G. Margaritis for his support, assistance and motivation. He was an excellent teacher for introducing me into many aspects of parallel and distributed systems, parallel programming languages and mapping neural networks in parallel architectures. His comments and advises during the evolution of this thesis were valuable. Next I wish to thank the first of my doctoral committee member Prof. Ioannis Refanidis. He did a great work on teaching us artificial intelligence and neural networks during the courses in the preceded 2 years Master in Computer Systems program, where I was ranked first in my class. In addition, I wish to thank Prof. Nikolaos Samaras, the second of my doctoral committee member, from whom I learn a lot on numerical linear algebra, algorithmic design and optimization. I am grateful to all three of them for their time and guidance. Practically they all contribute in forming the subject of this Thesis.

I would also like to express my gratitude to Prof. Lazaros Iliadis and Prof. Yannis Manolopoulos for their support in AIAI and EANN conferences.

Likewise I wish to thank the people in the Parallel and Distributed Processing Laboratory as well as those working in the computer center for supporting the computer clusters and making all the required computer language installations.

Keywords/Descriptors: artificial neural networks, large scale, data mining, classification, regression, clustering, parallel learning, distributed learning, neural network ensembles, hierarchical markovian neural networks, local learning systems.

TABLE OF CONTENTS

1 Introduction and Thesis structure

Reasoning, problem solving, inference and decision making are some of the admirable abilities attributed to **intelligence**, in which the most vital and prominent characteristic is **learning**. During the learning process an individual or a system changes inside and continuously adapts to its environmental stimuli coming from outside. The human endeavors of using mathematics and computer machines to mimic learning are described in Machine Learning.

1.1 Machine Learning

Machine Learning [1][2][3][4], a branch of Artificial Intelligence, is a scientific discipline concerned with the design and development of algorithms that allow computers to evolve behaviours based on empirical data, such as sensor data or databases. A major focus of machine learning research is to automatically learn to recognize complex patterns and make intelligent decisions based on data. Data can be seen as examples that illustrate relations between observed variables. Statistical learning, kernel methods, graphical models, artificial neural networks, fuzzy logic, linear models, instance-based schemes, clustering algorithms, Bayesian methods are few sub-disciplines of machine learning. During the past two decades, with the exponentially increasing amounts of available data and the growing needs for data analysis, machine learning has become a part of our daily life (to see an example turn on your computer and perform an internet search). This data growth has shifted the research stream from highly sophisticated algorithms that can operate only on small datasets to less complex but more efficient algorithms suitable for much larger datasets (the latter characteristic is sometimes attributed to data mining). Machine Learning is a continually growing research field in both academia and industry as its algorithms are widely used in real-world problems for pattern recognition, image processing, computer vision, knowledge discovery and data mining.

1.2 Data mining

Data mining [5] [6] [7] [8] refers to discovery of implied and previously unknown but potentially useful information like knowledge rules, repeated patterns, outliers, anomalies and statistically important structures from large data banks. Many problems in science and industry have been addressed by data mining. Among the main data mining tasks we are going to see in this thesis are classification, regression and clustering, which are also very common tasks in the field of machine learning.

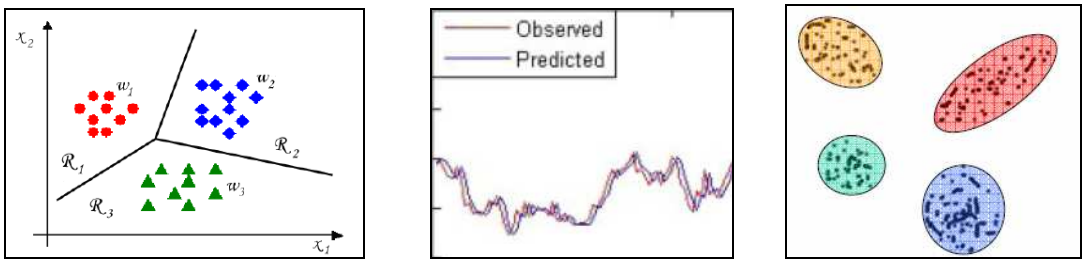


Figure 1.1 (a) classification (b) regression (c) Clustering

Classification is the assignment of objects into predefined classes. Regression task learns a non-linear function $y = f(\mathbf{x})$, which is a mapping between observed known inputs $\mathbf{x}$ and their real-valued outputs y, in order to predict the output of unknown inputs with the least possible error. Clustering is the task of discovering structures by grouping the data objects into collections that are similar locally.

Classification or regression can also cluster data into small groups as a pre-processing step before training. Several methodologies of data mining are derived from machine learning and artificial intelligence. Such methods include decision trees, k-nearest-neighbours, fuzzy logic, Bayesian methods, statistical methods and artificial neural networks.

1.3 Artificial Neural Networks

Artificial Neural Networks (ANNs) [9] [10] are learning algorithms that are inspired by the structure and/or functional aspects of biological neural networks. Computations are structured in terms of interconnected groups of artificial neurons. Artificial neural networks are in the core of Machine Learning and Artificial Intelligence. However, they are less frequently in the core of data mining, although it is known for over 20 years (see [11]) that, owning to their robustness, self-organization, fault tolerance, parallel processing and adaptability, artificial neural networks are quite appropriate for solving data mining problems. ANNs are usually applied to model complex relationships between inputs and outputs, to find patterns in data, or to capture the statistical structure in an unknown joint probability distribution between observed variables.

Neural Networks are considered to be time consuming. Their general abilities when compared with other data mining techniques to deal with data features show that they are very good in handling noisy data, not scalable in processing large data sets, have excellent predictive accuracy and good integration abilities. However as a rule of thumb, ANNs are more accurate than many data mining techniques and the choice decision of the appropriate data mining tool is usually a cost-benefit analysis when it comes to real life applications. One percent increase in the accuracy may have important improvements for many domains in the data mining process.

Over the last years they attract considerable interest as data mining tools for rule extraction, classification and feature selection tasks [7]. This thesis describes some new algorithms for training and parallelisation of:

- Extreme Learning Machines (ELM)
- Probabilistic Neural Networks (PNN)
- Radial Basis Function Neural Networks (RBFNN)
- Regularization Networks (RN)
- Neural Network Ensembles and Committee Machines

All these algorithms are considered also for large scale data mining tasks.

1.4 Large scale data mining

Large scale data mining [12] [13] [14] may have many definitions:

- the size of datasets can be very large and simple cannot fit in memory
- the algorithm cannot be applied on a single machine in reasonable time (e.g., for linear classifiers 100K is medium size but for kernel classifiers, it is large)
- the number of classes may be large (i.e. in image categorization)
- the weights (the parameters) of desired models may be large,
- in a parallel environment the number of distributed processors is large.

The characteristic of large scale data mining algorithms is the need for speed and efficiency. While an algorithm that has quadratic computational complexity O(N^2) can take 1 second for processing 10,000 data records, it needs days for 10,000,000. Thus, a computational complexity larger than quadratic is usually not suitable for large scale tasks. However, the majority of artificial intelligence and machine learning algorithms fall into this category.

Table 1.1. Different types of scale

Category	N examples	Desired complexity
Small scale	$10^0 – 10^2$	$O(N^4) – O(N^3)$
Medium scale	$10^3 – 10^4$	$O(N^3) – O(N^2)$
Large scale	$10^5 – 10^6$	$O(N^2) – O(N^{3/2})$
Big scale	$10^7 – 10^{12}$	$O(N^{3/2}) – O(N\ logN)$

The how-to of making machine learning and data mining methods like neural networks to operate and scale well on large scale data banks is based on improving their time complexity, using algorithmic approximations, and employing parallel and distributed systems with many processor nodes. These directives are recognized nowadays as ongoing research streams.

The subject of the current thesis is anchored in these directives with which we attempt to deal with large scaling in the Neural Network training. For such scalable machine learning the recommended parallel computing frameworks (see also [12]) are MapReduce, Project B and Message Passing Interface (MPI).

1.5 Message Passing

Message Passing concerns communications between processors in a parallel system and is the essence of this thesis that is based on several Message Passing paradigms for the parallel, distributed and even hierarchical Neural Network training where messages are exchanged between processors. The general view on this subject is shown graphically in fig. 1.2.

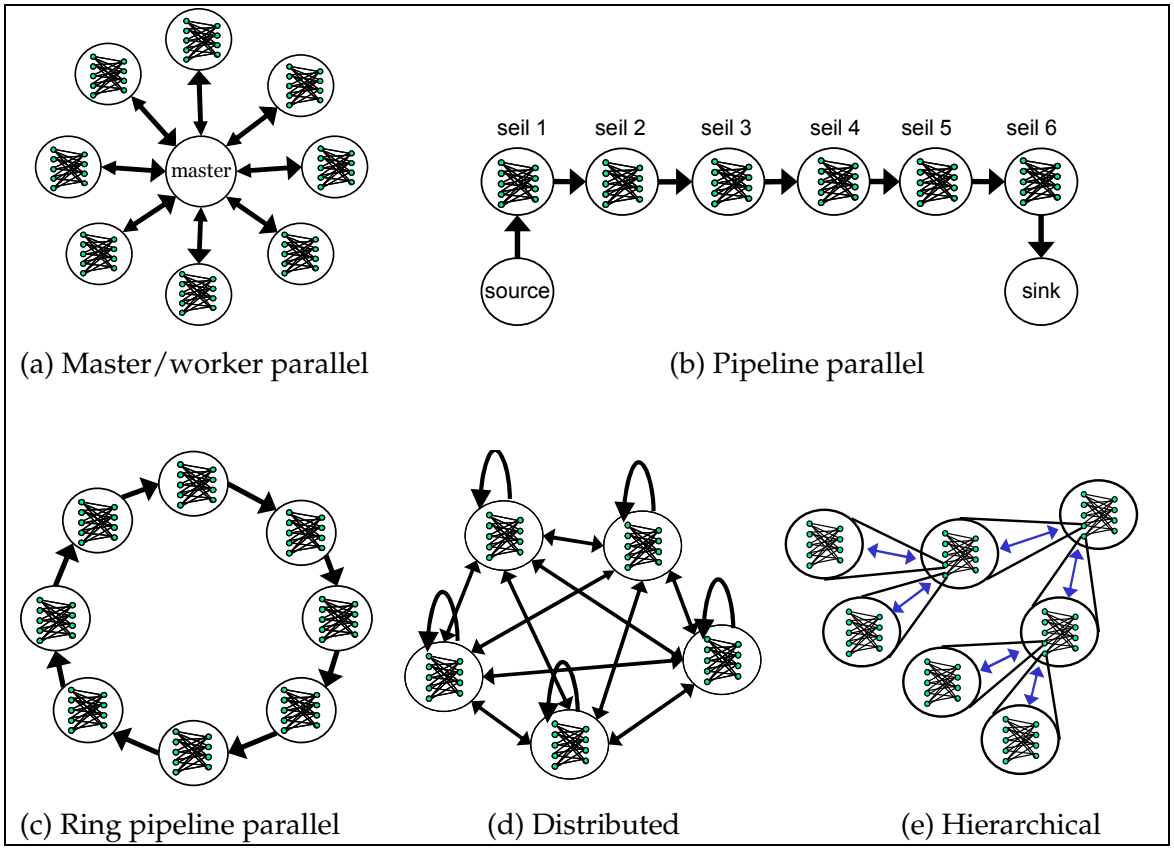


Figure 1.2. Message Passing paradigms on the distributed, parallel and hierarchical Neural Network training, by partitioning neurons and data across: (a), (b) and (c) are Parallel topologies (open systems) with master/worker, or pipeline or ring pipelining respectively, (d) Distributed isolated data locations (close systems), (e) Hierarchical structures (nested systems). Arrows represent the messages passing from and to processor nodes that hold neurons or data or both.

Assume that several points in a plane must be connected. Then anyone can draw a star, a line, a ring, or connect all points together in a clique, or arrange them hierarchically and create a tree. So the sketches in fig. 1, that actually represent possible ways of connecting points, are rather familiar and our only concern is how to use them as neural network learning paradigms as well. While fig. 1.2 illustrates how **Message passing** is the basis in the parallel, distributed or hierarchical paradigms we adopt and implement for neural network learning, there are distinct differences in exchanging those messages.

In fig 1.2(a) the well known *master/worker parallel architecture* is an open system that allows centralized data access for all processors in order to create one model. The master processor holds the training data and the workers hold portions of neurons. There is one program executed by the master coordinator and another different

program executed by the workers. Messages are broadcasted by the master and received by the workers which do the work and send their local results back to the master in a synchronous manner. The communication must use **synchronous message passing**, illustrated by the arrows, with synchronization locks in the master, which merges the results.

In fig 1.2(b) the processors are aligned in a *pipeline architecture*. When neurons are distributed the source (the first processor) must hold the data, each seil hold a different neuron portion, and the sink (the last processor) finalize the results. There is one program executed by the source which sends data batches only to the next processor, another program executed by the seils and another program executed by the sink. Each processor just pass a message to its next and receive a message from its previous, thus using a simple **asynchronous send-receive message passing** in every step.

In fig. 1.2(c) we especially adopt the *Ring pipeline parallel architecture*. In the *ring pipelining* every processor has a portion of the neurons and a portion of the training data. There is a single program all processors execute with no synchronization points. Like pipeline every processor passes a message to its next and receive a message from its previous, thus using again **asynchronous send-receive message passing**.

In fig. 1.2(d) distributed learning from isolated data locations (close systems where each processor access its own subset of data) are independent parallelism strategies that use **asynchronous message passing** with possible queues. There is an all-to-all message exchanging with no synchronization points. Each processor has an entire neural network model and a portion of the data and they all create an ensemble of models.

In fig. 1.2(e) Hierarchical structures (nested systems that allow data access in a recursive fashion) are classical divide-and-conquer parallel programming strategies that use **tree-based message passing** (recursive) to create one hierarchical model. We apply all these paradigms in the neural network training.

1.6 Neural Network training

Neural Network training mainly depends on the volume of the training dataset. The network must scale well on large datasets. Like human training which is slow when we have to learn to respond in many stimuli, the ANN scalability is quite low. Improving the speed and efficiency in the construction and training of several ANN types, is of highest importance and it is usually done via approximations or mappings into parallel and distributed environments.

Mapping Neural Networks in parallel architectures and schemes have been studied for a long time [15] [15] [16] [17]. A lot of effort has also been put into scalable parallel implementations of Multi-Layer Perceptrons with the Back-propagation algorithm [18] [19]. In this thesis we show that when a dataset is too large for a particular neural network learning algorithm to work efficiently the learning speed can be improved by scalable model selection algorithms, parallel learning (master-worker or pipelined), data clustering, distributed learning (via ensemble methods) and hierarchical learning.

1.7 Structure of the thesis

The next figure illustrates the big picture of the current thesis. The parallel as well as the hierarchical approaches are dealing with one neural network, while the distributed approaches create neural network ensembles.

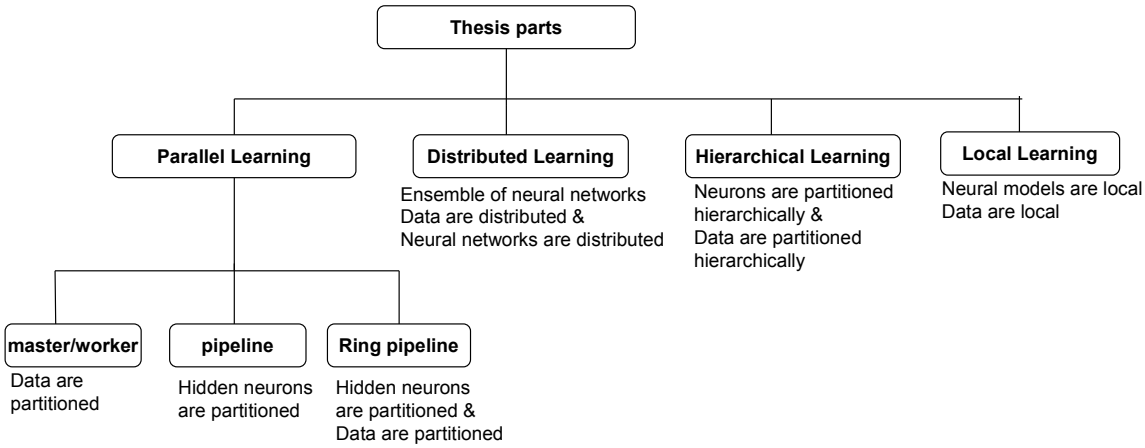


Figure 1.3. The big picture and the basic differences between the approaches we use.

There is a large literature of related works that are analysed in each chapter separately.

Scalable model selection algorithms like the classical model selection create several learning models and validate them via cross-validation in order to finally select the best model among them and estimate the expected true performance of the learning scheme. Cross-validation prevents over-fitting when the learned model is too closely tied to the particular training data from which it was built. However, classical model selection that constructs all models independently (and in parallel) is extremely slow. The model construction and the model selection phase should be treated not separately, but instead as though they were an integral part of the training. To this end scalable model selection economizes the computations as much as possible by using one construction-selection phase simultaneously for all models. We investigate such scalable algorithms for ELM model selection via cross-validation in chapter 3.

Parallel learning with Master-worker architecture splits the problem (data) into smaller portions. Master-worker requires centralized synchronous computations. At each iteration step the master processor communicates the work load portions into the worker processors where each one performs computations for its own portion. After finishing the workers send their results back to the master, which uses synchronization points at each step in order to synchronize the workers. While Master-worker parallel systems are suitable for mapping the majority of algorithms, they are not very sufficiently scalable since they suffer from delays and bottlenecks, especially when the synchronization points are many (the number of learning steps) or the communication load is high in comparison with the computation load. This is not the case when few learning steps are needed and small amounts of data must be communicated at each learning step, like the parallel ECI-ELM described in chapter 4.

Parallel learning with Pipelined architecture is another way of reducing the time complexity of learning by split the problem (data and tasks) into smaller parts and distributes them in a pipeline fashion. Pipelining is a most efficient architecture for parallel computations since it does not suffer from frequent communication delays and bottleneck effects, commonly encountered in typical master/worker parallel mappings.

However few algorithms can be entirely operate in a pipeline. Neural network training algorithms that can be performed exclusively in a ring pipelining architecture, which distributes both data and tasks across the processors, are described in chapters 5 and 6.

Data clustering can be used as a pre-processing step for the neural network learning algorithms in order to reduce their model parameters and creating parsimonious models. We describe a novel leave-one-out mini-batch kernel gradient descent with subtractive clustering in chapters 5 and 6.

Distributed ensemble learning is another way to learn from a large dataset by using an ensemble of neural network models. This method splits the dataset into several data partitions (chunks) of limited size and learns a model separately for each distributed partition. Then all these models form a neural network ensemble and combine their results by using voting or averaging or a committee machine. We present ensemble methods for distributed data mining that also preserves privacy in chapters 7, 8, and 9.

Hierarchical learning can speed up the training time of the problem by using proper hierarchical problem domain decompositions, using a divide-and-conquer strategy for both data and tasks, based on specific Hierarchical neural network topologies and algorithms like those we are going to see in chapter 10.

Local learning algorithms build on the fly a different model for a different testing point. By using only a local list of neighbour training points closer to each testing point they learn the local parameters of the local model at runtime after a particular testing point is known and apply the model to predict only this testing point. We explore four different cases for improving the training of local versions of Regularization Networks (RN) optimized for accuracy and speed in chapter 11.

1.8 Analogies of few concepts with real life paradigms

In real life the ‘production chain’ is an industrial successful paradigm that uses pipelined parallel tasks (in our case neuron operations) in order to avoid delays and bottlenecks. The ‘supply chain’ in logistics is an efficient paradigm for transferring provisions (in our case data). Like in real life, the ring pipeline neuron-data parallel model tries to combine concepts from the ‘production chain’ and the ‘supply chain’.

Distributed approaches correspond to ensembles. In real life, we use the word 'ensemble' to denote a group of collaborating individuals that concerted in harmony for the same goal or task. Representative paradigms are the musical ensembles which are met in orchestras and concerts. Like in musical ensembles, the keys for any successful ensemble are coordination, noise removal, diversity, and performance.

In real life various biological organisms, engineering machines and human constructs are arranged hierarchically into well organized smaller modules which have a lower cost of sub-module connections. Like in engineering, a module can be composed of many other sub-modules, and each of them could have inside other sub-modules in a nested fashion. The proposed Hierarchical Markovian RBFNN, which consists of nested nodes that have the same operation in every level, is such a hierarchical modular nested paradigm.

In real life the locality principle prevails. Our decisions and actions are usually driven by our nearest neighbors, and the surrounding environment. Location is a key that characterizes perceptions, events, patterns, beliefs and more. Like in real life the Local learning neural network models, which are created on-the-fly when a new unknown

example arrives, use the k-nearest neighbor training data in order to train a local model suitable only for predicting this unknown example.

1.9 What we have studied so far

The initial planning was to investigate Parallel Learning, Distributed Learning, Hierarchical Learning and Local Learning in Radial Basis Function Neural Networks (RBFNN), Probabilistic Neural Networks (PNN), Regularization Networks (RN) and Extreme Learning Machines (ELM) for large scaling (improving efficiency and scalability). Fig. 1.4 illustrates what we have studied so far.

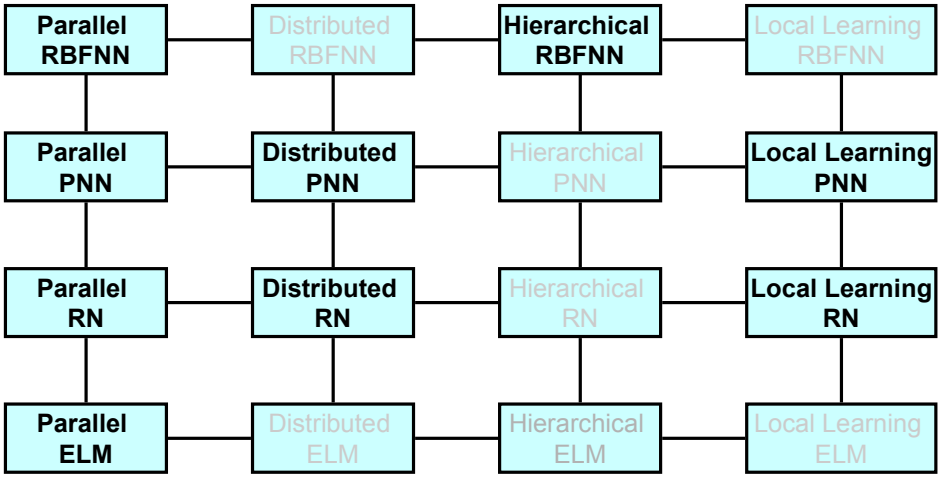


Figure 1.4. What we have studied so far are appeared in bold.

1.10 Organization of Chapters

Chapter 2 presents related concepts and some background material for parallel computing technology, parallel data mining, parallel neural network learning schemes, distributed data mining and ensemble learning.

Chapter 3 investigates the computational cost of model selection and cross-validation in Extreme Learning Machines via Cholesky, SVD, QR and Eigen decompositions. This chapter, in essence, describes the reason for preferring purely parallel algorithmic designs instead of just running a sequential program in many processors 'in parallel'.

Chapter 4 describes big data regression with Parallel Enhanced and Convex Incremental Extreme Learning Machines, by first combining the popular Enhanced I-ELM with the Convex I-ELM and then implementing the new ECI-ELM algorithm in a master-worker parallel architecture.

Chapter 5 demonstrates that a parsimonious Probabilistic Neural Network (PNN) classifier can be efficiently designed for fast and scalable training by using the combined force of the proposed leave-one-out gradient descent algorithm in conjunction with Subtractive Clustering algorithm and Expectation Maximization algorithm in a ring pipeline parallel scheme suitable for shared-nothing parallel systems.

Chapter 6 gives a new algorithm for parallel training of Radial Basis Function Neural Networks (RBFNN) for regression tasks. Based on a leave-one-out mini-batch kernel averaged gradient descent that in a supervised manner extracts from the training data a single piece of information, namely a bandwidth, which can then be used in the well

known subtractive clustering algorithm for automatically finding the number of RBF neurons and their centers. After that the set of centers is given as input to a mini-batch gradient descent training which simultaneously optimizes the centers, widths and weights of the RBF units. We further show how the whole process can be efficiently performed in a ring pipelined parallel architecture that renders the RBFNN training more scalable. The method does not require any used-defined parameters.

Chapter 7 describes a completely asynchronous, scalable and Distributed Privacy-Preserving Regularization Network (RN) committee machine of isolated Peer classifiers for Peer-to-peer distributed data mining. Regularization neural networks are used for all the Peer classifiers and the combiner committee in an embedded architecture. At the end of the training phase no Peer will know anything else besides its own local data. This privacy-preserving obligation is a challenging problem for trainable combiners but is crucial in real world applications.

Chapter 8 presents distributed Confidence ratio Affinity Propagation in ensemble selection of neural network classifiers for distributed data mining. We consider classification tasks for large decentralized data locations which can build several neural networks to form an ensemble. The best neural network classifiers are selected via the proposed confidence ratio affinity propagation in an asynchronous distributed and privacy-preserving computing cycle.

Chapter 9 presents a probabilistic neural network (PNN) committee machine for Peer-to-Peer data mining. The pattern neurons of the PNN committee are composed of locally trained class-specialized regularization network Peer classifiers. The training takes into account the asynchronous distributed and privacy-preserving requirements that can be met in P2P systems. We demonstrate that it is possible to perform weight based ensemble selection of best peer members for every class and in this way to find class-specialized Peer modules for the committee machine.

Chapter 10 gives the topology and simulations of a Hierarchical Markovian Radial Basis Function Neural Network (HiMarkovRBFNN) classifier model that enables recursive operations. All hidden neurons in the hierarchy levels are composed of truly RBF Neural Networks with two weight matrices, for the RBF centers and the linear output weights, in contrast to the simple summation neurons with only linear weighted combinations which are usually encountered in ensemble models and cascading networks. Thus the neural network operation in every node is exactly the same at all levels of the hierarchical integration.

Chapter 11 considers local learning versions of Regularization Networks (RN) and investigates several options for improving their online prediction performance, both in accuracy and speed. In this chapter we exploit the interplay between locally optimized and globally optimized parameters, in order to reduce the optimization time of the hyper-parameters that are needed at runtime (online). We also examine matrix decompositions. While Eigen decomposition is suitable for validating cost–effectively several regularization parameters, Cholesky decomposition should be preferred when validating several neighbourhood sizes (the number of k-nearest neighbours) as well as when the local network operates online. Parallelism in a multi-core system for these local computations demonstrates that their execution times can be further reduced.

References for chapter 1

[1] Mitchell T. (1997) Machine Learning. Burr Ridge, IL: Mcgraw Hill.

[2] Hastie T., Tibshirani R. and Friedman J., (2008) The Elements of Statistical Learning: Data Mining, Inference, and Prediction. Second Edition, Springer.

[3] Smola A. and Vishwanathan S.V.N., (2010) Introduction to Machine Learning. Cambridge University Press.

[4] Welling M. (2010) A First Encounter with Machine Learning. Donald Bren School of Information and Computer Science, University of California Irvine,

[5] Dunham M.H. (2004) Data Mining introductory and advanced topics. Prentice Hall.

[6] Witten I. H. and Frank E., (2005) Data Mining: Practical machine learning tools and techniques. Morgan Kaufmann, 2nd ed.

[7] Wang L., and Fu X. (2005) Data Mining with Computational Intelligence, Springer-Verlag.

[8] Han J. and Kamber M. (2006) Data Mining: Concepts and Techniques, Morgan Kaufmann Publishers.

[9] Bishop C.M. (1995) Neural Network for Pattern Recognition, Oxford University Press., New York

[10] Haykin S. (1999) Neural Networks: A Comprehensive Foundation, Upper Saddle River, NJ, Prentice Hall, 2nd ed.

[11] Lu H., Setiono R. and Liu H. (1996) Effective data mining using neural networks, Knowledge and Data Engineering, IEEE Transactions, vol. 8, pp. 957-961.

[12] Chang E. (2009) Large-scale Data Mining Tutorial, Director, Google Research, China, CIKM.

[13] Tan A. X., Liu V. L., Kantarcioglu M., Thuraisingham B. (2010) A Comparison of Approaches for Large-Scale Data Mining, Tech. report UTDCS-24-10, University of Texas.

[14] Papadimitriou S., Sun J., Yan R. (2010) Large-scale Data Mining, tutorial, IBM Research.

[15] Šerbedžija N. (1996) Simulating Artificial Neural Networks on Parallel Architectures, IEEE Computer, Special Issue on Neural Computing, vol. 29, no. 3, pp.56–63.

[15] Margaritis K.G., Adamopoulos M., Goulianas K., Evans D.J. (1994) Artificial neural networks and iterative linear algebra methods, Parallel Algorithms and Applications, (3), pp. 31-44.

[16] Margaritis K.G. (1995) On the systolic implementation of associative memory artificial neural networks, Parallel Computing, 21, pp. 825-840.

[17] Misra M. (1997) Parallel Environments for Implementing Neural Networks, Neural Computing Surveys, 1, pp. 48-60.

[18] Pethick M., Liddle M., Werstein P. and Huang Z. (2003) Parallelization of a backpropagation neural network on a cluster computer. Fifteenth IASTED International Conference on Parallel and Distributed Computing and Systems; Marina del Ray, CA; USA; 3-5 Nov. pp. 574-582.

[19] Suresh S., Omkar S.N., and Mani V. (2005) Parallel Implementation of Back-Propagation Algorithm in Networks of Workstations, IEEE Transactions on Parallel and Distributed Systems, vol. 16, no. 1, pp. 24-34.

2 Related concepts and background

2.1 Parallel Computing Technology

Parallel computing uses many computers to solve a problem. Nowadays multi-computers and multi-processors are available in many forms. Numerous aspects of our daily life operations are benefited from parallel implementations. Parallel computer architectures especially those used for data-intensive applications, like data mining, are usually classified into several categories [1], namely shared-memory, shared-disk, shared-nothing, and shared-something architectures.

A **shared-memory** system is a natural extension of the CPU architecture where several processors are connected with shared to all main memory, with a uniform address space, and shared to all disks. The processors access the memory through a memory bus. The communication between them is fast, since one processor can write something in memory and another can read it instantly. These multi-processors are scale well for matrix-matrix operations, while plain vector-vector operations are not scalable owning to common bottlenecks in multi-threading. For this scaling up limitation a number of CPUs larger than 32 is not so common.

In a **shared-disk** system the distributed processors share the disks, but each one now has its own local main memory. That is the main memory space is distributed and each processor is independent and has no access to the main memory of the others. Processors usually communicate via message passing through the communication network. They are scaling better than the share-memory since there is now no single memory bus to be congested, and bottleneck effect can appear only in their interconnecting network. This architecture is also used in many database systems like Oracle servers and IBM servers.

In a **share-nothing** system (often called distributed-memory system or cluster), each processor has its own local main memory and disks. Data storage remains local. The processing units are communicating among each other via their interconnecting network. It is most scalable architecture and the only weakness is the communication cost through the network. This architecture ranges from workstation farms to Massively Parallel Processors and is also used in Teradata servers. The network is a common Ethernet for workstation farms while a fast system bus is commonly used in Massively Parallel Processors.

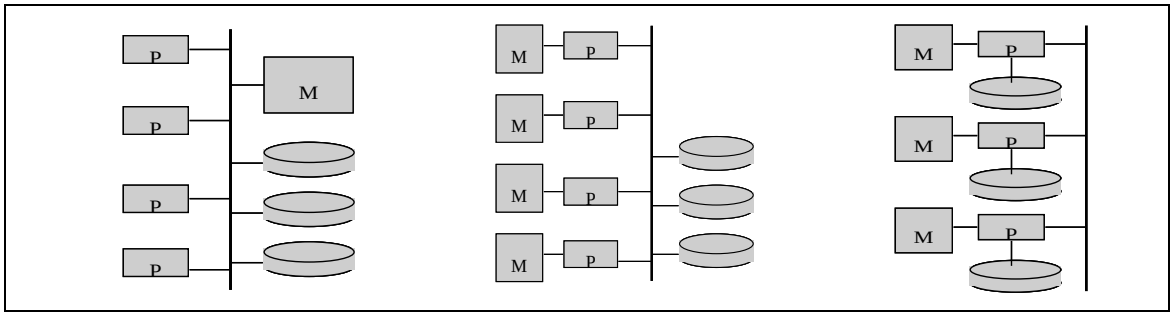


Figure 2.1 Shared Memory Shared Disk Shared Nothing

2.2 Parallel Data Mining

Parallel data mining algorithms [2] [3] [4] [5] can cope with the increasing demands of memory and speed. Adding parallel CPUs in the problem more data will be processed more models will be build and the accuracy will be improved. However, it is not easy to parallelize. In the field of parallel data mining many complexities appear depending on the various forms of parallelism.

2.2.1 Data-parallelism or task-parallelism

From the view of tasks or data one form is known as task-parallelism or data-parallelism [2] [3]. Data-parallelism splits the data across several processors and executes the same task (operation or instruction) on the different data partitions concurrently. Task-parallelism (also called operation parallelism or control parallelism) splits the tasks across several processors and executes the different sub-tasks (instructions) on the same dataset concurrently. How to partition the tasks is the common question here. There are three components: 1) accessing the dataset, 2) computations, 3) communications between processors. These three components are in a mutual trade-off. More partitioning of computations creates more communications and more requirements in data accessing. Finding an efficient parallel algorithm depends on the balance between these three components and their total cost.

In general, we need the ratio communications/computations to be low. The reason is that the computation speed (instructions per second) of the processors is much higher than the speed of communications (data per second) between them. Data-parallelism attacks efficiently a problem of many data. In this case if few instructions are concurrently communicated per iteration then the ratio tasks/data is low. Task-parallelism attacks efficiently a problem of many tasks (instructions). In this case if few data are concurrently communicated per iteration then the ratio data/tasks is low.

It is worth mentioning that task-parallelism and data-parallelism are not mutually exclusive. When both can be used simultaneously then such a hybrid task-data parallel system can speedup the operation. Although these designs are quite challenging, when implemented they are very efficient algorithms. Such task or data parallel algorithms are presented in chapters 4, 5 and 6.

2.2.2 Intra-model parallelism or inter-model parallelism

A data mining model describes a specific aspect of a dataset. For given input data the model produces outputs. Examples are clustering models, classification models and regression models. A clustering model describes the data as groups of objects such that the objects in a group will be similar to one another and different from the objects in other groups. A classification model describes the class attribute as a function of the values of the other attributes in the data. A regression model describes how the independent variables of the data can be used to determine a dependent variable (target). Building a data mining model consists of identifying the relevant independent variables and minimizing the generalization error. Finding the model with the least error might require the construction of hundreds of models and the selection of the best.

The training methods search for the best models and their best parameters. A model almost always has free adjustable parameters [5] [6] which are used-defined. In

parameter searching the algorithm seeks for those free adjustable parameters that optimize the model performance. In model searching the algorithms seeks for the appropriate model and for any model architecture it finds it applies parameter searching also. These two searches are very time-consuming.

Hence, from the view of model construction there is intra-model parallelism that uses several processors to build one model using a parallel algorithm, and inter-model parallelism, where each processor builds its own model, and all together provide (in parallel) a number of independently generated models that will be used for model selection.

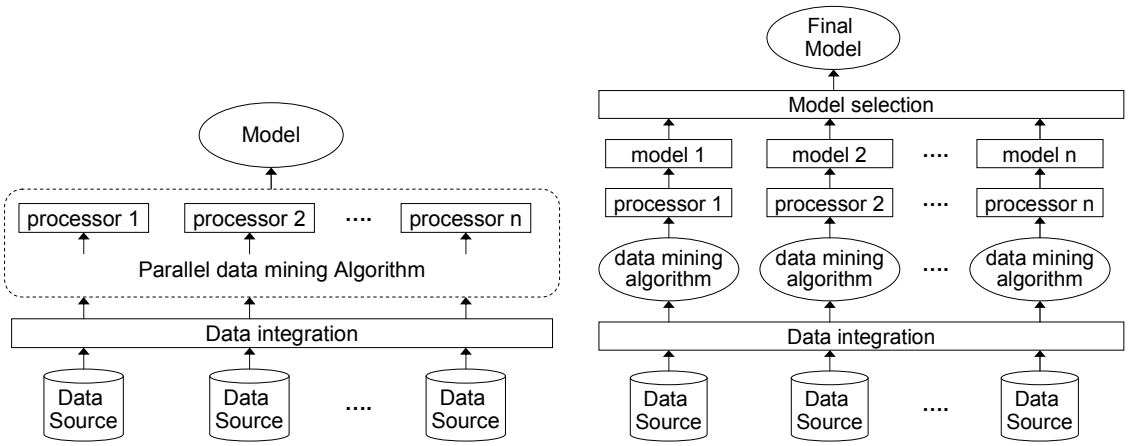


Figure 2.2. (a) Intra-model parallelism, (b) inter-model parallelism

2.2.3 Challenges in parallel data mining

Normally, high performance data mining algorithms first examine the underlined architecture-shared memory systems, or distributed memory systems or clusters of workstations-and then they try to address typical challenges in parallel processing like data locality, load balancing of computations, minimizing communication between processors etc. Several other challenges are imposed when taking into account specific algorithms for data mining tasks.

The first is efficient parallel versions of known sequential algorithms where we may not require searching the whole parameter space. Secondly is inter-model parallelism where many processors seek to construct many data mining models possibly independent. There is a need to investigate if the models are truly independent from each other or if there are hierarchical dependencies etc. Thirdly is the challenge of intra-model-parallelism where many processors, with efficient load balancing and minimum communications between them, construct a single model. There is the need to investigate how the construction of several models can use this intra-model inside it. Next is parallelism in search for the best model (model selection). Then is parallelism in exploit speedups of the final model towards unknown data, as well as parallelism in meta-models and ensemble systems.

For the exploration of many challenges in parallel and distributed data mining we will use models of Artificial Neural Networks whose operation is not a 'black box' but the final structure and architecture can be interpreted easily and clearly in terms of statistical analysis, or locality principles like the Extreme Learning Machines, Radial Basis Functions Neural Networks, Probabilistic Neural Networks and Regularization

Networks, and that seem to be ideal candidates for mapping studies in distributed memory processors and high performance systems.

2.3 Parallel Neural Network Learning

Artificial Neural Networks (ANNs) are parallel, distributed and adaptive information processing structures which are consisted of neurons aligned in layers. Neurons can carry out localized information processing operations. Neurons in one layer are connected, via weighted unidirectional connections, with the neurons in the next layer and so on. Each neuron has a single output signal that branches into all neurons of the next layer. Neural network learning modifies all the connection weights in order to create a model that describes the given training set.

To improve the Artificial Neural Networks (ANNs) performance, it is necessary to use the hardware platform as much as possible via parallelism in learning, testing and operation. Implementation of parallel ANNs [7] [8] [9] is not a straightforward task. In general it requires complete control on the Neural Network software architecture [8] in order to reproduce specific parallel structures and control mechanisms. In [8] a parallel Back-Propagation in Networks of Workstations was studied. The work in [7] implements a parallel feed-forward neural network by dividing up training sets among the processors. The work in [9] focuses on the parallel implementations of neural networks on SIMD architectures. Since the architecture and training algorithms of each neural network model defines its possible parallel solutions, and as they differ from model to model, it is difficult to have a generic solution.

Exploitation of parallelism in parallel systems can be achieved in many different levels such as bit-level parallelism, instruction-level parallelism, data parallelism, task parallelism and system-level parallelism. In the same spirit, structuring approaches for implementing parallelism in ANNs propose general guidelines (see taxonomy review in [10]) on how to break an ANN structure from coarse grained to fine grained pieces in order to run them in parallel [11].

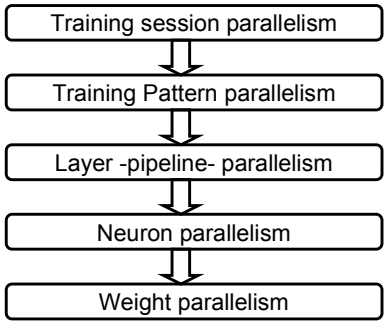


Figure 2.3. Levels of parallelism

Session parallelism places different training model sessions on different processors. A given ANN is trained simultaneously with different learning parameters in the same training dataset. Hence, like the inter-model parallelism it builds several different ANN models simultaneously and in parallel on different processors for learning and testing in order to find the best model among them. **Data parallelism** (or pattern-parallel), is implemented by simultaneous learning in different training data examples within the same session. A dataset is split among processors and a corresponding neural network model, same for all processors, is trained simultaneously. Hence the calculations for the weights are performed in parts, locally inside each processor, and

at the end of each learning cycle (a pass through all data) these partial results are merged. **Layer parallelism** provides concurrent computation when the neural network has many layers. The different layers are distributed on different processors and are pipelined so that learning examples are fed through the network in a way that each layer works in a different training example simultaneously in the pipeline. It works only when several layers exist. **Neuron parallelism** is the most commonly used type of parallelism for both training and operation. Neuron parallelism splits and distributes the ANNs among the different processors. The distributed neurons perform weighed input summation and other computations in parallel. This typical biologically realistic structured parallelism also called network partitioning, is working well for moderate communication per computation ratios. **Weight parallelism** refines Neuron parallelism allowing the simultaneous calculation of each weighed input. This form of parallelism which is at the level of each synapsis can be implemented in most of ANN models. However, depending on the ANN type and train method, this fine-grained implementation can be many times proved to lead to performance slowing down, due to thread overload and the bottleneck effect. **Bit parallelism** is the last approach, where each bit of the calculations is processed in parallel, and is a hardware dependent solution. **Hybrid parallelism** is also possible.

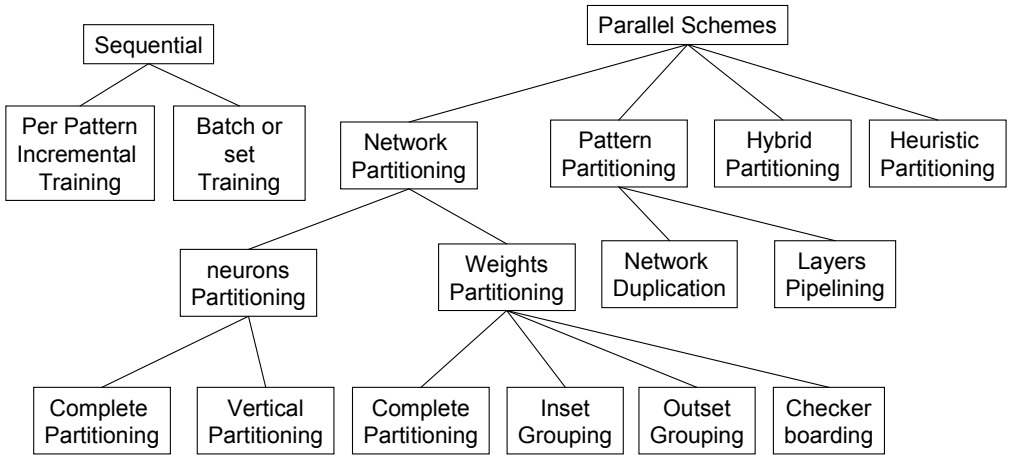


Figure 2.4 (from [12]). A map of Parallel Schemes for partition Neural Networks on several processors. In chapter 3 we examine pattern partitioning and in chapters 4, 5 and 6 we examine neurons partitioning together with pattern partitioning.

Which neurons to partition is an issue. A typical Feed Forward Neural Network has three layers, consist of input neurons, hidden neurons and output neurons, with two weight matrices **C** and **W** in between. The hidden layer transforms the inputs into the new space and the output layer sums up the hidden neuron responses. The input layer neurons are actually the data features and increasing data dimension usually has sub-linear impact on training time. The neurons in the output summation layer are intrinsically linear and increasing them has linear impact on training time. The hidden layer neurons are highly non-linear. Given L hidden neurons the computational complexity of the training phase is $O(aL^{\beta})$, with $a>1$ and $\beta>1$, and is super-linear. Increasing the hidden neurons has super-linear impact on training time depending also on network complexity. Assign them to different processors is the most important issue here. On a typical hidden neurons partitioning, one sends portions of them to different processors. Their weights follow those neurons.

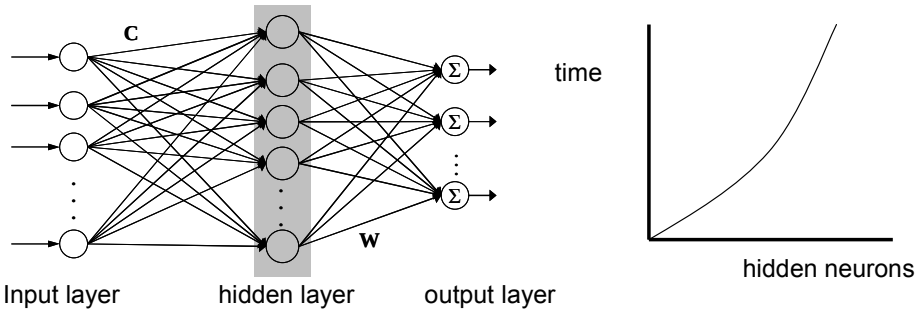


Figure 2.5. Location of hidden layer neurons and their training time impact which is always much larger than linear.

2.4 Distributed Data Mining & Ensemble Learning

Large data repositories are usually distributed. Then a distributed data mining algorithm must be able to cope with very large and high dimensional data sets that might be geographically distributed and stored in different repositories. Distributed data mining on systems that are more loosely coupled than parallel environments studies the exploitation of distribute data and network computational resources. Distributed memory machines [6] are multicomputers having disjoint address memory space. Here every processor has explicit access to its own local memory and hard disks. The Communication between processors is through message passing, commonly using a typical interface like the MPI parallel programming standard.

Distributed memory machines provide usually better scalability due to their completely parallel I/O channels but their communication (message-passing) is much more expensive on these loosely coupled systems. Therefore efforts for optimization are concentrating on minimize the number of messages exchanged. Target environments for distributed data mining include workstation clusters, distributed memory clusters or even geographically distributed systems.

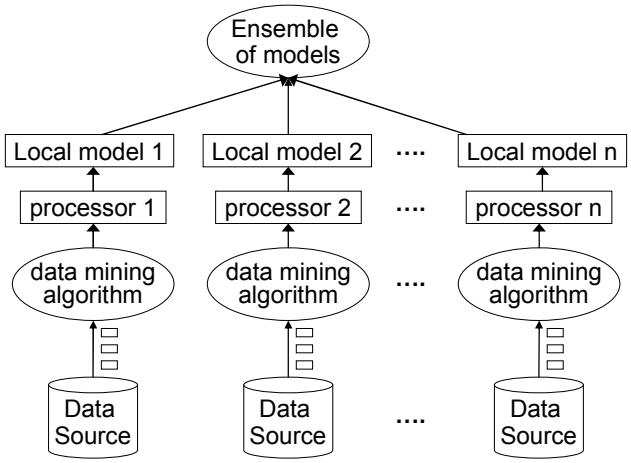


Figure 2.6. Distributed processing that combines several learning models to create an ensemble

Typically distributed data mining algorithms [13] include local model analysis accompanied by the merging of local models in order to produce the final ensemble of models each one describes one aspect of data. The same algorithm operates on a different data region and produces a different local model. Distributed classifier systems, known as ensembles of classifiers, are such paradigms of distributed learning. The distributed models are integrated using weighted voting or other meta-learning schemes.

2.5 Hierarchical Learning

Hierarchical learning questions appear many times in literature [14] (see also other related references in chapter 10). The issues concern hierarchical systems are open, and likely to be of great importance for the next developments of efficient predictors. A hierarchical neural network (see several related works in chapter 10) tries to break up a complex task into a series of simpler and faster computations at each level of the hierarchy. Hierarchical structures intrinsically facilitate parallelism, and naturally alleviate many common scalability problems through the divide-and-conquer strategy. We propose a new hierarchical Markovian RBF Neural Network approach, suitable for parallelization studies. The Markovian property permits the hierarchical RBF function in a fashion that supports a clear recursion. We further analyze the general framework and some potential training methods. All the hidden 'neurons' in the hierarchy levels are composed of fully functional RBF in nature Neural Networks having the two classical synaptic weight sets, namely the **C** matrix that holds the RBF centers and the **W** matrix that holds the linear output weights. Thus the Neural Network operation is exactly the same at all levels of the hierarchical integration.

2.6 Local Learning

Local learning algorithms emerge as an alternative to the global learning strategy. Local learning algorithms use a neighbourhood of training data close to a given testing query point in order to learn the local parameters and create on-the-fly a local model specifically designed for this query point. A local learning framework as described in [15] is: "For each testing point **q** do the following: (1) select the training examples closely located in the vicinity of **q**, (2) train a neural network $f_k(\mathbf{q})$ with only these examples and (3) apply the resulting neural network to the testing pattern". Hence, while a global learning algorithm creates a global model $f(\mathbf{x})$ using all the training data, a local learning algorithm builds on the fly a different local model $f_k(\mathbf{q})$ for a different testing point **q** using only the training points most nearest to **q**. After the local learning approach introduced in [15] it was adapted for various tasks and brought breakthrough performance in many application domains (see related works in chapter 11). We explore in this thesis four local learning versions of Regularization Networks (RN) and we find several options for improving their online prediction performance, both in accuracy and speed.

References for chapter 2

[1] Rajasekaran S. and Reif J.H., (2006) Handbook of Parallel Computing: Models, Algorithms and Applications, CRC Press.

[2] Freitas A.A., (1998) A Survey of Parallel Data Mining. In Proceedings of 2nd International Conference on the Practical Applications of Knowledge Discovery and Data Mining, 287–300

[3] Skillicorn D. (1999) Strategies for Parallel Data Mining. IEEE Concurrency, 26–35.

[4] Jianwei L, Ying L., Wei-keng L., Alok Ch., (2006) Parallel Data Mining Algorithms for Association Rules and Clustering. In Rajasekaran S. and Reif J.H. eds, Handbook of Parallel Computing: Models, Algorithms and Applications, CRC Press.

[5] Kumar V. et al, (2007) Workshop on High Performance Data Mining, Omaha, USA.

[6] Karniadakis G. and Kirby R. II, (2003) Parallel Scientific Computing in C++ and MPI. Cambridge University Press.

[7] Morchen F. (2004) Analysis of speedup as function of block size and cluster size for parallel feed-forward neural networks on a beowulf cluster. IEEE Transactions on Neural Networks 15(2), 515–527.

[8] Suresh S., Omkar S.N., and Mani V. (2005) Parallel Implementation of Back-Propagation Algorithm in Networks of Workstations. IEEE Transactions on Parallel and Distributed Systems, 16(1), 24–34.

[9] Blas A.D., Jagota A., Hughey R. (2005) Optimizing neural networks on SIMD parallel computers, Parallel Computing 31, 97–115.

[10] Šerbedžija N. (1996) Simulating Artificial Neural Networks on Parallel Architectures. IEEE Computer, Special Issue on Neural Computing, 29(3), 56–63.

[11] Pethick M., Liddle M., Werstein P., and Huang Zh., (2003) Parallelization of a backpropagation neural network on a cluster computer. Fifteenth IASTED International Conference on Parallel and Distributed Computing and Systems; Marina del Ray, CA; USA; 3-5 Nov. 574–582.

[12] Sudhakar V., Siva C., and Murthy R. (1998) Efficient Mapping of Back-Propagation Algorithm onto a Network of Workstations. IEEE Trans. Man, Machine, and Cybernetics—Part B: Cybernetics, 28(6), 841–848,

[13] Park B. and Kargupta H., (2002) Distributed Data Mining: Algorithms, Systems, and Applications. Data Mining Handbook, 341–358. IEA.

[14] Poggio T., and Smale S., (2005) The Mathematics of Learning: Dealing with Data. Studies in Fuzziness and Soft Computing, 180, 3–19.

[15] Bottou L, Vapnik V (1992) Local learning algorithms. Neural Computation 4(6), 888–900

3 Scalable Model Selection in Extreme Learning Machines via matrix decompositions

This chapter, in essence, describes a reason for preferring purely parallel algorithmic designs instead of just running a sequential program in many processors 'in parallel'. The machine learning models usually require some form of model selection for their parameters. The model selection phase creates thousands models and selects the best. Why even bother to design a parallel algorithm when the simple alternative is to run each sequential model in a different processor, and gather the results. The university grid runs a lot of of those kinds of programs. The reason is that while the models are different, they have some computations same to all models. Instead of repeatedly run again and again these computations thousands times, we can save a huge amount of computational cost by considering the whole model selection phase as one algorithm and then finding a method to parallelize it.

The most representative paradigm is the model selection in the training phase of Extreme Learning Machine (ELM) algorithms. ELMs use two adjustable hyper-parameters, namely the number L of hidden nodes and the regularization parameter C. The typical model selection strategy that is applied in most ELM papers depends on a cross-validation and a grid search in order to select the best pair of the hyper-parameters (L, C) that minimizes the validation error. However, by testing only 30 values for L, 30 values for C, via 10 fold cross-validation then the learning phase must build 9000 different ELM models. Since these models are not actually independent the essence of managing and drastically reducing the computational cost of ELM model selection relies on matrix decompositions that avoid direct matrix inversion and appropriate cross-validations. Still, one can find many matrix decompositions and many cross-validation versions that result in several combinations. We identify these combinations and analyze them theoretically and experimentally in order to discover which is the fastest. We compare Singular Value Decomposition (SVD), Eigenvalue Decomposition (EVD), Cholesky decomposition and QR decomposition which produce matrices (orthogonal, eigen, singular, upper triangular) that are re-usable many times. These decompositions can be combined with different cross-validation approaches and we present a direct and thorough comparison of k-fold cross-validation versions as well as leave-one-out cross-validation. By analyzing the computational cost we demonstrate theoretically and experimentally that while the type of matrix decomposition plays one important role another equally important role plays the version of cross-validation. A rather scalable version of k-fold cross-validation as simulated here saves a huge amount of computational time.

3.1 Introduction

Extreme Learning Machines (ELM) [1] [2] [3] have attracted a lot of interest, owning to their simplicity, generalization capability and straightforward learning models. ELM is a single-hidden-layer feedforward neural network with randomly generated hidden neurons, in which only the linear output weights $\boldsymbol{\beta}$ that connect the hidden layer with the output layer must be tuned. ELMs have been successfully used for regression, classification, feature selection and many other machine learning and artificial intelligence applications (for reviews see [2] [3] [4] [5]).

The ELM model has a hidden layer with L randomly generated neurons, which map the input data into the L-dimensional ELM random feature space. During ELM training [2][3] the hidden layer mapping matrix $\mathbf{H}$ is produced from the training data and the output linear weights $\boldsymbol{\beta}$ are calculated by the regularized least squares solution $\boldsymbol{\beta} = (\mathbf{H}^T\mathbf{H}+\mathbf{I}/C)^{-1}\mathbf{H}^T\mathbf{y}$, where $\mathbf{I}$ is the identity matrix, $\mathbf{y}$ is the vector of desired targets and C is the regularization parameter. A critical issue during ELM training [2][3] is how to select the two appropriate adjustable hyper-parameters of the model, namely the number L of hidden neurons and the regularization parameter C. The regularization term C makes the solution more robust and improves the generalization performance of ELM. A very small C may disturb the information on the regularized gram matrix $\mathbf{H}^T\mathbf{H}+\mathbf{I}/C$, while a large C may not be effective in improving the generalization performance of ELM. The optimal number L of hidden neurons is also difficult to choose. An ELM with small L, and in consequence few hidden neurons, may have poor accuracy, while an ELM with very large L may become slow and over-fitted. Therefore, the best pair (L, C) of the free adjustable model parameters that produces the best ELM model must be found during the training phase.

The search for the best model parameters is called model selection [6], in which cross-validation [7] is commonly employed to choose the optimal parameters from a set of candidate values. Typically, either k-fold Cross-Validation (CV) or leave-one-out Cross-Validation (CV) is used. In the ELM case both types generate a set of several candidate values of L and a set of several candidate values of C and perform a grid search in order to select the best pair of the hyper-parameters (L, C) that minimizes the validation error of the ELM model. A minimum set of several different C values for testing, may have 30 elements $\{2^{-14}\, 2^{-13} \ldots 2^{14}\, 2^{15}\}$ and a full grid search can use up to 50 elements $\{2^{-24}\, 2^{-23} \ldots 2^{24}\, 2^{25}\}$. A minimum set of different L values may start from 30 elements {5 10 15 ... 150} and depending on the problem size may go beyond 40 values. Thus, by trying a minimum of 30 different values for L, 30 values for C, and k=10 fold cross-validation then the learning phase must build 9000 different ELM models in order to test their error. Many times this cost must be multiplied by the several experimental rounds that are performed, as a rule, in any ELM comparison when one has to report an average accuracy. While fast least squares solutions for solving a single ELM model exist in the literature these solutions have to be executed hundreds of times during cross-validation. For large scale applications we must economize model selection via cross validation as much as possible. To this end we compare existing matrix decompositions as well as different cross validation methods for the solution of the same problem, which is the search of the best pair (L, C) of ELM model parameters

The essence of the compared ELM model selection solutions via cross-validation relies on all the well known matrix decompositions, namely Singular Value Decomposition (SVD), Eigenvalue Decomposition (EVD), Cholesky decomposition and QR

decomposition that produce matrices that are reusable many times. These matrix decompositions have not been previously compared in the literature for revealing which is fast and most efficient for selecting the best ELM model parameters.

While one part of the comparisons is based on the aforementioned matrix decompositions, another part compares different cross-validation versions, which are: the virtual leave-one-out Cross Validation, standard *k*-fold Cross-Validation, efficient *k*-fold Cross-Validation, as well as scalable *k*-fold Cross-Validation. These versions of cross-validations have not been previously compared side-by-side for fast ELM model selection. To our knowledge this is the first time of reporting detailed comparison results of their combinations with the matrix decompositions. Contrary to what is was believed, in the ELM case a scalable implementation of *k*-fold Cross-Validation plays as much an important role as the choice of matrix decomposition method we use. This is proved by analyzing the cost of operations of all the algorithms in order to find out which steps can be performed with a maximum economy of computations.

We go further by examining all the required matrix-matrix multiplications and matrix-vector multiplications in the computational steps inside either *k*-fold cross-validation or leave-one-out cross-validation procedures together with the required matrix decompositions. When the training dataset is large and expensive matrix computations must be performed for validating each value of C or L even if the computational steps are wrapped around the known matrix decompositions. There are however few practical guidelines that we can follow:

- *Caching*: identify all matrices and vectors whose elements are required many times during the computation steps of model selection and cache them. For example if the hidden layer mapping matrix $\mathbf{H}$ is stored then there would be no need to re-compute it many times. The same holds in caching the matrix $\mathbf{H}^T\mathbf{H}$.
- *Costly computations*: Identify the most costly computations and separate them from the inexpensive ones. For example, in ELM case the $\mathbf{H}^T\mathbf{H}$ is one costly matrix-matrix computation with cost L^2N for N samples, while all matrix-vector multiplications like $\mathbf{H}^T\mathbf{y}$ are considered inexpensive.
- *Decomposition*: utilize matrix decompositions in order to produce reusable matrices that can be employed within the cross-validation steps. For example, Singular Value Decomposition of $\mathbf{H}=\mathbf{U}\mathbf{\Sigma}\mathbf{V}^T$ splits the matrix $\mathbf{H}$ into reusable matrices $\mathbf{U}$, $\mathbf{\Sigma}$ and $\mathbf{V}$.
- *C dependence*: In the cases of testing many regularization values C_i, some computation steps may depend on C while others not. During searching within different C_i values only the computations that depend on C are repeated many times. These computations must be based on reusable diagonal matrices, like $\mathbf{\Sigma}$ or $\mathbf{\Lambda}$, which contain singular or eigen values that depend on C_i, since this parameter regularizes only these values.
- *Incremental computations*: find out which of the most costly computations are incremental with respect to increments δL of the L parameter, meaning that by adding an extra hidden neuron and column in $\mathbf{H}$ the past values remain the same. For example the matrix $\mathbf{H}^T\mathbf{H}$ is a result of incremental computations.
- *Computational series*: identify results that are produced by a series of consecutive computation steps.

- *Scalability rule*: discover which of the most costly computations are incremental with respect to L increments and independent from C. We have found that in all the alternatives for ELM cross-validation that will be presented next, the scalable are only those in which the most costly term L^2N obeys this rule.

Among all the guidelines the scalability rule has been proven most useful in revealing the fastest algorithms. However, we must follow the other guidelines first, in order to decompose the problem, otherwise it would be very difficult to find out which of the most costly computations are independent when C changes and incremental when L increases. We compare 10-fold Cross-Validation (efficient and scalable versions) as well as leave-one-out Cross-Validation via the HAT matrix. All alternatives for ELM cross-validation are presented in 7 algorithms (we have examined other variants also and found out that these are the most representative). In essence, by analyzing these algorithms with respect to their most costly computation steps, we demonstrate why some of them are slow, others are fast, and others are fast and scalable.

Slow algorithms contain costly computations that are not incremental with respect to L increments and not independent from C. Fast algorithms contain costly computations that are not incremental with L but independent from C. Fast and scalable algorithms contain some costly computations that are incremental with L and independent from C, and others that are not incremental with L but independent from C.

While the common belief was that leave-one-out cross-validation using HAT matrix should be faster than k-fold cross-validation we find that in ELM cross-validation none of the possible alternatives that use the HAT matrix obey the scalability rule completely. Surprisingly, only the last algorithm, which uses scalable k-fold Cross-Validation and Eigen Value Decomposition, has all its costly computation steps incremental with respect to L and independent from C. That is way it is most scalable.

The rest of the chapter is organized as follows. Section 2 overviews the ELM model, the related model selection solutions that use k-fold cross validation or leave-one-out cross validation and the basics of matrix decompositions. Section 3 gives the ELM model selection algorithms based on Cholesky Decomposition. Section 4 presents those algorithms that use Singular Value Decomposition. Section 5 discusses a combination of QR decomposition and Singular Value Decomposition. Section 6 presents algorithms based on Eigen Value decomposition. Section 7 gives the summary of the theoretical comparisons of the computational cost and the analysis of all the algorithms. Section 8 describes extensive experimental simulations that support the theoretical findings. Finally section 9 summarizes our conclusions and gives some future directions.

3.2 Preliminaries and Relevant Work

In the mathematical formulas a capital bold letter will symbolize a matrix, while a lowercase bold letter will symbolize a vector. Italic letters will denote scalar variables. Fig. 3.1 shows the meaning of several symbols and notations used.

N number of training examples	$h_{n,j}$ a scalar element n^{th} row and j^{th} column of $\mathbf{H}$
L number of ELM random hidden neurons	$h_j(\mathbf{x})$ the activation function $G(\mathbf{a}_j, b_j, \mathbf{x})$
D number of data features	$\boldsymbol{\beta}$ the vector of output weights
$\mathbf{H}$ the hidden layer output matrix of size $N \times L$	$\mathbf{e}$ the residual vector given by $\mathbf{y} - \mathbf{H}\boldsymbol{\beta}$
$\mathbf{H}_{-\mathbf{f}}$ matrix $\mathbf{H}$ without the f fold	C the regularization parameter
$\mathbf{H}_{\mathbf{f}}$ matrix $\mathbf{H}$ with only the f fold	HAT the HAT matrix

Figure 3.1. Common symbols and notations.

3.2.1 The ELM model

Given a training set of N examples $\{\mathbf{x}_n, y_n\}_{n=1}^{N}$ in a d-dimensional space the ELM topology is given in fig. 3.2. The regularized ELM model solution [2][3][5] has a straightforward three-step training which consists of:

I) randomly generating a set of L hidden neuron parameters $\{\mathbf{a}_j, b_j\}_{j=1}^{L}$, where $\mathbf{a}_j$ is the input weight vector and b_j is the bias of the jth hidden node,

II) compute the hidden layer mapping matrix $\mathbf{H}$ (of size $N \times L$), with elements $h_j(\mathbf{x}_n) = G(\mathbf{a}_j, b_j, \mathbf{x}_n)$, which maps the N examples into the L hidden neuron responses,

III) solve the linear output weights $\boldsymbol{\beta}=(\mathbf{H}^T\mathbf{H}+\mathbf{I}/C)^{-1}\mathbf{H}^T\mathbf{y}$, where $\mathbf{I}$ is the identity matrix and C the user-defined positive regularization parameter that stabilizes the solution.

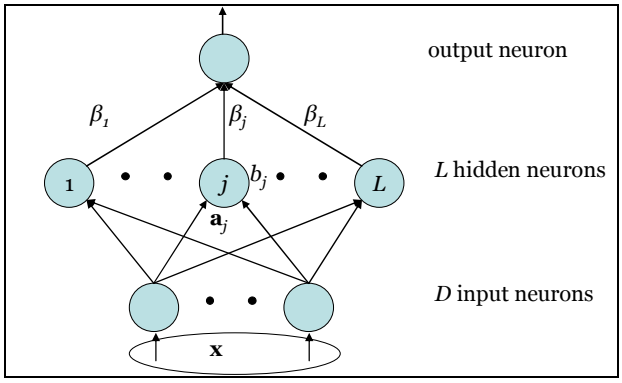


Figure 3.2. Extreme Learning Machine (ELM) topology with L hidden neurons (nodes).

The ELM model with L hidden neurons is finally given by:

$$f_L(\mathbf{x}) = \sum_j^L \beta_j h_j(\mathbf{x}) = \mathbf{h}(\mathbf{x})\boldsymbol{\beta} \tag{3.1}$$

and is defined by the output weight vector $\boldsymbol{\beta} = [\beta_1\ \beta_2 \ldots \beta_L]^T$ and the hidden layer response vector

$$\mathbf{h}(\mathbf{x}) = [h_1(\mathbf{x})\ h_2(\mathbf{x})\ \ldots.\ h_L(\mathbf{x})] = [G(\mathbf{a}_1, b_1, \mathbf{x})\ G(\mathbf{a}_2, b_2, \mathbf{x})\ \ldots.\ G(\mathbf{a}_L, b_L, \mathbf{x})] \tag{3.2}$$

For N training examples the hidden layer mapping matrix $\mathbf{H}^{N\times L}$ is given by:

$$\begin{bmatrix} G(\mathbf{a}_1,b_1,\mathbf{x}_1) & G(\mathbf{a}_2,b_2,\mathbf{x}_1) & \ldots.. & G(\mathbf{a}_L,b_L,\mathbf{x}_1) \\ G(\mathbf{a}_1,b_1,\mathbf{x}_2) & G(\mathbf{a}_2,b_2,\mathbf{x}_2) & \ldots.. & G(\mathbf{a}_L,b_L,\mathbf{x}_2) \\ \vdots & \vdots & \vdots & \vdots \\ G(\mathbf{a}_1,b_1,\mathbf{x}_N) & G(\mathbf{a}_2,b_2,\mathbf{x}_N) & \ldots.. & G(\mathbf{a}_L,b_L,\mathbf{x}_N) \end{bmatrix}$$

The ELM algorithm has the universal approximation capability for several types of feature mapping functions (whose parameters are randomly generated), like:

- Sigmoid function $G(\mathbf{a},b,\mathbf{x})=1/(1+\exp(-(\mathbf{a}\ \mathbf{x}+b)))$
- Gaussian function $G(\mathbf{a},b,\mathbf{x}) = \exp(-b\|\mathbf{x}-\mathbf{a}\|^2)$
- multi-quadric function $G(\mathbf{a},b,\mathbf{x}) = (\|\mathbf{x}-\mathbf{a}\|^2+b)^{1/2}$

The cost of operations for computing the output weights $\boldsymbol{\beta}$ for each ELM model is $LNd+L^2N+L^3$, which corresponds to the computations for the hidden layer mapping matrix $\mathbf{H}$, the multiplications necessary to calculate the matrix $\mathbf{H}^T\mathbf{H}$, and the least squares solution.

Different ELM models have different generalization performance. Hence the problem in question is to how to generate all the required models and validate their performance in a cost-effective way for model selection.

3.2.2 Model Selection solutions

K-fold Cross-Validation (k-fold CV)

Typical model selection approaches search for the minimum cross-validation estimate [7][8] of some performance statistic, like the mean squared error, in order to find the best values of the model parameters. Training with the best of these values produces the most unbiased model that generalizes well on new data. The *k-fold Cross-Validation* (*CV*) is the most common performance estimate and has been used in several papers for ELM model selection (see for example [2][3][5][9] and several other references therein).

The *k-fold Cross-Validation* (*CV*) procedure partitions the training dataset into k disjoint subsets, or k-folds. Then (k–1) of these folds are used for training the model with the given parameters, and the remaining k fold is used for validating the performance. Thus for one pair (L,C) of ELM parameters a series of k models are created each using a different combination of (k–1) folds. The model selection criterion is taken as the average performance of these k models.

Generalized Cross-Validation Error and HAT matrix

Generalized Cross-Validation (GCV) of Golub and co-workers [10][11][12] is an alternative to Cross-Validation for the standard regression model $\hat{\mathbf{y}} = \mathbf{H}\boldsymbol{\beta}$, that is given in terms of the regression matrix $\mathbf{H}$ and the weights $\boldsymbol{\beta}$. *GCV* uses the HAT matrix [13], which projects the original target vector $\mathbf{y} = [y_1 \ldots y_n \ldots y_N]^T$ into the estimated output vector $\hat{\mathbf{y}} = [f(\mathbf{x}_1) \ldots f(\mathbf{x}_n) \ldots f(\mathbf{x}_N)]^T$, and satisfies the equality $\hat{\mathbf{y}} = \text{HAT}\ \mathbf{y}$. In terms of the HAT matrix, which puts a "hat" on $\mathbf{y}$ by transforming it into the fitted values $\hat{\mathbf{y}}$, the Generalized Cross-Validation Error is given by [10]:

$$GCV = N\,||(\mathbf{I}-\text{HAT})\,\mathbf{y}||^2 / trace(\mathbf{I}-\text{HAT})^2 \tag{3.3}$$

where the HAT matrix, of size $N \times N$, describes the influence each observed value has on each fitted value. Both the matrices HAT and ($\mathbf{I}$ – HAT) are symmetric and idempotent (meaning that a matrix multiplied by itself is equal to itself) and the HAT matrix is given in terms of the regression matrix $\mathbf{H}$ and the regularized matrix $\mathbf{H}^T\mathbf{H}$ (Golub uses the term $\lambda\mathbf{I}$ in place of $\mathbf{I}/C$) by [13][10]:

$$\text{HAT} = \mathbf{H}(\mathbf{H}^T\mathbf{H}+\mathbf{I}/C)^{-1}\mathbf{H}^T \tag{3.4}$$

GCV is the closed-form expression for the leave-one-out error in regularized least squares models but it requires the trace of $\mathbf{I}$–HAT. The *GCV* estimate is the rotation-invariant version of Allen's [14] PRESS residual (Prediction Error Sum of Squares). Thus instead of *GCV* the faster version of virtual leave-one-out error (PRESS residual via the HAT matrix) is commonly used and this is presented in the next section.

Virtual Leave One Out error and the HAT matrix

Leave one-out cross-validation [15] is the most extreme form of cross-validation that gives an almost unbiased estimate for the error. The sum of squares of the leave-one-out errors is also called the PRESS statistic [14].

In leave one-out cross-validation each partition consists of a single pattern. Suppose we drop the n^{th} observation $\mathbf{x}_n$ from the training dataset and then retrain the learning model $f_{(-n)}()$. With the model $f_{(-n)}()$ constructed with the n^{th} sample omitted from the dataset we use the $\mathbf{x}_n$ to predict the output $\hat{y}_{(-n)} = f_{(-n)}(\mathbf{x}_n)$. Then for each $\mathbf{x}_n$ the leave-one-out residual error is $e_{(-n)}(\mathbf{x}_n) = y_n - \hat{y}_{(-n)}$. This procedure is repeated for each one of the N examples, and then the PRESS statistic proposed by Allen [14] is the sum of squares of the leave-one-out errors:

$$\text{PRESS} = \sum_n^N e_{(-n)}^2(\mathbf{x}_n) \tag{3.5}$$

Obviously it is very time-consuming to repeat the procedure N times, since leave-one-out cross-validation requires building N different models using each dataset of $N-1$ points. This is computationally prohibited. Although each model is different, they are similar since the datasets only differ by a single point. However employing the HAT matrix can save a lot of calculation. One needs not actually re-train the regression model N times. Rather, it is true that the PRESS residual $e_{(-n)}(\mathbf{x}_n)$ is equal to the ordinary residual $e(\mathbf{x}_n)$ which is the one that is found when the model is trained on all examples including $\mathbf{x}_n$, divided by 1 minus the diagonal of the HAT matrix, given by (see details in [16] [17] [18] [19] [20] [21]:

$$e_{(-n)}(\mathbf{x}_n) = e(\mathbf{x}_n)/(1-hat_{nn}) = (y_n-\hat{y}_n)/(1-hat_{nn}) = (y_n-\mathbf{h}(\mathbf{x}_n)\boldsymbol{\beta})/(1-hat_{nn}) \tag{3.6}$$

where the vector $\mathbf{h}(\mathbf{x}_n) = [h_1(\mathbf{x}_n)\ h_2(\mathbf{x}_n)\ \ldots.\ h_L(\mathbf{x}_n)]$ is the n^{th} row of the regression matrix $\mathbf{H}$ (also called the n^{th} hidden layer response vector) and the value hat_{nn} is the n^{th} diagonal element of the HAT matrix given by:

$$hat_{nn} = \mathbf{h}(\mathbf{x}_n)\,(\mathbf{H}^{\text{T}}\mathbf{H}+\mathbf{I}/C)^{-1}\,\mathbf{h}^{\text{T}}(\mathbf{x}_n) \tag{3.7}$$

Recall that the HAT matrix transforms the targets $\mathbf{y}$ into their estimations via $\hat{\mathbf{y}}$=HAT $\mathbf{y}$, from which we can get the ordinary residual vector as $\mathbf{e} = (\mathbf{y} - \hat{\mathbf{y}}) = (\mathbf{I} - \text{HAT})\,\mathbf{y}$. Then in matrix form the PRESS statistic (virtual leave-one-out squared error estimator) from eq. 3.5 and eq. 3.6 is given by [18][20]:

$$\text{PRESS} = \mathbf{e}^2/diag(\mathbf{I}-\text{HAT})^2 = ((\mathbf{I} - \text{HAT})\,\mathbf{y})^2/diag(\mathbf{I} - \text{HAT})^2 \tag{3.8}$$

For model selection in regularized kernel methods the PRESS statistic, also called virtual leave-one-out squared error estimator, has been used in several works like [18] [19] [20][21]. For model structure selection in ELM the PRESS statistic with the HAT matrix has been used in various papers like [22][23][24][25].

3.2.3 Basics of Matrix Decomposition and inversion techniques

There are many methods for matrix decompositions [26][27] (matrix factorizations) that allow expressing a matrix as the product of simpler matrices in order to solve the normal equation $\boldsymbol{\beta} = (\mathbf{H}^{\text{T}}\mathbf{H})^{-1}\mathbf{H}^{\text{T}}\mathbf{y}$, without costly matrix inversions. A common decomposition is Singular Value Decomposition (SVD) via $\mathbf{H} = \mathbf{U}\boldsymbol{\Sigma}\mathbf{V}^{\text{T}}$ that is very accurate, but time consuming when the matrix is large. Although SVD is slow, it has many uses, particularly in the analysis of the problem (the smaller than machine precision singular values) and in the production of the generalized inverse

$\mathbf{H}^{\dagger}=\mathbf{V}(\mathbf{\Sigma})^{-1}\mathbf{U}^{T}$. Another decomposition is QR decomposition $\mathbf{H} = \mathbf{QR}$ which gives an orthogonal matrix $\mathbf{Q}$ and a right triangular matrix $\mathbf{R}$, which are appropriate especially if the basis for the residual space is required. A widely used decomposition for symmetric square matrices is the spectral decomposition called Eigen Value Decomposition (EVD) $\mathbf{H}^{T}\mathbf{H} = \mathbf{Q\Lambda Q}^{T}$ that has been extensively applied in pattern recognition literature [28]. For regularized solutions $\mathbf{H}^{T}\mathbf{H}+\mathbf{I}/C$ that guarantee the positive definiteness (a tiny value to the diagonal ensures positive definiteness) we can safely use the Cholesky decomposition $\mathbf{H}^{T}\mathbf{H}+\mathbf{I}/C=\mathbf{R}^{T}\mathbf{R}$, that gives the factoring solution in terms of a right triangular matrix $\mathbf{R}$.

Besides the previous matrix decompositions several techniques are proposed for direct matrix inversion [29], like a full rank Cholesky decomposition in [30] for fast computation of the Moore-Penrose inverse, or expansions of the Moore-Penrose inverse that can be used to develop effective algorithms [31], or special algorithms that devoted to parallel computation [32]. However the use of such alternative methods cannot take advantage of re-usable matrices that avoid matrix inversion. That is why the aforementioned SVD, EVD, QR and Cholesky decompositions have been applied extensively in kernel methods [17] [18] [19] [20][21] for model selection of the best regularization parameters. In the next sections we will compare the computational steps that are wrapped around the well known matrix decompositions when applied within the Extreme Learning Machine model selection via cross-validation.

3.2.4 Standard problem solution with cross-validation

One trivial but usually preferred method, from our experience, is: *"Send all different ELM models as different independent jobs into the University Grid"*.Typically each processor gets a pair of model parameters (C_i,L_j) and learns a different model. This has a very large computational cost for moderate or large datasets. How many models must be issued? Usually thousands models must be created for only one experimental round.

The not-recommended model selection with naive 10-fold cross-validation for testing say 30 regularization parameters C_i, proceeds by finding for each C the 10-fold cross-validation error using a double loop of the type "For each one C_i of the 30 values of C parameter:, For each one of the f=1 to 10 folds: Solve $\boldsymbol{\beta} = (\mathbf{H}_{-\mathbf{f}}{}^{T}\mathbf{H}_{-\mathbf{f}} + \mathbf{I}/C_i)^{-1}\,\mathbf{H}_{-\mathbf{f}}{}^{T}\,\mathbf{y}_{-\mathbf{f}}$ and find residual $\mathbf{e}_{\mathbf{f}} = \mathbf{y}_{\mathbf{f}} - \mathbf{H}_{\mathbf{f}}\boldsymbol{\beta}$. This not-recommended approach builds 300 matrices $\mathbf{H}^{T}\mathbf{H}$ and 300 models only for testing one L value. Many methods are way much better. A well-organized algorithm can be derived if the "for each fold f" loop is executed first and the "for each parameter C" loop is placed inside it, like the following algorithm 0.

Algorithm 0. Standard 10-fold cross-validation that tests one L parameter and 30 regularization parameters C_i	*Operations*
1. Calculate and store the hidden layer output matrix $\mathbf{H}$	
Split $\mathbf{H}$ into 10 folds as $\mathbf{H} = [\mathbf{H}_1 \dots \mathbf{H}_{\mathbf{f}} \dots \mathbf{H}_{10}]^{T}$	
Split $\mathbf{y}$ into 10 folds as $\mathbf{y} = [\mathbf{y}_1 \dots \mathbf{y}_{\mathbf{f}} \dots \mathbf{y}_{10}]^{T}$	
2. For each one of the f=1 to 10 folds	
2.1 Calculate matrix $\mathbf{H}_{-\mathbf{f}}{}^{T}\mathbf{H}_{-\mathbf{f}}$ without the f fold	L^2N
2.2 Calculate vector $\mathbf{H}_{-\mathbf{f}}{}^{T}\mathbf{y}_{-\mathbf{f}}$ without the f fold	LN
2.3 For each one C_i of the 30 values of C parameter	
2.3.1 Solve weights $\boldsymbol{\beta} = (\mathbf{H}_{-\mathbf{f}}{}^{T}\mathbf{H}_{-\mathbf{f}} + \mathbf{I}/C_i)^{-1}\,\mathbf{H}_{-\mathbf{f}}{}^{T}\,\mathbf{y}_{-\mathbf{f}}$	L^3
2.3.2 find the residual vector for the f fold $\mathbf{e}_{\mathbf{f}} = \mathbf{y}_{\mathbf{f}} - \mathbf{H}_{\mathbf{f}}\boldsymbol{\beta}$	$LN/10$
2.3.3 Add to squared errors $E(C_i,L_j) = E(C_i,L_j) + (\mathbf{e}_{\mathbf{f}})^{T}\mathbf{e}_{\mathbf{f}}$	
3. Select the pair (C_i,L_j) with the minimum error $E(C_i,L_j)$	

Assuming that there are N training examples and L hidden neurons the overall cost of operations in algorithm 0 is $10L^2N + 300L^3 + 10LN$, and it avoids the $300L^2N$ operations which would have been required if we have used the naive 10-fold cross-validation mentioned previously. Still, we must repeat algorithm 0 for each one of the different L_j values of hidden neurons that range from δL to $L_{\max}$ with step δL. All the next algorithms we are going to examine are dealing with such scalability problems.

3.2.5 Incremental computations

Throughout the following sections, the superscript for the rectangular matrix $\mathbf{H}$ will denote its current column size, since the rows that correspond to N samples remain the same for all such matrices. Thus $\mathbf{H}^{Lj}$ means that $\mathbf{H}$ has L_j columns (L_j are the ELM hidden neurons) and N rows. Also for square matrices only one superscript will be used to designate the matrix dimension. The corresponding square regularized gram matrix will be denoted as $\mathbf{G}^{Lj} \equiv [(\mathbf{H}^{Lj})^{\mathrm{T}}(\mathbf{H}^{Lj})]+\mathbf{I}/C$ which has L_j rows and equal in number columns. Also in the following sections, the superscript for the vector $\boldsymbol{\beta}$ will indicate the current length of the vector; hence $\boldsymbol{\beta}^{Lj}$ denotes a vector of length L_j.

The augmented by δL extra columns regression matrix $\mathbf{H}^{Lj}$ will be denoted by $\mathbf{H}^{Lj+\delta L}$. If we augment $\mathbf{H}$ by δL extra columns then the corresponding gram matrix $\mathbf{G}$ is augmented by δL extra rows and δL extra columns and it will be symbolized as $\mathbf{G}^{Lj+\delta L}$.

Incremental computations regarding additional columns δL of $\mathbf{H}$ are those that construct the matrices $\mathbf{H}$ and $\mathbf{H}^{\mathrm{T}}\mathbf{H}$ as well as the vector $\mathbf{H}^{\mathrm{T}}\mathbf{y}$. If we add δL extra columns and produce $\mathbf{H}^{Lj+\delta L}$ the previous values of $\mathbf{H}^{Lj}$, $(\mathbf{H}^{Lj})^{\mathrm{T}}(\mathbf{H}^{Lj})$ and $(\mathbf{H}^{Lj})^{\mathrm{T}}\mathbf{y}$ do not change. See fig. 3.3 for illustrative examples.

Incremental with respect to δL increments are the computations involved in the matrices $\mathbf{H}$, $\mathbf{H}^{\mathrm{T}}\mathbf{H}$, $(\mathbf{H}^{\mathrm{T}}\mathbf{H}+\mathbf{I}/C)$, the vector $\mathbf{H}^{\mathrm{T}}\mathbf{y}$, the Cholesky decomposition of the gram matrix $(\mathbf{H}^{\mathrm{T}}\mathbf{H}+\mathbf{I}/C)=\mathbf{R}^{\mathrm{T}}\mathbf{R}$, and the QR decomposition $\mathbf{H}=\mathbf{Q}\mathbf{R}$.

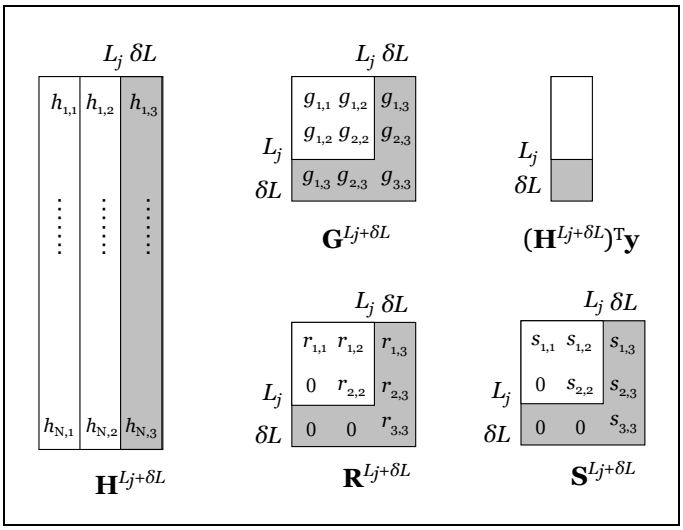


Figure 3.3. *Incremental computations* regarding the addition of new columns δL, indicated in grey scale, to the ELM's hidden layer mapping matrix $\mathbf{H}$ (or regression matrix) that is augmented by δL. The regularized gram matrix $\mathbf{G}^{Lj+\delta L} \equiv [(\mathbf{H}^{Lj+\delta L})^{\mathrm{T}}(\mathbf{H}^{Lj+\delta L})]+\mathbf{I}/C$, its upper triangular Cholesky factor $\mathbf{R}^{Lj+\delta L}$, as well as the vector $(\mathbf{H}^{Lj+\delta L})^{\mathrm{T}}\mathbf{y}$, and the inverse of the upper triangular Cholesky factor indicated as $\mathbf{S}^{Lj+\delta L} \equiv (\mathbf{R}^{Lj+\delta L})^{-1}$, all retain their previous values.

3.2.6 Computational Series

When testing L_j hidden neurons from δL to $L_{\max}$ with step δL the computation steps that are composed of consecutive δL increments can be summed up in series. The well known sum of first n is the series $1+2+3+\cdots+ n = n(n+1)/2$. The sum of first n squares is $1+4+9+16+\cdots+n^2$ which is $n(n+1)(2n+1)/6$. The sum of first n cubes is $1+8+27+64+\cdots+n^3$ is the square of the sum of first n which equals $n^2(n+1)^2/4$.

In the same way, the sum of first L_j, incrementing by δL until $L_{\max}$ is reached, is given by the series $(\delta L) + 2\,\delta L + 3\,\delta L + \ldots + (L_{\max}/\delta L)\delta L = \delta L(1+2+\ldots+L_{\max}/\delta L) = \delta L(L_{\max}/\delta L)(L_{\max}/\delta L+1)/2 = L_{\max}(L_{\max}+\delta L)/(2\delta L) \approx L_{\max}^2 / (2\delta L)$.

Similarly the sum of first L_j squares is $(\delta L)^2 + 2^2(\delta L)^2 + 3^2(\delta L)^2 + \ldots + (L_{\max}/\delta L)^2(\delta L)^2 = (\delta L)^2\ (1+4+9+\cdots+(L_{\max}/\delta L)^2) = (\delta L)^2(L_{\max}/\delta L)((L_{\max}/\delta L)+1)(2(L_{\max}/\delta L)+1)/6 = (L_{\max})(L_{\max}+\delta L)(L_{\max}+\delta L/2) / (3\delta L) \approx L_{\max}^3 / (3\delta L)$.

The expressions for these series will be required for calculating the computational costs.

3.2.7 Important Remarks on economizing computations

For testing several different values of L_j hidden neurons we usually randomly generate $L_{\max}$ neurons and validate all the intermediate ELM models they produce. Then the cross-validations proceed for $L_{\max}/\delta L$ incrementally increased different values of L_j hidden neurons.

Caching H: For efficiently economized computations in all the cross-validation methods (k-fold, leave-one-out etc.) presented in the next sections it is suggested to store the hidden layer output matrix **H**. This is essential for faster finding both the output weights as well as the error residuals. That is because all the methods need to calculate the matrix $\mathbf{H}^T\mathbf{H}$ as well as the residuals $e_n = y_n - f(\mathbf{x}_n)$ in order to compute the cross-validation errors and we must avoid the cost of re-computing **H** each time. Calculating the hidden layer output matrix **H** requires LNd operations and storing it is imperative for economising computations. That is why we repeatedly mention this step as the first most important one.

Caching $\mathbf{H}^T\mathbf{H}$ and subtracting $\mathbf{H}_{-f}^T\mathbf{H}_{-f}$ per f fold: In the 10-fold cross-validation it is suggested to first produce and store the matrix $\mathbf{H}^T\mathbf{H}$ and afterwards for each f fold to remove from it by subtraction the influence of the f fold, that is the matrix $\mathbf{H}_f^T\mathbf{H}_f$. In this way we can create matrix $\mathbf{H}_{-f}^T\mathbf{H}_{-f}$ without the f fold. Then the cost of 10-fold CV that needs 10 $\mathbf{H}_{-f}^T\mathbf{H}_{-f}$ matrices reduces from $10L^2N$ to $2L^2N + 2kL^2$. This simple strategy results in a major cost reduction within the efficient and scalable 10-fold CV algorithms which are presented next.

Compute only the upper-triangle of symmetric matrices: The square Gramian matrices such as $\mathbf{H}^T\mathbf{H}$ and $\mathbf{H}_{-f}^T\mathbf{H}_{-f}$ are symmetric. Thus we can compute only their upper-triangle and then copy the upper-triangle into the lower-triangle. This simple trick saves half of the time for computing $\mathbf{H}^T\mathbf{H}$ and reduces this cost from L^2N to $L^2N/2$.

Ordering the 10-fold CV loops: There are 3 loops one inside another for the 10-fold cross-validation. One loop iterates through the C_i values, another iterates through the L_j values and another loop iterates through the 10 folds. Depending on the order we apply the 10-fold CV loop (first, second or third) there are the three versions of 10-fold

CV. Examples of these versions for testing 30 values of C are presented in the following:

1) Naive 10-fold cross-validation (10-fold CV loop is third):

For each L_j value range from 0 to L_{max} with step δL:

For each one C_i of the 30 values of C parameter:

For each one of the f=1 to 10 folds:

find the ELM weights and the error residuals

2) Efficient 10-fold cross-validation (10-fold CV loop is second):

For each L_j value range from 0 to L_{max} with step δL:

For each one of the f=1 to 10 folds:

For each one C_i of the 30 values of C parameter:

find the ELM weights and the error residuals

3) Scalable 10-fold cross-validation (10-fold CV loop is first):

For each one of the f=1 to 10 folds:

For each L_j value range from 0 to L_{max} with step δL:

For each one C_i of the 30 values of C parameter:

find the ELM weights and the error residuals

The second as well as the third 10-fold CV version can directly include the subtraction of $\mathbf{H}_{-f}^T\mathbf{H}_{-f}$ per fold from the cached matrix $\mathbf{H}^T\mathbf{H}$, which was mentioned earlier, and hence can economize a lot of computations, whereas the third version is more scalable than the others.

In the following sections we are going to reveal that some ELM model selection methods are slow (10-fold CV and SVD, HAT and Cholesky), others are fast (HAT and SVD) and others are fast & scalable (Efficient 10-fold CV and Cholesky, HAT with QR and SVD, HAT and EVD, Scalable 10-fold CV and EVD).

3.3 Cholesky based

Cholesky decomposition performs matrix factorization of the regularized matrix $\mathbf{H}^T\mathbf{H}+\mathbf{I}/C=\mathbf{R}^T\mathbf{R}$, given in terms of an upper triangular matrix $\mathbf{R}$. Then the inverse is $(\mathbf{H}^T\mathbf{H}+\mathbf{I}/C)^{-1}=\mathbf{R}^{-1}(\mathbf{R}^{-1})^T=\mathbf{S}\mathbf{S}^T$ where $\mathbf{S} = \mathbf{R}^{-1}$ represents the inverse of the upper triangular (in practise we first compute the inverse of the lower triangular $\mathbf{R}^T$ which is simpler to compute and then we transpose it). The Cholesky decomposition requires $L^3/3$ operations [26].

3.3.1 Efficient 10-fold cross-validation and Cholesky

An efficient version of 10-fold cross-validation calculates and store once the entire matrix $\mathbf{H}^T\mathbf{H}$ as well as entire vector $\mathbf{H}^T\mathbf{y}$ before the splitting into folds. Then for each f fold it removes the influence of the f fold from matrix $\mathbf{H}^T\mathbf{H}$ in order to produce the corresponding matrix $\mathbf{H}_{-f}^T\mathbf{H}_{-f}$ for this fold. The following algorithm 1 illustrates the detailed computational steps of the efficient 10-fold cross-validation by employing Cholesky decomposition.

The strategy in algorithm 1 is faster than the standard 10-fold cross-validation in algorithm 0. If we first store the matrix $\mathbf{H}_f^T\mathbf{H}_f$ and then remove this influence of the f fold from it by subtraction and afterwards restore this influence by addition then the

cost of 10-fold CV reduces from $10L^2N$ to $2L^2N+2kL^2$. With this simple strategy of removing-restoring influences many *k*-fold CV versions like 5-fold cross-validation and 10-fold cross validation have practically the same cost for all $\mathbf{H}^T\mathbf{H}$ computations. This cost is always $2L^2N+2kL^2$ and the most costly term $2L^2N$ does not depend on the *k* number of folds in *k*-fold CV.

Algorithm 1. *Efficient 10-fold cross-validation and Cholesky for $L_j=\delta L$ up to $L_j=L_{\max}$ random hidden neurons with step δL and 30 C_i regularization parameters*	*Operations*	*total*
1. Generate and store the hidden layer matrix $\mathbf{H}^{L\max}$	---	---
Split $\mathbf{H}^{L\max}$ into 10 folds as $[\mathbf{H_1}^{L\max} \dots \mathbf{H_f}^{L\max} \dots \mathbf{H_{10}}^{L\max}]^T$	---	---
Split $\mathbf{y}$ into 10 folds as $\mathbf{y} = [\mathbf{y_1} \dots \mathbf{y_f} \dots \mathbf{y_{10}}]^T$	---	---
2. Store the square matrix $[(\mathbf{H}^{L\max})^T\mathbf{H}^{L\max}]$	$L_{\max}^2 N$	$L_{\max}^2 N$
3. Store the vector $(\mathbf{H}^{L\max})^T \mathbf{y}$	$L_{\max} N$	$L_{\max} N$
4. For each L_j value range from 0 to $L_{\max}$ with step δL	---	---
4.1 Set $L_j = L_j + \delta L$, $\mathbf{H} = \mathbf{H}^{Lj}$, $\mathbf{H}^T\mathbf{H} = [(\mathbf{H}^{Lj})^T\mathbf{H}^{Lj}]$	---	---
4.2 For each one of the *f*=1 to 10 folds	---	---
4.2.1 Store the matrix $[(\mathbf{H_f}^{Lj})^T\mathbf{H_f}^{Lj}]$	$L_j^2 N/10$ (Not incremental)	$\boxed{L_{\max}^3 N/3\delta L}$
4.2.2 Remove the influence of the *f* fold from the matrix $[(\mathbf{H}^{Lj})^T\mathbf{H}^{Lj}]$ and produce the matrix $(\mathbf{H_{-f}}^{Lj})^T\mathbf{H_{-f}}^{Lj} = [(\mathbf{H}^{Lj})^T\mathbf{H}^{Lj}] - (\mathbf{H_f}^{Lj})^T\mathbf{H_f}^{Lj}]$	$L_j^2/10$ (Not incremental)	$L_{\max}^3/3\delta L$
4.2.3 Remove the influence of the *f* fold from the vector $(\mathbf{H}^{Lj})^T \mathbf{y}$ in order to produce the vector $[(\mathbf{H_{-f}}^{Lj})^T \mathbf{y_{-f}}] = (\mathbf{H}^{Lj})^T \mathbf{y} - [(\mathbf{H_f}^{Lj})^T\mathbf{y_f}]$	$L_j N/10$ (Not incremental)	$L_{\max}^2 N/2\delta L$
4.2.4 For each one C_i of the 30 values of *C* parameter	---	---
4.2.4.1 Find Cholesky $(\mathbf{H_{-f}}^T\mathbf{H_{-f}} + \mathbf{I}/C_i) = \mathbf{R}^T\mathbf{R}$	$L_j^3/3$ (Depend on *C*)	$100L_{\max}^4/4\delta L$
4.2.4.2 Solve $\boldsymbol{\beta} = (\mathbf{H_{-f}}^T\mathbf{H_{-f}} + \mathbf{I}/C_i)^{-1}\,\mathbf{H_{-f}}^T\,\mathbf{y_{-f}}$	L_j^2 (Depend on *C*)	$100L_{\max}^3/\delta L$
4.2.4.3 Find residuals for the *f* fold $\mathbf{e_f} = \mathbf{y_f} - \mathbf{H_f}\boldsymbol{\beta}$	$L_j N/10$ (Depend on *C*)	$30L_{\max}^2 N/2\delta L$
4.2.4.4 Add to Errors $E(C_i,L_j) = E(C_i,L_j) + (\mathbf{e_f})^T\mathbf{e_f}$	---	---
5. Select the pair (C_i,L_j) with the minimum error $E(C_i,L_j)$	---	---

The overall cost of operations of algorithm 1, excluding some minor terms, is the sum of the terms in the column 'total' and is given later in table 3.1.

The not incremental terms are summed up in computational series, so where there is an L_j in the second column it becomes $L_{\max}^2/2\delta L$ in the third column. Similarly an L_j^2 in the second column gives $L_{\max}^3/3\delta L$ in the third column, and if these terms depend on *C* also then they are multiplied by 30, which is the number of regularization parameters we need to test here.

The matrices involved in the 10-fold cross validation are illustrated in fig. 3.4 for one *f* fold.

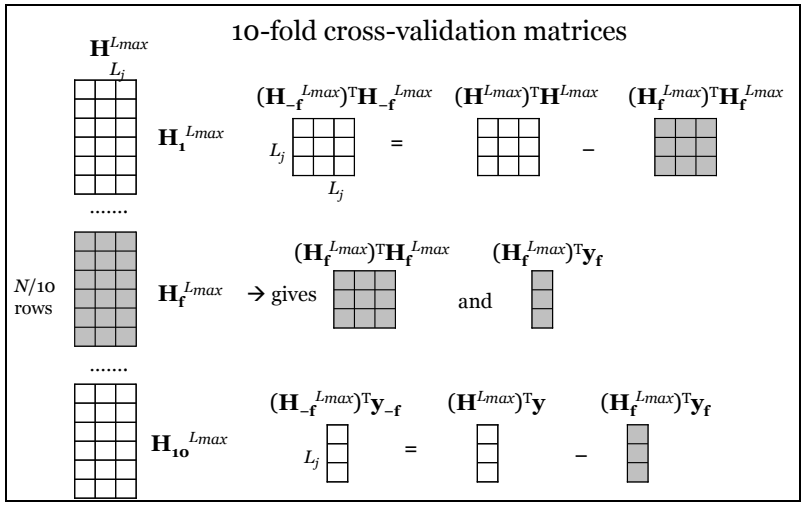


Figure 3.4. Matrix $\mathbf{H}^{Lmax}$ has N rows (correspond to training examples) and $L_{\max}$ columns (correspond to ELM hidden neurons), ranging from $L_j = L_{\min}$ to $L_j = L_{\max}$ with step δL. $\mathbf{H}^{Lmax}$ is partitioned into 10 folds as $[\mathbf{H_1}^{L\max} \ldots \mathbf{H_f}^{L\max} \ldots \mathbf{H_{10}}^{L\max}]^T$. Each f-fold $\mathbf{H_f}^{L\max}$ has $N/10$ rows. Each matrix $\mathbf{H_{-f}}^{L\max}$ is constructed without the f-fold $\mathbf{H_f}^{L\max}$. Each symmetric square matrix $(\mathbf{H_{-f}}^{L\max})^T\mathbf{H_{-f}}^{L\max}$ has $L_{\max}$ rows and $L_{\max}$ columns and it is created without the contribution of the f-fold $[(\mathbf{H_f}^{L\max})^T\mathbf{H_f}^{L\max}]$. The vectors $(\mathbf{H_{-f}}^{L\max})^T\mathbf{y_{-f}}$ are also illustrated.

3.3.2 Leave-one-out cross-validation with Cholesky Decomposition

For the values of the leave-one-out error $e_{-n}(\mathbf{x}_n) = (y_n - \hat{y}_n)/(1 - hat_{nn})$ we need the diagonal hat_{nn} elements of the HAT matrix (see eq. 3.7). Cholesky decomposition is incremental, regarding additional rows and columns (see fig. 3.3). Thus, if we compute once the upper triangular $\mathbf{R}^{L\max}$ for the matrix $\mathbf{G}^{L\max} \equiv [(\mathbf{H}^{L\max})^T(\mathbf{H}^{L\max})] + \mathbf{I}/C$, then any smaller than $L_{\max}$ sub-matrix $\mathbf{G}^{Lj}$ (with size $L_j \times L_j$), that starts from the first row and column of $\mathbf{G}^{L\max}$, has decomposition $\mathbf{R}^{Lj}$ which is also known.

The computation of the inverse matrix $(\mathbf{G}^{Lj+1})^{-1} = (\mathbf{S}^{Lj+1})(\mathbf{S}^{Lj+1})^T$ in terms of $\mathbf{S}^{Lj+1}$, that denotes the inverse of the upper triangular Cholesky factor $(\mathbf{R}^T)^{-1}$ is illustrated in fig. 3.5. The computation steps is a series of rank one updates based on the previous inverse $(\mathbf{G}^{Lj})^{-1}$ values. This cost is L_j^2 for each one of the iteration. For δL to $L_{\max}$ iterations (random neurons) the total cost for iteratively inverting the matrix $\mathbf{G}$ is the series known as the sum of first L_j squares $1+4+9+16+\cdots+L_{\max}^2 \approx O(L_{\max}^3/3)$.

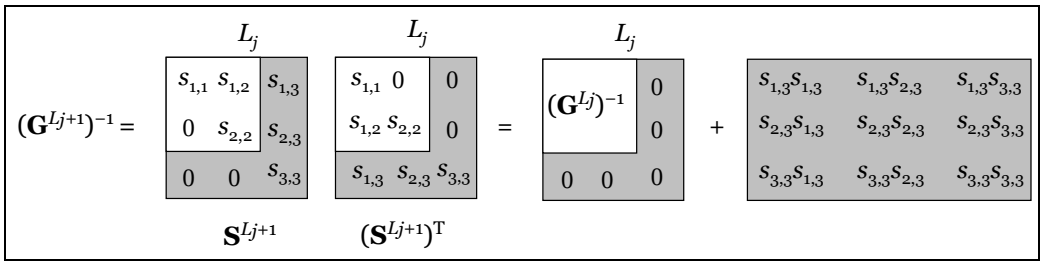


Figure 3.5. Rank one updates for the inverse of $\mathbf{G}$ regarding the addition of new columns δL, that are indicated in grey scale, to $\mathbf{H}$ which is augmented by 1 column. The corresponding inverse of the matrix $(\mathbf{G}^{Lj+1})^{-1} = (\mathbf{S}^{Lj+1})(\mathbf{S}^{Lj+1})^T$ is given in terms of $\mathbf{S}^{Lj+1}$ that denotes the inverse of the upper triangular Cholesky factor.

Then, the leave-one-out cross-validation with HAT and Cholesky decomposition is given in algorithm 2.

Algorithm 2. *Leave-one-out cross-validation with HAT and Cholesky for $L_j=\delta L$ up to $L_j=L_{\max}$ random hidden neurons with step δL and 30 C_i regularization parameters*	*Operations*	*total*
1. Generate and store the hidden layer matrix $\mathbf{H}^{L\max}$	---	---
2. Store the matrix $[(\mathbf{H}^{L\max})^{\mathrm{T}}\mathbf{H}^{L\max}]$	$L_{\max}^2 N$	$L_{\max}^2 N$
3. Store the vector $(\mathbf{H}^{L\max})^{\mathrm{T}}\,\mathbf{y}$	$L_{\max} N$	$L_{\max} N$
4. For each one C_i of the 30 values of C parameter	---	---
4.1 Find $\mathbf{G}^{L\max} \equiv [(\mathbf{H}^{L\max})^{\mathrm{T}}\mathbf{H}^{L\max}]+\mathbf{I}/C_i$	$L_{\max}$ } Depend on C	$30L_{\max}$
4.2 Find Cholesky of $\mathbf{G}^{L\max} = (\mathbf{R}^{L\max})^{\mathrm{T}}(\mathbf{R}^{L\max})$	$L_{\max}^3/3$ } Depend on C	$30L_{\max}^3/3$
4.3 store the inverse $\mathbf{S}^{L\max} = (\mathbf{R}^{L\max})^{-1}$	$L_{\max}^2$ } Depend on C	$30L_{\max}^2$
4.6 For each L_j value range from 0 to $L_{\max}$ with step δL	---	---
4.6.1 Set $L_j = L_j + \delta L$, $\mathbf{H}^{Lj}$, $\mathbf{S}^{Lj} = (\mathbf{R}^{Lj})^{-1}$	---	---
4.6.2 Find the inverse $(\mathbf{G}^{Lj})^{-1} = \mathbf{S}^{Lj}\,(\mathbf{S}^{Lj})^{\mathrm{T}}$	L_j^2 } Not incremental & Depend on C	$30L_{\max}^3/3$
4.6.3 Calculate weights $\boldsymbol{\beta}^{Lj} = (\mathbf{G}^{Lj})^{-1}\,[(\mathbf{H}^{Lj})^{\mathrm{T}}\mathbf{y}]$	L_j^2 } Not incremental & Depend on C	$30L_{\max}^2/3\delta L$
4.6.4 find the residuals $\mathbf{e} = \mathbf{y} - \mathbf{H}^{Lj}\,\boldsymbol{\beta}^{Lj}$	$L_j N$ } Not incremental & Depend on C	$30L_{\max}^2 N/3\delta L$
4.6.5 Calculate all $hat_{nn} = \mathbf{h}(\mathbf{x}_n)\,(\mathbf{G}^{Lj})^{-1}\,\mathbf{h}^{\mathrm{T}}(\mathbf{x}_n)$	$L_j^2 N$ } Not incremental & Depend on C	$\boxed{30L_{\max}^3 N/3\delta L}$
4.6.6 Set error $E(C_i,L_j) = (1/N)\sum_n^N (e_n)^2/(1-hat_{nn})^2$	---	---
5. Select the pair (C_i,L_j) with the minimum error $E(C_i,L_j)$	---	---

The overall cost of operations of algorithm 2, excluding some minor terms, is the sum of the terms in the column 'total' and is given in table 3.1.

If the inverse $(\mathbf{G}^{Lj})^{-1}$ is known and given matrix $\mathbf{H}^{Lj}$ then the computations of one diagonal element of the HAT matrix is $hat_{nn} = \mathbf{h}(\mathbf{x}_n)\,(\mathbf{G}^{Lj})^{-1}\,\mathbf{h}^{\mathrm{T}}(\mathbf{x}_n)$ and has cost $\mathrm{O}(L_j^2)$. For N such elements it becomes $L_j^2 N$. For δL to $L_{\max}$ neuron increments by step δL this computation is a series of consecutive steps based on the repeatedly augmented matrix $(\mathbf{G}^{Lj+\delta L})^{-1}$ whose values change in every step. Hence the total cost gives the series known as the sum of first L_j squares $(\delta L)^2 N + 2^2(\delta L)^2 N + \ldots + (L_{\max}/\delta L)^2(\delta L)^2 N$ which in section 2.5 we have seen that it is equal to $\mathrm{O}(L_{\max}^3 N/(3\delta L))$.

Given $(\mathbf{G}^{Lj})^{-1}$ the computation of the weights vector $\boldsymbol{\beta}^{Lj} = (\mathbf{G}^{Lj})^{-1}\,[(\mathbf{H}^{Lj})^{\mathrm{T}}\mathbf{y}]$ is a simple matrix-vector multiplication of a matrix of size $L_j \times L_j$ with a vector of length L_j which has cost L_j^2. For δL to $L_{\max}$ neuron increments by step δL the total cost is again given by the sequence $(\delta L)^2(1+4+9+\cdots+(L_{\max}/\delta L)^2)$ which is the well-known sum of first $L_j = (L_{\max}/\delta L)$ squares equal to $L_{\max}^3/(3\delta L)$.

Given vector $\boldsymbol{\beta}^{Lj}$ and matrix $\mathbf{H}^{Lj}$ the calculation of the residuals $\mathbf{e} = \mathbf{y} - \mathbf{H}^{Lj}\,\boldsymbol{\beta}^{Lj}$ has cost $L_j N$. By adding random neurons from δL to $L_{\max}$ the total cost becomes the series $(\delta L)(1+2+3+\cdots+(L_{\max}/\delta L)) \approx L_{\max}^2 N/(2\delta L)$.

3.4 SVD based

In the following Singular Value Decomposition (SVD) is performed on the hidden layer mapping matrix $\mathbf{H}=\mathbf{U}\mathbf{\Sigma}\mathbf{V}^T$ that produces three matrices, namely $\mathbf{U}^{N\times L}$ which is the orthonormal matrix containing the left singular vectors of $\mathbf{H}$, with $\mathbf{U}^T\mathbf{U}=\mathbf{I}$ but $\mathbf{U}\mathbf{U}^T\neq\mathbf{I}$, $\mathbf{\Sigma}^{L\times L}$ which is the diagonal matrix containing the singular values of $\mathbf{H}$, and $\mathbf{V}^{L\times L}$ which is the orthonormal matrix containing the right singular vectors, for which it holds $\mathbf{V}^T\mathbf{V}=\mathbf{I}=\mathbf{V}\mathbf{V}^T$. For a single SVD of $\mathbf{H}$ a minimum cost of operations is $4L^2N+10L^3$ [26] .

Singular Value Decomposition of the hidden layer output matrix $\mathbf{H}$ allows factoring in terms of the reusable matrices $\mathbf{U}$, $\mathbf{\Sigma}$ and $\mathbf{V}$, by the following well known mathematical transformations:

$$\mathbf{H}^T\mathbf{H} = (\mathbf{U}\mathbf{\Sigma}\mathbf{V}^T)^T\,\mathbf{U}\mathbf{\Sigma}\mathbf{V}^T = \mathbf{V}\mathbf{\Sigma}^T\mathbf{U}^T\mathbf{U}\mathbf{\Sigma}\mathbf{V}^T = \mathbf{V}(\mathbf{\Sigma}^T\mathbf{\Sigma})\mathbf{V}^T \tag{3.9a}$$

$$\mathbf{H}\mathbf{H}^T = \mathbf{U}\mathbf{\Sigma}\mathbf{V}^T(\mathbf{U}\mathbf{\Sigma}\mathbf{V}^T)^T = \mathbf{U}\mathbf{\Sigma}\mathbf{V}^T\,\mathbf{V}\mathbf{\Sigma}\mathbf{U}^T = \mathbf{U}(\mathbf{\Sigma}\mathbf{\Sigma}^T)\mathbf{U}^T \tag{3.9b}$$

The pseudo-inverse is also given by:

$$\mathbf{H}^{\dagger} = (\mathbf{H}^T\mathbf{H})^{-1}\mathbf{H}^T = \mathbf{V}(\mathbf{\Sigma}^T\mathbf{\Sigma})^{-1}\,\mathbf{V}^T\,\mathbf{V}\mathbf{\Sigma}^T\mathbf{U}^T = \mathbf{V}(\mathbf{\Sigma}^T\mathbf{\Sigma})^{-1}\mathbf{\Sigma}^T\mathbf{U}^T = \mathbf{V}(\mathbf{\Sigma})^{-1}\mathbf{U}^T \tag{3.10}$$

Note that a square and column-orthonormal matrix $\mathbf{V}^T$ is also row-orthonormal $\mathbf{V}\mathbf{V}^T=\mathbf{I}$. If $\mathbf{V}=[\mathbf{v}_i\ \mathbf{v}_2\ \ldots\ \mathbf{v}_L]$ then for the orthogonal columns $\mathbf{v}_i$ it holds that $\mathbf{v}_i^T\mathbf{v}_j\neq 0$ for i=j and $\mathbf{v}_i^T\mathbf{v}_j=0$ for i≠j, where the $norm\,|\,\mathbf{v}_i\,| = \mathbf{v}_i^T\mathbf{v}_i = const > 0$. For orthonormal columns the normalization of each $\mathbf{v}_i$ is also performed and $\mathbf{v}_i^T\mathbf{v}_i=1$.

Hence, it is easy to find the output weights $\mathbf{\beta}$ by using the reusable matrices $\mathbf{U}$, $\mathbf{\Sigma}$ and $\mathbf{V}$. Since $\mathbf{V}\mathbf{V}^T = \mathbf{I}$ the inexpensive validation of several values of C is performed via:

$$\mathbf{H}^T\mathbf{H}+\mathbf{I}/C = \mathbf{V}(\mathbf{\Sigma}^T\mathbf{\Sigma})\mathbf{V}^T + \mathbf{I}/C = \mathbf{V}(\mathbf{\Sigma}^T\mathbf{\Sigma})\mathbf{V}^T + \mathbf{V}\mathbf{V}^T/C = \mathbf{V}(\mathbf{\Sigma}^T\mathbf{\Sigma}+\mathbf{I}/C)\mathbf{V}^T \tag{3.11}$$

Then the ELM output weights are given by:

$$\mathbf{\beta} = \mathbf{V}(\mathbf{\Sigma}^T\mathbf{\Sigma}+\mathbf{I}/C)^{-1}\,\mathbf{V}^T\,\mathbf{H}^T\mathbf{y} = \mathbf{V}(\mathbf{\Sigma}^T\mathbf{\Sigma}+\mathbf{I}/C)^{-1}\,\mathbf{V}^T\,\mathbf{V}\,\mathbf{\Sigma}^T\mathbf{U}^T\mathbf{y} = \mathbf{V}(\mathbf{\Sigma}^T\mathbf{\Sigma}+\mathbf{I}/C)^{-1}\,\mathbf{\Sigma}^T\mathbf{U}^T\mathbf{y} \tag{12}$$

This is a well known solution in which the orthonormal matrices $\mathbf{V}$ and $\mathbf{U}$ are reusable many times during testing for several values of the regularization parameter C.

At this point we can also clearly see why and how the Tikhonov's regularization works because it cleverly modifies the singular values as:

$$\mathbf{\beta} = \mathbf{V}(\mathbf{\Sigma}^T\mathbf{\Sigma}+\mathbf{I}/C)^{-1}\,\mathbf{\Sigma}^T\mathbf{U}^T\mathbf{y} = \mathbf{V}\,diag\left(\frac{\sigma_1}{\sigma_1^2+1/C},\frac{\sigma_2}{\sigma_2^2+1/C},\ldots,\frac{\sigma_L}{\sigma_L^2+1/C}\right)\mathbf{U}^T\mathbf{y} \tag{3.13}$$

For comparison the pseudo-inverse solution is $\mathbf{\beta} = \mathbf{V}\,diag(1/\sigma_1,\ 1/\sigma_2,\ \ldots,\ 1/\sigma_L)\ \mathbf{U}^T\mathbf{y}$, where for stability purposes the pseudo-inverse strategy is: "for each singular value σ_i smaller than machine precision set $1/\sigma_i = 0$". This is avoided in regularization.

Thus, even if some singular values are smaller than machine precision, the Tikhonov's regularization stabilizes the results by smoothly filtering out the singular values that are small relative to $1/C$. Therefore, if some singular values σ_i are close to zero their regularized inverses $\sigma_i/((\sigma_i)^2+1/C)$ are also close to zero, in contrast to the simple pseudo-inverse solution where their inverses $1/\sigma_i$ become huge. As a result the condition number $\sigma_{\max}/\sigma_{\min}$ is well improved.

3.4.1 Model selection with k-fold cross-validation and SVD

The computational steps of 10-fold cross-validation and SVD are illustrated in algorithm 3 which examines L_j parameters from δL up to $L_{\max}$ with step δL and 30 C_i regularization parameters.

Algorithm 3. *10-fold Cross-Validation with SVD for L_j=δL up to L_j=$L_{\max}$ random hidden neurons with step δL and 30 C_i regularization parameters*	*Operations*	*total*
1. Generate and store the hidden layer matrix $\mathbf{H}^{L\max}$	---	---
Split $\mathbf{H}^{L\max}$ into 10 folds as $[\mathbf{H_1}^{L\max} \dots \mathbf{H_f}^{L\max} \dots \mathbf{H_{10}}^{L\max}]^T$	---	---
Split $\mathbf{y}$ into 10 folds as $\mathbf{y} = [\mathbf{y_1} \dots \mathbf{y_f} \dots \mathbf{y_{10}}]^T$	---	---
2. For each L_j value range from 0 to $L_{\max}$ with step δL	---	---
2.1 Set $L_j = L_j + \delta L$, $\mathbf{H} = \mathbf{H}^{Lj}$	---	---
2.2 For each one of the f=1 to 10 folds	---	---
2.2.1 Construct matrix $\mathbf{H_{-f}}$ without the f fold	$L_j N$ (Not incremental)	$10L_{\max}^2 N/2\delta L$
2.2.2 Construct vector $\mathbf{y_{-f}}$ without the f fold	L_j (Not incremental)	$10L_{\max}^2/2\delta L$
2.2.3 Perform Singular Value Decomposition on $\mathbf{H_{-f}} = \mathbf{U\Sigma V}^T$	$4L_j^2 N$ (Not incremental)	$\boxed{40L_{\max}^3 N/3\delta L}$
	$10L_j^3$ (Not incremental)	$100L_{\max}^4/4\delta L$
2.2.4 Store the vector $[\mathbf{U}^T\mathbf{y_{-f}}]$	$L_j N$ (Not incremental)	$10L_{\max}^2 N/2\delta L$
2.2.5 For each one C_i of the 30 values of C parameter	---	---
2.2.5.1 Find weights $\boldsymbol{\beta} = \mathbf{V}\,[(\mathbf{\Sigma}^T\mathbf{\Sigma} + \mathbf{I}/C_i)^{-1}\mathbf{\Sigma}^T]\,[\mathbf{U}^T\mathbf{y_{-f}}]$	L_j^2 (Depend on C)	$300L_{\max}^3/3\delta L$
2.2.5.2 Find the residuals for the f fold $\mathbf{e_f} = \mathbf{y_f} - \mathbf{H_f}\boldsymbol{\beta}$	$L_j N/10$ (Depend on C)	$30L_{\max}^2 N/2\delta L$
2.2.5.3 Add to errors $E(C_i,L_j) = E(C_i,L_j) + (\mathbf{e_f})^T\mathbf{e_f}$	---	---
3. Select the pair (C_i,L_j) with the minimum error $E(C_i,L_j)$	---	---

The overall cost of operations of algorithm 3, excluding some minor terms, is the sum of the terms in the column 'total' and is given in table 3.1. We assume that the cost for SVD is about $4L^2N+10L^3$ operations [26].

The not incremental terms are again summed up in computational series, so an L_j in the second column gives $L_{\max}^2/2\delta L$ in the third, L_j^2 gives $L_{\max}^3/3\delta L$, L_j^3 gives $L_{\max}^4/4\delta L$, and if these depend on C then they are multiplied by the 30 regularization parameters we need to test.

For deriving the cost of the weights $\boldsymbol{\beta} = \mathbf{V}\,[(\mathbf{\Sigma}^T\mathbf{\Sigma} + \mathbf{I}/C)^{-1}\mathbf{\Sigma}^T]\,[\mathbf{U}^T\mathbf{y_{-f}}]$ we note that each j^{th} element of the weights vector $\boldsymbol{\beta}$ is given in terms of the v_{jk} elements of the matrix $\mathbf{V}$ and the diagonal elements σ_{kk} of the matrix $\mathbf{\Sigma}$ by the following:

$$\beta_j = \sum_k^L v_{jk}\,\sigma_{kk}/(\sigma_{kk}^2 + 1/C)\,[\mathbf{U}^T\mathbf{y_{-f}}]_k \tag{3.14}$$

which proves that the corresponding cost for all the L elements of $\boldsymbol{\beta}$ is L^2 operations.

In the next section we will examine the application of SVD in leave-one-out cross-validation that avoids the extra cost of using 10 folds by employing the HAT matrix.

3.4.2 Model selection with HAT and SVD

The algorithm 4 shows the computational steps of leave-one-out cross-validation with the HAT matrix and SVD by testing L_j parameters from δL up to $L_{\max}$ with step δL and 30 C_i regularization parameters. The SVD based leave-one-out cross-validation has also been used effectively in kernel ridge regression methods [17] [18][20].

If $\mathbf{H}=\mathbf{U\Sigma V}^T$ then using $\mathbf{U}$ and $\mathbf{\Sigma}$ for the HAT matrix gives:

$$\begin{aligned} \text{HAT} &= \mathbf{H}\,(\mathbf{H}^T\mathbf{H} + \mathbf{I}/C)^{-1}\,\mathbf{H}^T \\ &= \mathbf{U\Sigma V}^T\,\mathbf{V}(\mathbf{\Sigma}^T\mathbf{\Sigma} + \mathbf{I}/C)^{-1}\mathbf{V}^T\,\mathbf{V\Sigma}^T\mathbf{U}^T \\ &= \mathbf{U\Sigma}(\mathbf{\Sigma}^T\mathbf{\Sigma} + \mathbf{I}/C)^{-1}\mathbf{\Sigma}^T\mathbf{U}^T \end{aligned} \tag{3.15}$$

Thus, each diagonal value is $hat_{nn} = \sum_k^L u_{nk}^2\ \sigma_{kk}^2\ /(\sigma_{kk}^2 + 1/C)$ where u_{nk} is an element of the matrix $\mathbf{U}$, and σ_{kk} is a diagonal element of the matrix $\mathbf{\Sigma}$.

Algorithm 4. *Leave-one-out cross-validation with HAT and SVD for L_j=δL up to L_j=$L_{\max}$ random hidden neurons with step δL and 30 C_i regularization parameters*	*Operations*	*total*
1. Generate and store the hidden layer matrix $\mathbf{H}^{L\max}$	---	---
2. For each L_j value range from 0 to $L_{\max}$ with step δL	---	---
2.1 Set $L_j = L_j + \delta L$, $\mathbf{H} = \mathbf{H}^{Lj}$	---	---
2.2 Perform Singular Value Decomposition on $\mathbf{H} = \mathbf{U\Sigma V}^T$	$4L_j^2N$ $10L_j^3$ } Not incremental	$4L_{\max}^3 N/3\delta L$ $10L_{\max}^4/4\delta L$
2.3 Store the vector $[\mathbf{U}^T\ \mathbf{y}]$	L_jN	$L_{\max}^2 N/2\delta L$
2.4 For each one C_i of the 30 values of C parameter	---	---
2.5.1 Find weights $\mathbf{\beta} = \mathbf{V}\,[(\mathbf{\Sigma}^T\mathbf{\Sigma} + \mathbf{I}/C_i)^{-1}\mathbf{\Sigma}^T]\,[\mathbf{U}^T\mathbf{y}]$	L_j^2 } Depend on C	$30L_{\max}^3/3\delta L$
2.5.2 Find the residuals $\mathbf{e} = \mathbf{y} - \mathbf{H\beta}$	L_jN	$30L_{\max}^2 N/2\delta L$
2.5.3 Find all $hat_{nn} = \sum_k^{L_j} u_{nk}^2\ \sigma_{kk}^2\ /(\sigma_{kk}^2 + 1/C_i)$	L_jN	$30L_{\max}^2 N/2\delta L$
2.5.4 Set $E(C_i,L_j) = (1/N)\sum_n^N (e_n)^2/(1-hat_{nn})^2$	---	---
3. Select the pair (C_i,L_j) with the minimum error $E(C_i,L_j)$	---	---

The overall cost of operations of algorithm 4, excluding some minor terms, is the sum of the terms in the column 'total' and is given in table 3.1. The leave-one-out cross-validation with HAT and SVD has a clear advantage over 10-fold cross-validation and SVD. However the step which performs SVD on $\mathbf{H}$ is slow. The important issue here is the most costly term $4L_{\max}^3 N/3\delta L$. For larger datasets one can work with more scalable decompositions. This algorithm is much helpful for introducing SVD factorings that are also used in the next section which presents QR and SVD.

3.5 QR and SVD based

3.5.1 Leave-one-out cross-validation with QR and SVD

The first choice when we can thing for incremental computations is QR decomposition. QR can decompose the rectangular hidden layer mapping matrix $\mathbf{H}^{N\times L}$ into an orthogonal and unitary matrix $\mathbf{Q}^{N\times L}$ with orthonormal columns, and an upper (or right) triangular matrix $\mathbf{R}^{L\times L}$, which has zeros below the diagonal. QR is incremental since any previous orthonormal column of $\mathbf{Q}$ remains the same when calculating the next.

Like SVD the QR decomposition can be applied even if the matrix is singular. Different from SVD the QR decomposition of **H** is easily parallelizable, as well as scalable [33] if the matrix **H** is distributed in several machines. For performing **H**=**QR** safely by using Householder transformations the cost of operations is $2L^2N$ [26]. In practice, the matrix **Q** is a virtual matrix represented as a product of L Householder reflections and is not usually evaluated explicitly.

Recall at this point that a fast cross-validation that searches for the best regularization parameter C needs a fast computation of the $E_{loo}(C) = (1/N)\sum_n^N e_{-n}^2(\mathbf{x}_n)$. For the values of the leave-one-out error $e_{-n}(\mathbf{x}_n) = (y_n - \hat{y}_n)/(1-hat_{nn})$ in matrix form (see also eq. 3.8) we get the vector $\mathbf{e}_{-n}(\mathbf{X})$ = ((**I** − HAT) **y**)/diag(**I** − HAT) where the numerator and denominator of $\mathbf{e}_{-i}(\mathbf{X})$ are both vectors of size N. Thus the only thing to do is to decompose the matrix **I** − HAT = **I** − **H** $(\mathbf{H}^T\mathbf{H}+\mathbf{I}/C)^{-1}$ $\mathbf{H}^T$, in terms of reusable matrices.

With QR decomposition **H** = **QR**, the hidden layer mapping matrix **H** is expressed in terms of the orthogonal matrix **Q** and the right triangular matrix **R**. In this case it gives $\mathbf{H}^T\mathbf{H}$ = $\mathbf{R}^T\mathbf{Q}^T\mathbf{Q}\mathbf{R}$ = $\mathbf{R}^T\mathbf{R}$, since $\mathbf{Q}^T\mathbf{Q}$ = **I**, and the classical regularized least squares estimate becomes $\boldsymbol{\beta} = (\mathbf{H}^T\mathbf{H}+\mathbf{I}/C)^{-1}\mathbf{H}^T\mathbf{y} = (\mathbf{R}^T\mathbf{R}+\mathbf{I}/C)^{-1}\mathbf{R}^T\mathbf{Q}^T\mathbf{y}$.

At this point we can continue further by decomposing the right triangular matrix **R** via Singular Value Decomposition **R** = $\mathbf{U}\boldsymbol{\Sigma}\mathbf{V}^T$, where $\mathbf{U}^{L\times L}$ is the orthonormal matrix containing the left singular vectors of **R**, $\boldsymbol{\Sigma}^{L\times L}$ is the diagonal matrix containing the singular values of **R**, and $\mathbf{V}^{L\times L}$ is the orthonormal matrix containing the right singular vectors of **R**. In this way we produce:

$$\mathbf{H} = \mathbf{Q}\mathbf{U}\boldsymbol{\Sigma}\mathbf{V}^T \tag{3.17a}$$

$$\mathbf{H}^T = \mathbf{V}\boldsymbol{\Sigma}^T\mathbf{U}^T\mathbf{Q}^T \tag{3.17b}$$

Obviously $\boldsymbol{\Sigma}=\boldsymbol{\Sigma}^T$, meaning that the diagonal matrix and its transpose is the same but we keep the different symbolism for clarity, in order to follow the transpositions.

Then $\mathbf{H}^T\mathbf{H} = \mathbf{V}\boldsymbol{\Sigma}^T\mathbf{U}^T\mathbf{Q}^T\mathbf{Q}\mathbf{U}\boldsymbol{\Sigma}\mathbf{V}^T$ and given that $\mathbf{Q}^T\mathbf{Q}=\mathbf{I}$ and $\mathbf{U}^T\mathbf{U}=\mathbf{I}$ we get $\mathbf{H}^T\mathbf{H} = \mathbf{V}\boldsymbol{\Sigma}^T\boldsymbol{\Sigma}\mathbf{V}^T$ and from that $\mathbf{H}^T\mathbf{H}+\mathbf{I}/C = (\mathbf{V}\boldsymbol{\Sigma}^T\boldsymbol{\Sigma}\mathbf{V}^T+\mathbf{I}/C) = \mathbf{V}(\boldsymbol{\Sigma}^T\boldsymbol{\Sigma}+\mathbf{I}/C)\mathbf{V}^T$. The inverse of this result is given by:

$$(\mathbf{H}^T\mathbf{H}+\mathbf{I}/C)^{-1} = \mathbf{V}(\boldsymbol{\Sigma}^T\boldsymbol{\Sigma}+\mathbf{I}/C)^{-1}\mathbf{V}^T \tag{3.18}$$

Substituting eq. 3.17 and eq. 3.18 into the HAT matrix we get:

$$\text{HAT} = \mathbf{H}\,(\mathbf{H}^T\mathbf{H}+\mathbf{I}/C)^{-1}\,\mathbf{H}^T = \mathbf{H}\,\mathbf{V}(\boldsymbol{\Sigma}^T\boldsymbol{\Sigma}+\mathbf{I}/C)^{-1}\mathbf{V}^T\,\mathbf{H}^T \tag{3.19}$$

From eq. 3.19 we can compute diagonal elements of [**I** − HAT] by using the reusable matrices **H**, **V** and **Σ**. The matrix $(\boldsymbol{\Sigma}^T\boldsymbol{\Sigma}+\mathbf{I}/C)^{-1}$ is a diagonal matrix that can be easily inverted with L operations. Thus we can find the N diagonal values of [**I** − HAT] in LN operations by using the matrix [**HV**] as follows:

$$[\mathbf{I} - \text{HAT}]_{nn} = 1 - \sum_k^L [\mathbf{HV}]_{nk}^2/(\sigma_{kk}^2 + 1/C) \tag{3.20}$$

where $[\mathbf{H}\,\mathbf{V}]_{nk}$ denotes the hv_{nk} element of the matrix [**HV**], and σ_{kk} is the k^{th} diagonal element of the matrix **Σ** that contains the singular values. The matrix [**H V**] is a product of matrix-matrix multiplication with cost of operations L^2N. Once [**H V**] is computed then we can test several different values of C with a small cost.

The numerator of the leave-one-out error in matrix form $\mathbf{e}_{-n}(\mathbf{X})$ is the residual vector:

$$\mathbf{e} = (\mathbf{I} - \text{HAT})\,\mathbf{y} = \mathbf{y} - \mathbf{H}\,\mathbf{V}\,(\boldsymbol{\Sigma}^T\boldsymbol{\Sigma}+\mathbf{I}/C)^{-1}\mathbf{V}^T\mathbf{H}^T\,\mathbf{y} \tag{3.21}$$

Algorithm 5 illustrates the leave-one-out cross-validation via the HAT matrix by employing QR and SVD. It searches for the best pair of $\{C_i, L_j\}$ by using different values L_j of hidden neurons incrementally increased from δL to $L_{\max}$ with step δL and 30 regularization parameters C_i.

Algorithm 5. *Leave-one-out cross-validation via the HAT matrix by using QR and SVD for $L_j=\delta L$ up to $L_j=L_{\max}$ hidden neurons with step δL and 30 C_i parameters*	*Operations*	*total*
1. Generate and store the hidden layer matrix $\mathbf{H}^{L\max}$	---	---
2. Compute once the QR decomposition $\mathbf{H}^{L\max} = \mathbf{Q}^{L\max}\mathbf{R}^{L\max}$	$2L_{\max}^2 N$	$2L_{\max}^2 N$
3. For each L_j value range from 0 to $L_{\max}$ with step δL	---	---
3.1 Set $L_j = L_j + \delta L$, $\mathbf{H} = \mathbf{H}^{Lj}$, $\mathbf{R} = \mathbf{R}^{Lj}$	---	---
3.2 Perform SVD on $\mathbf{R} = \mathbf{U\Sigma V}^{\mathrm{T}}$	$10L_j^3$	$10L_{\max}^4 / 4\delta L$
3.3 Store the matrix $[\mathbf{HV}]$	$L_j^2 N$ Not incremental	$\boxed{L_{\max}^3 N / 3\delta L}$
3.4 Calculate and store the vector $[\mathbf{V}^{\mathrm{T}}\mathbf{H}^{\mathrm{T}}\mathbf{y}]$	$L_j N$	$L_{\max}^2 N / 2\delta L$
3.5 For each one C_i of the 30 values of C parameter	---	---
3.5.1 Find $\mathbf{e} = \mathbf{y} - \mathbf{HV}\,(\mathbf{\Sigma\Sigma}^{\mathrm{T}}+\mathbf{I}/C_i)^{-1}\,[\mathbf{V}^{\mathrm{T}}\mathbf{H}^{\mathrm{T}}\mathbf{y}]$	$L_j N$	$30L_{\max}^2 N / 2\delta L$
3.5.2 Calculate the output weights $\boldsymbol{\beta} = \mathbf{H}^{\mathrm{T}}\, C_i\, \mathbf{e}$	$L_j N$ Depend on C	$30L_{\max}^2 N / 2\delta L$
3.5.3 Find $(1-hat_{nn}) = 1 - \sum_k^L [\mathbf{HV}]_{nk}^2 / (\sigma_{kk}^2 + 1/C_i)$	$L_j N$	$30L_{\max}^2 N / 2\delta L$
3.5.4 Set $E(C_i,L_j) = (1/N) \sum_n^N (e_n)^2/(1-hat_{nn})^2$	---	---
4. Select the pair (C_i,L_j) with the minimum error $E(C_i,L_j)$	---	---

The overall cost of operations of algorithm 5, excluding some minor terms, is the sum of the terms in the column 'total' and is given in table 3.1, assuming the cost of SVD on the square matrix $\mathbf{R}$ is $10L^3$.

Note that each n^{th} element of the residual vector $\mathbf{e}$ is given by:

$$e_n = y_n - \sum_k^L [\mathbf{H\,V}]_{nk}\,[\mathbf{V}^{\mathrm{T}}\mathbf{H}^{\mathrm{T}}\mathbf{y}]_k\,/(\sigma_{kk}^2 + 1/C) \tag{3.22}$$

where $[\mathbf{H\,V}]_{nk}$ denotes the hv_{nk} element of the stored matrix $[\mathbf{H\,V}]$, σ_{kk} denotes the kth diagonal element of the matrix $\mathbf{\Sigma}$, and $[\mathbf{V}^{\mathrm{T}}\mathbf{H}^{\mathrm{T}}\mathbf{y}]_k$ is the kth element of this vector. This proves that for the residual vector $\mathbf{e}$ the cost is LN operations.

Also note that using this simplified $\mathbf{e}_{-n}(\mathbf{X})$ during model selection there is no need to compute directly the weights or the outputs $f(\mathbf{x}_n)$ for the training examples. If the residual vector $\mathbf{e} = (\mathbf{I} - \mathrm{HAT})\mathbf{y}$ is already known then there is an easy way to get the output weights $\boldsymbol{\beta}$ in terms of $\mathbf{e}$. Using the Sherman–Morrison–Woodbury formula [26] :

$$\mathbf{I} - \mathrm{HAT} = \mathbf{I} - \mathbf{H}(\mathbf{H}^{\mathrm{T}}\mathbf{H}+\mathbf{I}/C)^{-1}\mathbf{H}^{\mathrm{T}} = (1/C)(\mathbf{H}\mathbf{H}^{\mathrm{T}}+\mathbf{I}/C)^{-1} \tag{3.23}$$

Multiplying both sides of eq. 3.23 by $\mathbf{y}$ we get $(\mathbf{I} - \mathrm{HAT})\,\mathbf{y} = (1/C)(\mathbf{H}\mathbf{H}^{\mathrm{T}}+\mathbf{I}/C)^{-1}\,\mathbf{y}$ and thus $(\mathbf{H}\mathbf{H}^{\mathrm{T}}+\mathbf{I}/C)^{-1}\,\mathbf{y} = C\,(\mathbf{I} - \mathrm{HAT})\,\mathbf{y} = C\,\mathbf{e}$, where $\mathbf{e}$ is the numerator of $\mathbf{e}_{-n}(\mathbf{X})$. Substituting into the equation for the weights we get $\boldsymbol{\beta} = \mathbf{H}^{\mathrm{T}}(\mathbf{H}\mathbf{H}^{\mathrm{T}}+\mathbf{I}/C)^{-1}\mathbf{y} = \mathbf{H}^{\mathrm{T}}\,C\,\mathbf{e}$. In this case the weights $\boldsymbol{\beta}$ are computed after the computation of the residual vector $\mathbf{e}$.

The not incremental terms are again summed up in computational series. Although QR decomposition is incremental, the step that performs SVD on $\mathbf{R}$ is not incremental and the total cost is the sum of first L_j cubes equal to $(\delta L)^3(1+4+9+\cdots+(L_{\max}/\delta L)^3) \approx L_{\max}^4 / (4\delta L)$. The step that finds the matrix $[\mathbf{HV}]$ is also not incremental and this appears to be the most costly term, as a result of the sum of first L_j squares which

gives $L_{\max}^3 N/(3\delta L)$. Notice however that using HAT, QR and SVD is much faster than using HAT and SVD. The experimental simulations will verify that. This algorithm is also much useful for introducing the next ones with which it shares many steps.

3.6 Eigen based

The following algorithms exploit Eigen Value Decomposition (EVD). The real symmetric matrix $\mathbf{H}^T\mathbf{H}$ can be decomposed into $\mathbf{Q}\mathbf{\Lambda}\mathbf{Q}^T$ via EVD, where $\mathbf{Q}^{L\times L}$ is the column orthonormal matrix containing the eigenvectors in its columns, and $\mathbf{\Lambda}^{L\times L}$ is the diagonal matrix containing the Eigenvalues. Since $\mathbf{H}^T\mathbf{H}$ is symmetric it holds that $\mathbf{Q}^{-1} = \mathbf{Q}^T$, $(\mathbf{H}^T\mathbf{H})^{-1} = \mathbf{Q}\mathbf{\Lambda}^{-1}\mathbf{Q}^T$ and $\mathbf{Q}^T\mathbf{Q}=\mathbf{I}=\mathbf{Q}\mathbf{Q}^T$. Hence we can validate cost-effectively several values of C via:

$$\mathbf{H}^T\mathbf{H}+\mathbf{I}/C = \mathbf{Q}\mathbf{\Lambda}\mathbf{Q}^T + \mathbf{I}/C = \mathbf{Q}(\mathbf{\Lambda}+\mathbf{I}/C)\mathbf{Q}^T \quad (3.24)$$

from which the inverse is easily produced in terms of the reusable matrices $\mathbf{Q}$ and $\mathbf{\Lambda}$ as:

$$(\mathbf{H}^T\mathbf{H}+\mathbf{I}/C)^{-1} = \mathbf{Q}(\mathbf{\Lambda}+\mathbf{I}/C)^{-1}\mathbf{Q}^T \quad (3.25)$$

3.6.1 Leave-one-out cross-validation with HAT and EVD

Algorithm 6 uses leave-one-out cross-validation with HAT and EVD and is suitable for large datasets. The first step here is to perform Eigen decomposition in order to find the regularized inverse $(\mathbf{H}^T\mathbf{H}+\mathbf{I}/C)^{-1} = \mathbf{Q}(\mathbf{\Lambda}+\mathbf{I}/C)^{-1}\mathbf{Q}^T$ and in this way to test cheaply several values of the parameter C, by using the same orthogonal matrix $\mathbf{Q}$ which is reusable. The next step is to store the matrix $\mathbf{HQ}$ in order to find the HAT matrix.

Algorithm 6. *Leave-one-out cross-validation via the HAT matrix and EVD for $L_j=\delta L$ up to $L_j=L_{\max}$ random hidden neurons with step δL and 30 C_i regularization parameters*	*Operations*	*total*
1. Generate and store the hidden layer matrix $\mathbf{H}^{L\max}$	---	---
2. Store the matrix $[(\mathbf{H}^{L\max})^T\mathbf{H}^{L\max}]$	$L_{\max}^2 N$	$L_{\max}^2 N$
3. Store the vector $(\mathbf{H}^{L\max})^T\mathbf{y}$	$L_{\max} N$	$L_{\max} N$
4. For each L_j value range from 0 to $L_{\max}$ with step δL	---	---
4.1 Set $L_j = L_j + \delta L$, $\mathbf{H} = \mathbf{H}^{Lj}$, $\mathbf{H}^T\mathbf{H} = (\mathbf{H}^{Lj})^T\mathbf{H}^{Lj}$	---	---
4.2 Perform EVD via $\mathbf{H}^T\mathbf{H} = \mathbf{Q}\mathbf{\Lambda}\mathbf{Q}^T$	$4L_j^3$ (Not incremental)	$4L_{\max}^4/4\delta L$
4.3 Store the matrix $[\mathbf{HQ}]$	$L_j^2 N$ (Not incremental)	$L_{\max}^3 N/3\delta L$
4.4 Calculate and store the vector $[\mathbf{Q}^T\mathbf{H}^T\mathbf{y}]$	$L_j N$ (Not incremental)	$L_{\max}^2 N/2\delta L$
4.5 For each one C_i of the 30 values of C parameter	---	---
4.5.1 Find the weights $\mathbf{\beta} = \mathbf{Q}(\mathbf{\Lambda}+\mathbf{I}/C_i)^{-1}[\mathbf{Q}^T\mathbf{H}^T\mathbf{y}]$	L_j^2 (Depend on C)	$30L_{\max}^3/3\delta L$
4.5.2 Find the residuals $\mathbf{e} = \mathbf{y} - \mathbf{H}\mathbf{\beta}$	$L_j N$ (Depend on C)	$30L_{\max}^2 N/2\delta L$
4.5.3 Find all values $hat_{nn} = \sum_k^L [\mathbf{HQ}]_{nk}^2/(\lambda_{kk}+1/C_i)$	$L_j N$ (Depend on C)	$30L_{\max}^2 N/2\delta L$
4.5.4 Set $E(C_i,L_j) = (1/N)\sum_n^N (e_n)^2/(1-hat_{nn})^2$	---	---
5. Select the pair (C_i,L_j) with the minimum error $E(C_i,L_j)$	---	---

The overall cost of operations of algorithm 6, excluding some minor terms, is the sum of the terms in the column 'total' and is given in table 3.1, assuming the workload of EVD is about $4L^3$ operations [34]. The not incremental terms are again summed up in computational series in the third column of algorithm 6.

Note that since HAT = $\mathbf{H}(\mathbf{H}^{\mathrm{T}}\mathbf{H}+\mathbf{I}/C)^{-1}\mathbf{H}^{\mathrm{T}}$ = $(\mathbf{HQ})$ $(\mathbf{\Lambda}+\mathbf{I}/C)^{-1}\mathbf{Q}^{\mathrm{T}}\mathbf{H}^{\mathrm{T}}$ we can find all the N diagonal values $hat_{nn} = \mathbf{h}(\mathbf{x}_n)\mathbf{Q}(\mathbf{\Lambda}+\mathbf{I}/C)^{-1}\mathbf{Q}^{\mathrm{T}}\mathbf{h}^{\mathrm{T}}(\mathbf{x}_n)$ by using the cached matrix $\mathbf{HQ}$ as:

$$hat_{nn} = \sum_{k}^{L} [\mathbf{HQ}]_{nk} [\mathbf{HQ}]_{nk}/(\lambda_{kk}+1/C) \tag{3.26}$$

where $[\mathbf{HQ}]_{nk}$ denotes the hq_{nk} element of the matrix $\mathbf{HQ}$, and λ_{kk} denotes a k^{th} diagonal element of the matrix $\mathbf{\Lambda}$.

The step for computing and storing $[\mathbf{HQ}]$ is the most costly term and cannot be avoided since all the N rows of $[\mathbf{HQ}]$ are needed for finding the diagonal values hat_{nn} of HAT.

For deriving the cost of the weight vector $\boldsymbol{\beta}$ note that its j^{th} element is given by $\beta_j = \sum_{k}^{L} q_{jk} [\mathbf{Q}^{\mathrm{T}}\mathbf{H}^{\mathrm{T}}\mathbf{y}]_k/(\lambda_{kk}+1/C)$ where q_{jk} is an element in the j^{th} row and k^{th} column of the matrix $\mathbf{Q}$, λ_{kk} denotes the k^{th} diagonal element of the matrix $\mathbf{\Lambda}$, and $[\mathbf{Q}^{\mathrm{T}}\mathbf{H}^{\mathrm{T}}\mathbf{y}]_k$ is the k^{th} element of the vector $[\mathbf{Q}^{\mathrm{T}}\mathbf{H}^{\mathrm{T}}\mathbf{y}]$. This proves that the cost for the weights is L^2 operations.

3.6.2 Scalable k-fold cross-validation and EVD

Algorithm 7 shows the computational steps of the scalable k-fold cross-validation with Eigenvalue decomposition. Note that like the efficient 10-fold CV in algorithm 1 the algorithm 7 also calculates and stores once the entire matrix $\mathbf{H}^{\mathrm{T}}\mathbf{H}$ as well as entire vector $\mathbf{H}^{\mathrm{T}}\mathbf{y}$ before the splitting into folds, and for each f fold it removes the influence $\mathbf{H_f}^{\mathrm{T}}\mathbf{H_f}$ from matrix $\mathbf{H}^{\mathrm{T}}\mathbf{H}$ in order to produce the corresponding matrix $\mathbf{H_{-f}}^{\mathrm{T}}\mathbf{H_{-f}}$ for this fold. There is a difference. While the efficient 10-fold CV in algorithm 1 proceeds by executing first the "for each L_j value" loop and then the "for each fold f" loop is placed inside it, the scalable 10-fold CV in algorithm 7 executes the "for each fold f" loop first and the "for each L_j value" loop inside it. In consequence, algorithm 7 uses EVD on each matrix $\mathbf{H_{-f}}^{\mathrm{T}}\mathbf{H_{-f}} = \mathbf{Q\Lambda Q}^{\mathrm{T}}$ which is more suitable for large datasets.

The cost for the weight vector $\boldsymbol{\beta}$ can again be derived as in algorithm 6. It can also be shown graphically by symbolizing $s_{kk} = 1/(\mathbf{\Lambda}_{kk}+1/C)$ and $o_k = [\mathbf{Q}^{\mathrm{T}}(\mathbf{H_{-f}})^{\mathrm{T}}\mathbf{y_{-f}}]_k$ then:

$$\mathbf{Q} = \begin{bmatrix} q_{1,1} & q_{1,2} & \cdots & q_{1,L} \\ q_{2,1} & q_{2,2} & \cdots & q_{2,L} \\ \vdots & \vdots & \ddots & \vdots \\ q_{L,1} & q_{L,2} & \cdots & q_{L,L} \end{bmatrix} \quad (\mathbf{\Lambda}+\mathbf{I}/C)^{-1} = \begin{bmatrix} s_{1,1} & 0 & \cdots & 0 \\ 0 & s_{2,2} & \cdots & 0 \\ \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & \cdots & s_{L,L} \end{bmatrix} \quad [\mathbf{Q}^{\mathrm{T}}(\mathbf{H_{-f}})^{\mathrm{T}}\mathbf{y_{-f}}] = \begin{bmatrix} o_1 \\ o_2 \\ \vdots \\ o_L \end{bmatrix}$$

We can see that the diagonal elements of $(\mathbf{\Lambda}+\mathbf{I}/C)^{-1}$ are multiplied with the vector $[\mathbf{Q}^{\mathrm{T}}(\mathbf{H_{-f}})^{\mathrm{T}}\mathbf{y_{-f}}]$ producing an intermediate vector $\mathbf{v}$ that has elements $(s_{kk}\cdot o_k)$, and then each weight value β_i is given from the multiplication of one row i^{th} of $\mathbf{Q}$ with this vector $\mathbf{v}$ ($\beta_i = \mathbf{Q_i}\cdot\mathbf{v}$), which gives L_j^2 operations.

An interesting observation so far is that a well-organized implementation of 10-fold cross validation becomes as important as the choice of the linear algebra solver we use, since the computational cost differences between the 10-fold CV versions (naive=$300L^2N$, standard=$10L^2N$, efficient=$2L^2N$) are much larger than the differences in the computational cost of the linear algebra solvers. For 30 C_i values the 30 Cholesky needs $30L^3/3$ while one EVD needs $4L^3$. The difference $10L^2N - 2L^2N$ between the

standard and efficient 10-fold CV version is more substantial than the difference $10L^3$–$4L^3$, between thirty Cholesky and one EVD.

Algorithm 7. *Scalable 10-fold cross-validation and EVD for L_j=δL up to L_j=L_{max} random hidden neurons with step δL and 30 C_i regularization parameters*	*Operations*	*total*
1. Generate and store the hidden layer matrix $\mathbf{H}^{L\max}$	---	---
Split $\mathbf{H}^{L\max}$ into 10 folds as $[\mathbf{H_1}^{L\max} \dots \mathbf{H_f}^{L\max} \dots \mathbf{H_{10}}^{L\max}]^T$	---	---
Split $\mathbf{y}$ into 10 folds as $\mathbf{y} = [\mathbf{y_1} \dots \mathbf{y_f} \dots \mathbf{y_{10}}]^T$	---	---
2. Store the square matrix $[(\mathbf{H}^{L\max})^T\mathbf{H}^{L\max}]$	$L_{\max}^2 N$	$L_{\max}^2 N$
3. Store the vector $(\mathbf{H}^{L\max})^T \mathbf{y}$	$L_{\max} N$	$L_{\max} N$
4. Set $\mathbf{H} = \mathbf{H}^{L\max}$, $\mathbf{H}^T\mathbf{H} = [(\mathbf{H}^{L\max})^T\mathbf{H}^{L\max}]$	---	---
5. For each one of the f=1 to 10 folds	---	---
5.1 Store the matrix $[(\mathbf{H_f}^{L\max})^T\mathbf{H_f}^{L\max}]$	$L_{\max}^2 N/10$	$L_{\max}^2 N$
5.2 $(\mathbf{H_{-f}}^{L\max})^T\mathbf{H_{-f}}^{L\max} = [(\mathbf{H}^{L\max})^T\mathbf{H}^{L\max}] - (\mathbf{H_f}^{L\max})^T\mathbf{H_f}^{L\max}]$	$L_{\max}^2$	$10 L_{\max}^2$
5.3 $[(\mathbf{H_{-f}}^{L\max})^T \mathbf{y_{-f}}] = (\mathbf{H}^{L\max})^T \mathbf{y} - [(\mathbf{H_f}^{L\max})^T\mathbf{y_f}]$	$L_{\max} N/10$	$L_{\max} N$
5.5 For each L_j value range from 0 to L_{max} with step δL	---	---
5.5.7 Set $L_j = L_j + \delta L$	---	---
5.5.1 Set $[(\mathbf{H_{-f}})^T \mathbf{y_{-f}}]$ equal to the first L_j elements of the vector $[(\mathbf{H_{-f}}^{L\max})^T \mathbf{y_{-f}}]$	---	---
5.5.2 Set $\mathbf{H_f}$ equal to the first L_j columns of $\mathbf{H_f}^{L\max}$	---	---
5.5.3 Set $\mathbf{H_{-f}}^T\mathbf{H_{-f}}$ equal to the first L_j rows and columns of the matrix $(\mathbf{H_{-f}}^{L\max})^T\mathbf{H_{-f}}^{L\max}$	---	---
5.5.4 Perform EVD on $\mathbf{H_{-f}}^T\mathbf{H_{-f}} = \mathbf{Q\Lambda Q}^T$	$4L_j^3$ } Not	$40L_{\max}^4/4\delta L$
5.5.5 Store the intermediate vector $\mathbf{Q}^T[(\mathbf{H_{-f}})^T \mathbf{y_{-f}}]$	L_j^2 } incremental	$10L_{\max}^3/3\delta L$
5.5.6 For each one C_i of the 30 values of C parameter	---	---
5.5.6.1 Find weights $\boldsymbol{\beta} = \mathbf{Q}(\mathbf{\Lambda}+\mathbf{I}/C_i)^{-1}[\mathbf{Q}^T (\mathbf{H_{-f}})^T \mathbf{y_{-f}}]$	L_j^2 } Depend	$300L_{\max}^3/3\delta L$
5.5.6.2 Find residuals for the f fold $\mathbf{e_f} = \mathbf{y_f} - \mathbf{H_f}\boldsymbol{\beta}$	$L_j N/10$ } on C	$30L_{\max}^2 N/2\delta L$
5.5.6.3 Add to Errors $E(C_i,L_j) = E(C_i,L_j) + (\mathbf{e_f})^T\mathbf{e_f}$	---	---
6. Select the pair (C_i,L_j) with the minimum error $E(C_i,L_j)$	---	---

The overall cost of operations of algorithm 7, excluding minor terms, is the sum of the terms in the column 'total' and is given in table 3.1. The fact that there exist no costly term $L_{\max}^3 N/3\delta L$, like the one that dominates in all the previous algorithms, makes algorithm 7 more suitable for large datasets.

It is worth mentioning that for a given dataset all the 10-fold Cross-Validation algorithms produce the same outcome, the same weights, and the same error. The same holds for the all the HAT-based algorithms. This is not surprising since they are based on very solid matrix decompositions. In addition all these algorithms are robust solutions since they remove ill-conditioning via regularization.

We can also note that the parallelization of all the steps in algorithms 6 and 7 is possible by using simple data-parallel matrix-matrix multiplications. Since the real matrix $\mathbf{H}^T\mathbf{H}$ is symmetric the Eigenvalue Decomposition step can also be fully parallelized. Symmetric matrices of this kind are special, first because they have only real eigenvalues and second also because the Hessenberg transformation reduces the matrix to a symmetric tridiagonal matrix (see [33] for details). For this symmetric

eigenvalue problem some fully data-parallel scalable algorithms are available, like the one developed by Dongarra and Sorensen [35] which is based on the QR iteration and the divide-and-conquer approach.

3.7 Summary of Cost Analysis

Table 3.1 shows the summary of the overall computational costs in all the algorithms, after removal of some minor terms for clarity reasons. The results clearly indicate that the most costly term $L_{\max}^3 N/3$ is absent only in the last algorithm.

Table 3.1. The overall costs of the seven algorithms.

Algorithm	**Overall cost**
1. Efficient CV_Cholesky	$L_{\max}^2 N + \frac{1}{\delta L}(\boxed{L_{\max}^3 N/3} + 15L_{\max}^2 N + 25L_{\max}^4 + 100L_{\max}^3)$
2. HAT_Cholesky	$L_{\max}^2 N + \frac{1}{\delta L}(\boxed{30L_{\max}^3 N/3} + 10L_{\max}^2 N) + 20L_{\max}^3$
3. CV_SVD	$\frac{1}{\delta L}(\boxed{40L_{\max}^3 N/3} + 25L_{\max}^2 N + 25L_{\max}^4 + 100L_{\max}^3)$
4. HAT_SVD	$\frac{1}{\delta L}(\boxed{4L_{\max}^3 N/3} + 30L_{\max}^2 N + 5L_{\max}^4/2 + 10L_{\max}^3)$
5. HAT_QR_SVD	$2L_{\max}^2 N + \frac{1}{\delta L}(\boxed{L_{\max}^3 N/3} + 45L_{\max}^2 N + 5L_{\max}^4/2)$
6. HAT_EVD	$L_{\max}^2 N + \frac{1}{\delta L}(\boxed{L_{\max}^3 N/3} + 30L_{\max}^2 N + L_{\max}^4 + 15L_{\max}^3)$
7. Scalable CV_EVD	$2L_{\max}^2 N + \frac{1}{\delta L}(15L_{\max}^2 N + 10L_{\max}^4 + 100L_{\max}^3)$

From table 3.1 we can infer that the most scalable method is algorithm 7 which is the only one that did not contain the most costly term $L_{\max}^3 N/3$ that is dominant.

Are HAT-based solutions better than 10-fold CV solutions? That depends on the type of decomposition. With Cholesky the HAT-based solution is slower than the 10-fold CV. With SVD the HAT-based solution is faster than 10-fold CV. The same holds for the HAT with QR and SVD (the 10-fold CV solution with QR and SVD was very slow, since ten QR were required and we decided not to report it here for saving space). With Eigen Value Decomposition the HAT-based solution is slower than 10-fold CV.

Slow methods are the algorithms 2 and 3: HAT and Cholesky, 10-fold CV and SVD.

Fast method is the algorithm 4: HAT and SVD.

Fast & scalable methods are the algorithms 1, 5, 6 and 7: Efficient 10-fold CV and Cholesky, HAT with QR and SVD, HAT and EVD, Scalable 10-fold CV and EVD.

A straightforward theoretical analysis that will reveal why algorithm 7 is the fastest, can be performed by setting $L_{\max} = a\sqrt{N}$, and $\delta L = a\sqrt{N}/40$. We will examine only the *fast & scalable* algorithms 5, 6 and 7 whose overall cost after simplifications become:

Algorithm 5): $a^2N^2\left(2+\frac{40}{a\sqrt{N}}\left(\frac{a\sqrt{N}}{3}+45+\frac{5a^2}{2}\right)\right)=a^2N^2\left(\frac{46}{3}+\frac{1800}{a\sqrt{N}}+\frac{100a}{\sqrt{N}}\right)$

Algorithm 6): $a^2N^2\left(1+\frac{40}{a\sqrt{N}}\left(\frac{a\sqrt{N}}{3}+30+a^2+\frac{15a}{\sqrt{N}}\right)\right)=a^2N^2\left(\frac{43}{3}+\frac{1200}{a\sqrt{N}}+\frac{40a}{\sqrt{N}}+\frac{600}{N}\right)$

Algorithm 7): $a^2N^2\left(2+\frac{40}{a\sqrt{N}}\left(15+10a^2+\frac{100a}{\sqrt{N}}\right)\right)=a^2N^2\left(2+\frac{600}{a\sqrt{N}}+\frac{400a}{\sqrt{N}}+\frac{4010}{N}\right)$

We can see that when N grows larger all the terms with denominators $\sqrt{N}$ diminish, and only the first term in the parenthesis becomes important. This means that for fixed δL steps whatever the factor a in $L_{\max} = a\sqrt{N}$ is, there is always one N value beyond which the speed of algorithm 7 outperforms all the others. In terms of scalability with respect to N this algorithm is the most promising. The same conclusion is also supported by various experimental simulations.

3.8 Experimental Simulations

We use Numerical Recipes in C++ [34] to implement all matrix decompositions and run the methods. The cross-validation proceeds by testing L_j hidden neurons from δL to $L_{\max}$ with step δL, meaning $L_{\max}/\delta L$ incrementally increased different values of L_j hidden neurons. In the set of experiments we simulate a realistic situation in which $L_{\max}$ is a function of the number N of training examples, since $L_{\max}$ depends on how large is the training dataset. By setting $L_{\max} = a\sqrt{N}$ and $\delta L = a\sqrt{N}/40$ we use a = 2, 4, and 6 in the experiments for examining how the methods scale by varying the number N of training examples.

For each algorithm the computational time by using $L_{\max} = 2\sqrt{N}$ and testing 30 C_i values is plotted in fig. 3.6 as a function of N. The ranking from the slowest algorithm, which is algorithm 3 CV_SVD, to the fastest one, which is 7 scalable_CV_EVD, is displayed in the right side of fig. 3.6. The results are similar in fig. 3.7 which displays time and rankings for 50 C_i values.

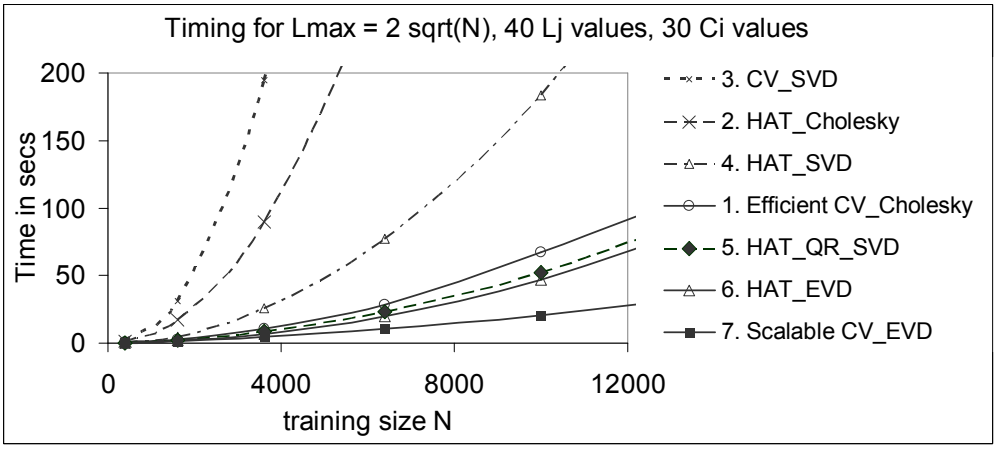


Figure 3.6. Time in seconds as a function of the training size N by using $L_{\max} = 2\sqrt{N}$ and testing 40 L_j values of hidden neurons and 30 C_i values of regularization parameters. The corresponding method for each curve and their ranking is presented in the right side.

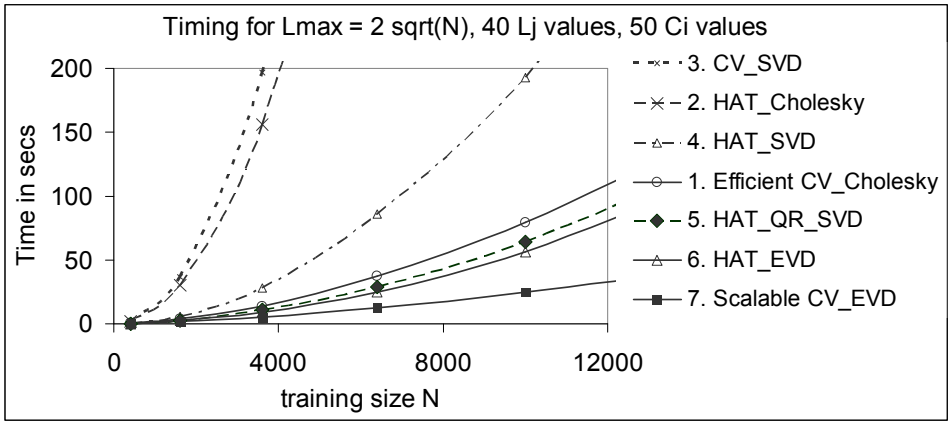


Figure 3.7. Time in seconds versus training size N by using $L_{\max} = 2\sqrt{N}$ and testing 40 L_j values of hidden neurons and 50 C_i values of regularization parameters.

By setting $L_{\max} = 4\sqrt{N}$ and testing 40 L_j values, the results for the computational time as a function of N are plotted in figs 3.8 and 3.9. Fig 3.8 corresponds to 30 C_i values and fig. 3.9 shows the results for 50 C_i values. The ranking from the slowest algorithm 3 CV_SVD, to the fastest one, which is algorithm 7 scalable_CV_EVD, is displayed in the right side of these figs.

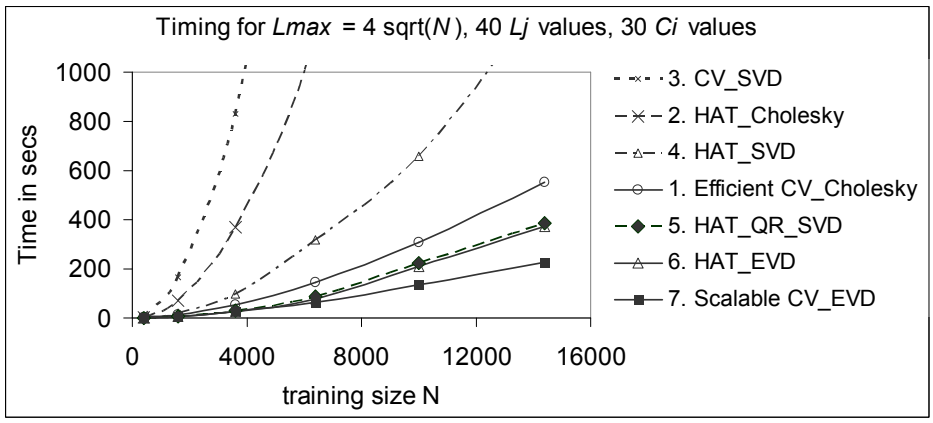


Figure 3.8. Time in seconds versus size N by using $L_{\max} = 4\sqrt{N}$ and testing 40 L_j values of hidden neurons and 30 C_i values of regularization parameters. The ranking for each curve is presented in the right.

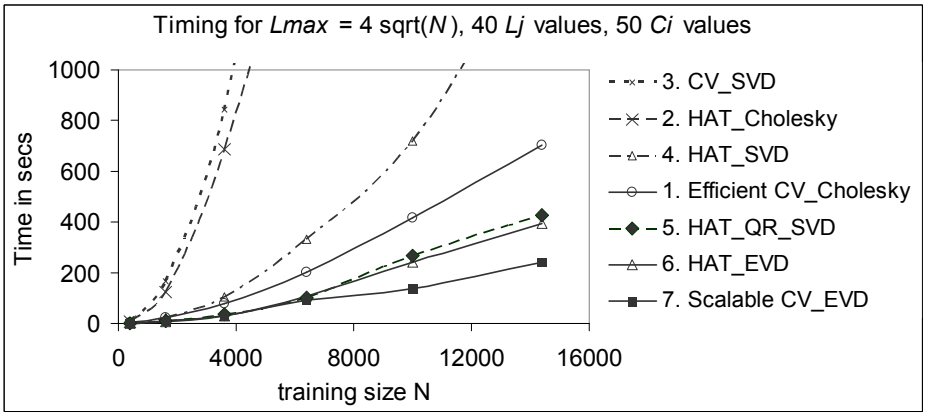


Figure 3.9. Time in seconds versus training size N by using $L_{\max} = 4\sqrt{N}$ and testing 40 L_j values of hidden neurons and 50 C_i values of regularization parameters.

The timing results for $L_{\max} = 6\sqrt{N}$, 40 L_j values and 30 C_i values as a function of N are plotted in fig. 3.10. The fastest one is again algorithm 7 scalable CV_EVD, while the second in speed now becomes algorithm 5 HAT_QR_EVD, which is slightly below algorithm 6 HAT_EVD.

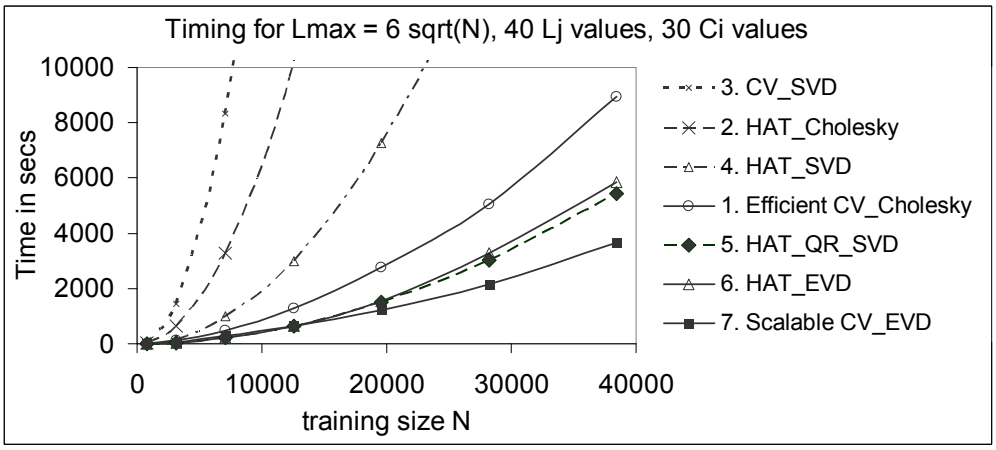


Figure 3.10. Time in seconds versus training size N by using $L_{\max} = 6\sqrt{N}$ and testing 40 L_j values of hidden neurons and 30 C_i values of regularization parameters.

3.9 Summary

Managing the computational cost of model selection and cross-validation in Extreme Learning Machines depends on the type of the matrix decomposition as well as the type of cross-validation we decide to use. To this end we conduct a thorough comparison of Cholesky, SVD, QR and Eigen Value Decomposition (EVD) which can produce matrices that are reusable many times. These matrix decompositions have not been previously compared in the literature in conjunction with the cross-validation strategies for revealing which is fast and most scalable for selecting the best ELM model. The alternatives for ELM cross-validation are presented in seven most representative algorithms for which we derive the theoretical cost of operations, and conduct various experimental simulations.

While we have found that the type of decomposition plays one important role in the cost of cross-validation another possibly more important role plays the type of cross-validation. Hence, we also compare the virtual leave-one-out Cross Validation via the HAT matrix, and efficient 10-fold Cross-Validation, as well as scalable 10-fold Cross-Validation version. The last version can save a huge amount of computational time.

The fast and scalable ELM model selection methods are efficient 10-fold CV with Cholesky decomposition, HAT matrix with QR and SVD, HAT matrix with EVD and the fastest of all is the last one, namely the scalable 10-fold CV with EVD. From the theoretical analysis we identify the most expensive term $L_{\max}^3 N / 3$ that is dominant in the computational cost of all the algorithms except the last one, where this term is absent, and this is the main reason why it is the fastest. From the implementation point of view there is no difference in the delivered outcome of the algorithms and the only difference resides in their computing time. What ever $L_{\max} = a\sqrt{N}$ we set there is always one N value beyond which scalable 10-fold CV with EVD outperforms all the others. Another interesting conclusion is that a well-organized implementation of 10-fold Cross Validation can become as important as the choice of the matrix

decomposition method we use. As we have already point out in the text the difference $10L^2N-2L^2N$ between a standard 10-fold CV and the efficient 10-fold CV version that uses caching $\mathbf{H}^T\mathbf{H}$ and subtracting $\mathbf{H}_{-f}^T\mathbf{H}_{-f}$ per f fold is more substantial than the difference $10L^3-4L^3$, that is produced between the required 30 Cholesky calls and the one EVD call when we had to test 30 C_i values. Such scalable cross-validation versions deserve further consideration and there is future merit in this observation, since other algorithms like incomplete principal Cholesky or non-negative matrix factorizations or regularized orthogonal least squares can be potentially used for guiding another important aspect that is the network pruning. Future experiments could explore such a possibility.

References for chapter 3

[1] Huang G.-B., Zhu Q.-Y., Siew C.-K. (2006) Extreme learning machine: theory and applications. Neurocomputing 70, 489–501.

[2] Huang G.-B., Wang D., Lan Y., (2011) Extreme learning machine: A survey. International Journal of Machine Learning and Cybernetics 2(2), 107–122.

[3] Huang G.-B., Zhou H., Ding X., Zhang R. (2012) Extreme learning machine for regression and multiclass classification. IEEE Trans Syst Man Cybern Part B. 42(2), 513–529.

[4] Liu X.Y., Gao C.H., Li P. (2012) A comparative analysis of support vector machines and extreme learning machines. Neural Networks 33, 58–66.

[5] Huang G., Huang G.-B., Song S., You K. (2015) Trends in extreme learning machines: A review. Neural Networks 61, 32–48.

[6] Hastie T., Tibshirani R., Friedman J.H. (2001) The Elements of Statistical Learning: Data Mining, Inference, and Prediction. Springer.

[7] Stone M. (1974) Cross-validatory choice and assessment of statistical predictions. J. R. Stat. Soc. B 36(1), 111–147.

[8] Kohavi R. (1995) A study of cross-validation and bootstrap for accuracy estimation and model selection. In Proceedings of the 14th International Joint Conference on Artificial Intelligence (IJCAI95), pp. 1137–1143.

[9] Liao S., Feng C. (2014) Meta-ELM: ELM with ELM hidden nodes. Neurocomputing 128, 81–87.

[10] Golub G.H., Heath M., Wahba G. (1979) Generalized cross-validation as a method for choosing a good ridge parameter. Technometrics, 21(2), 215–223.

[11] Elden L. (1984) A note on the computation of the generalized cross-validation function for ill-conditioned least squares problems, BIT, 24, 467-472.

[12] Wahba G. (1991) Spline Models for Observational Data. SIAM Philadelphia.

[13] Hoaglin D.C., Welsch R.E. (1978) The Hat Matrix in Regression and ANOVA. The American Statistician 32 (1), 17–22.

[14] Allen D.M. (1974) The relationship between variable selection and prediction. Technometrics 16, 125–127.

[15] Luntz A., Brailovsky V. (1969) On estimation of characters obtained in statistical procedure of recognition (in Russian). Techicheskaya Kibernetica 3

[16] Green P.J., Silverman B.W. (1994) Nonparametric Regression and Generalized Linear Models. Number 58 in Monographs on Statistics and Applied Probability. Chapman & Hall.

[17] Schölkopf B., Smola A. (2002) Learning with kernels: Support vector machines, regularization, optimization, and beyond. MIT Press, Cambridge, MA.

[18] Rifkin R., Yeo G., Poggio T. (2003) Regularized least-squares classification. Nato Science Series Sub Series III Computer and Systems Sciences 190, 131–54.

[19] Cawley G.C., Talbot N.L.C. (2003) Efficient leave-one-out cross-validation of kernel Fisher discriminant classifiers, Pattern Recognition 36, 2585–2592.

[20] Rifkin R.M., Lippert R.A. (2007) Notes on regularized least squares. Technical report, MIT Computer Science and Artificial Intelligence Laboratory.

[21] Cawley G.C., Talbot N.L.C. (2007) Preventing Over-Fitting during Model Selection via Bayesian Regularisation of the Hyper-Parameters. Journal of Machine Learning Research 8, 841-861.

[22] Miche Y., Sorjamaa A., Bas P., Simula O., Jutten C., Lendasse A. (2010) OP-ELM: optimally pruned extreme learning machine. IEEE Transactions on Neural Networks 21(1), 158–162.

[23] van Heeswijk M., Miche Y., Oja E., Lendasse A. (2011) GPU-accelerated and parallelized ELM ensembles for large-scale regression. Neurocomputing 74, 2430–2437.

[24] Miche Y., van Heeswijk M., Bas P., Simula O., Lendasse A. (2011) TROP-ELM: A double-regularized ELM using LARS and Tikhonov regularization. Neurocomputing 74, 2413–2421.

[25] Yu Q., Miche Y., Eirola E., van Heeswijk M., Séverin E., Lendasse A. (2013) Regularized extreme learning machine for regression with missing data. Neurocomputing 102, 45–51.

[26] Golub G.H., Loan C.F.V. (1996) Matrix Computations. 3rd ed., John Hopkins University Press, Baltimore, MD.

[27] Stewart G.W. (2000) Decompositional approach to matrix computation. Computing in Science and Engineering, 2(1), 50–59.

[28] Fukunaga K. (1990) Introduction to Statistical Pattern Recognition. Academic Press, USA.

[29] Israel A.B., Greville T.N.E. (2003) Generalized Inverses: Theory and Applications. 2nd ed., New York, Springer.

[30] Courrieu P., (2005) Fast Computation of Moore-Penrose Inverse Matrices. Neural Information Processing - Letters and Reviews 8(2), 25–29

[31] Rakha M.A. (2004) On the Moore-Penrose generalized inverse matrix. Applied Mathematics and Computation 158, 185–200.

[32] Wei Y., Wang G. (2002) PCR algorithm for parallel computing minimum-norm (T) least-squares (S) solution of inconsistent linear equations. Applied Mathematics and Computation 133, 547–557.

[33] Karniadakis G.E., Kirby R.M. (2003) Parallel Scientific Computing in C++ and MPI. Cambridge University Press.

[34] Press W.H., Teukolsky S.A., Vetterling W.T., Flannery B.P. (2002) Numerical Recipes in C++: The Art of Scientific Computing. 2nd ed., Cambridge University Press.

[35] Dongarra J.J., Sorensen D.C. (1987) A fully parallel algorithm for the symmetric eigenvalue problem. SIAM J. Sci. and Stat. Comp. 8, S139–S154.

4 Big data regression with Parallel Enhanced and Convex Incremental ELM

This chapter considers scalable Incremental Extreme Learning Machine (I-ELM) algorithms which may well be suitable for big data regression. During the training of Incremental ELMs, the hidden neurons are presented one-by-one, and the weights are based solely on simple direct summations which can be most efficiently mapped on parallel environments. Existing Incremental versions of ELMs are the Incremental ELM (I-ELM), Enhanced Incremental ELM (EI-ELM) and Convex Incremental ELM (CI-ELM). We study the proposed Enhanced Convex Incremental ELM (ECI-ELM) algorithm which is a combination of the last two. The main findings are that the proposed ECI-ELM is fast, accurate and fully scalable when operates in a parallel system of distributed memory workstations. Experimental simulations on several benchmark datasets demonstrate that the ECI-ELM is the most accurate among the existing I-ELM, EI-ELM and CI-ELM algorithms. We also analyze the convergence as a function of the hidden neurons and demonstrate that ECI-ELM has the lowest error rate curve and converges much faster than the other algorithms in all of the datasets. The parallel simulations also reveal that the data parallel training of the ECI-ELM can guarantee simplicity, straightforward mappings and can deliver speedups and scaleups very close to linear.

4.1 Introduction

Owning to the extensive demands of many applications their remarkably large datasets are daily collected in a distributed fashion and regularly stored. As a consequence, it has become important to study, develop, expand and test effective and efficient statistical and machine learning algorithms that can, in principle, operate for data analysis on big data collections, from which the synonymous big data trend has taken its name [1] [2]. It seems that, machine learning and computational intelligence methods have never been as critical and important to real-life applications as they are in the emerging big data era. While many conventional machine learning methods might work poorly on large data collections, parallel processing methods [3][4] [5] can in principal cop with common scalability issues. Such parallel algorithms should be both effective enough to capture the data complexity and scalable enough to process efficiently the data in a parallelized or fully decentralized manner.

Regression is among the traditional data analysis methods utilized for big data analysis [1]. Regression analysis is a mathematical tool for revealing correlations between one variable and several other variables. Neural network algorithms are most suitable for highly non-linear regression analysis although their training is usually slow hampering thus their applicability. It has been suggested [6] that relatively simple models can

effectively capture highly complex and structured problems when they are trained on very large datasets. For solving large scale regression problems one feed forward neural network model with simple architecture and training phase is the Extreme Learning Machine (ELM) [7], which has shown its good performance and fast speed in several applications [8][9][10]. From the implementation point of view, parallel computation techniques have substantially sped up the training of ELMs, making them feasible for large data processing [10].

The Extreme Learning Machine (ELM) [6] is a single-hidden-layer feedforward neural network with randomly generated hidden neurons. The main idea behind ELM relays in the hidden layer neurons which, different from conventional learning methods, are generated randomly and thus their parameters need not to be tuned. The nature of ELM techniques is twofold since they have the universal approximation capability with random hidden layer, and they also allow various learning techniques for training (see review in [9]). Some ELM models rely on the basic ELM training [7][8][9][10] during which the classical regression matrix is used. Other ELM models rely on Incremental ELM [11], Enhanced Incremental ELM (EI-ELM) [12] and Convex Incremental ELM (CI-ELM) [13] in which their fast training algorithms proceed incrementally by adding hidden neurons one-by-one. We present the Enhanced and Convex Incremental ELM algorithm that outperforms the existing incremental versions in terms of accuracy, and it scales very well, exhibiting excellent data parallelization capabilities on distributed memory machines.

The existing approaches of Incremental ELM (I-ELM) [11] approach and its variants, namely Enhanced I-ELM [12] and Convex I-ELM [13] are examples of constructive learning algorithms. I-ELM adds the hidden neurons one-by-one into the growing hidden layer and stops when the number of hidden neurons reaches a maximum or the training error reaches some predefined value. The universal approximation capability of ELM is proved from this incremental learning framework [11]. The computational complexity of the incremental ELM versions, given N training examples, with d features and L hidden neurons, is of the order O(LNd), which is linear in L and N and is also L times faster than the least squares solution of the basic ELM. No extra storage requirement is necessary since for each newly added hidden neuron one pass of this hidden node from all data is the only thing needed. Assuming that in large scale problems many hidden neurons are usually required to cover the space of all N data examples the study of parallel incremental implementations is promising.

The existing Incremental ELM algorithms exploit simple neuron summations and can be implemented in three different ways: 1) original I-ELM [11] in which every time only one hidden neuron is randomly generated and added to the existing network, 2) Enhanced I-ELM [12] in which every time step k hidden neurons are randomly generated but among these k hidden neurons only the most appropriate one will be added to the existing network, 3) Convex I-ELM [13] in which the Barron's convex optimization learning method is incorporated in I-ELM for recalculating the linear output weights of the existing hidden neurons after a new hidden neuron is randomly added.

We study and add to the pile of the previous three incremental ELM algorithms the proposed Enhanced and Convex Incremental ELM (ECI-ELM) which has not been explored before. ECI-ELM algorithm is a combination of Convex I-ELM and Enhanced I-ELM and the experimental comparisons with the other incremental ELMs reveal that it delivers better accuracy performance on several benchmark datasets.

Learning algorithms that are based solely on simple direct summations are worth exploiting since their parallelism can, in principle, scale up to several processors. The incremental versions of ELM are such algorithms. Their processing favours data parallel applications given that in practice only the hidden neurons are required to be in main memory and the distributed data partitions can be scanned every time a newly arrived hidden neuron must be processed. To this end, we also study data parallelisation of the Enhanced and Convex Incremental ELM algorithm that can be efficiently and effectively trained on fully distributed data partitions. The main contribution and the central result is that the proposed Enhanced Convex Incremental ELM (ECI-ELM) is one of fast algorithm for accurate regression and it is also fully parallelizable on distributed memory machines.

For the parallel performance measurements we use a system of distributed memory workstations. In this parallel architecture, each processor has its own memory and does not share disk drives or random access memory with the others. Processors can communicate with one another by sending messages through an interconnection Ethernet network. This shared-nothing architecture is characteristic of mass storage distributed database systems which have the main advantage that can scaled up to multiple distributed processors. The parallel performance measurements reveal that the proposed ECI-ELM algorithm has excellent speedup and scaleup behavior.

The rest of the chapter is organized as follows. Section 2 provides short literature review on existing incremental ELM algorithms and existing parallelization studies of basic ELM. Section 3 presents the training of the existing I-ELM, Enhanced I-ELM and Convex I-ELM as well as the proposed Enhanced Convex I-ELM algorithm. Section 4 describes the Data Parallelism of the Enhanced Convex I-ELM. Section 5 provides experimental results and comparisons on the effectiveness of the proposed method. Section 6 gives summarizes our conclusions.

In the mathematical formulas an uppercase bold letter will symbolize a matrix, while a lowercase bold letter will symbolize a vector, and an italic letter will denote a scalar variable. Fig. 4.1 illustrates the meaning of several symbols and notations used in this chapter.

N – the number of training examples
L – the number of hidden neurons
$\mathbf{x}_n$ – a n^{th} training example
y_n – the target value corresponding to $\mathbf{x}_n$ training example
P – the number of processors
N_p – the number of training examples in p^{th} processor ($N_p = N/P$)
$f_L(\mathbf{x})$ – the ELM predictor that has L hidden neurons
$\mathbf{H}$ – the regression matrix of size $N \times L$
$\mathbf{a}_i$ – input weight parameter for the i^{th} hidden neuron
b_i – bias parameter for the i^{th} hidden neuron
$g_i(\mathbf{x})$ – the activation function $g(\mathbf{x}, \mathbf{a}_i, b_i)$ of the i^{th} hidden neuron
w_i – the linear output weight of the i^{th} hidden neuron
$\mathbf{w}$ – the vector of output weights
$e_L(\mathbf{x}_n)$ – the residual error of $f_L(\cdot)$ predictor for the $\mathbf{x}_n$ example ($e_L(\mathbf{x}_n) = y_n - f_L(\mathbf{x}_n)$)

Figure 4.1. Common symbols and notations.

4.2 Related work and Preliminaries

4.2.1 Existing incremental ELMs

Incremental ELM (I-ELM) for single-hidden-layer feedforward neural networks is an algorithm that is proposed in [11] where Huang and co-workers demonstrate that I-ELM randomly generates hidden neurons one-by-one and then incrementally determines their linear output weights. It has cost of the order of O(*LNd*). It has been proved in theory [11] that although hidden neurons are generated randomly the neural network constructed by I-ELM works as a universal approximator. [11] has shown that I-ELM is a "random search" method that adds random neurons one by one to the hidden layer and freezes the output weights of the existing hidden neurons when a new hidden neuron is added. During such incremental random addition, the residual error of the neural network will decrease and I-ELM will move toward the target function further whenever a new hidden neuron is added.

Enhanced I-ELM was proposed in [12] which illustrated that some of the hidden neurons in I-ELM may play a minor role in the neural network output and thus may eventually increase the network complexity. In order to avoid this issue and to obtain much more compact network architecture they study an enhanced random search based incremental method for I-ELM. At each learning step of Enhanced I-ELM a list of *K* trials for hidden neurons is randomly generated and in that list the hidden neuron that leads to the smallest residual error will be finally added to the existing I-ELM network. The output weights of the network are calculated in a same simple way as in the original I-ELM. Compared with the original I-ELM, it has been demonstrated [12] that EI-ELM can achieve much more compact network architectures. Compact network architecture also implies faster prediction time. Training the Enhanced I-ELM has O(*KLNd*) computational cost. Usually 10 or 20 *K* trials are needed.

Convex I-ELM was proposed in [13] which showed that the convergence rate of I-ELM can be further improved. Here the Barron's convex optimization learning method is incorporated into I-ELM to create a Convex I-ELM [13] that recalculates the linear output weights of the existing hidden neurons after a new hidden neuron is randomly added. This Convex I-ELM has a faster convergence rate while it maintains the I-ELM's simplicity and efficiency. Training the Convex I-ELM has O(*LNd*) computational cost.

Different from the aforementioned studies, in this work we implement the proposed Enhanced Convex I-ELM which has not been investigated so far, although it has been entailed from [12]. Enhanced Convex I-ELM is a direct combination of Convex I-ELM and Enhanced I-ELM. Since I-ELM has been benefited from the Enhanced random searching of hidden neurons, the application of EI-ELM to the Convex I-ELM seems appealing to explore. Training the proposed Enhanced Convex I-ELM has O(*KLNd*) computational cost and we simulate its performance and compare it with the other three incremental ELM variants.

4.2.2 Parallel ELMs

So far the focus on which several ELM parallelization studies concentrated is only the basic version of ELM [7] [8] which is not incremental. The basic ELM [7] [8] structure has a fixed number of randomly generated hidden neurons, that are produced in one step, and not incrementally. For a given set of training examples $\{\mathbf{x}_n, y_n\}_{n=1}^{N}$ in R^d feature

space, the basic ELM algorithm generates L hidden neurons, it calculates the hidden-layer output matrix **H**, namely the regression matrix of size $N \times L$, and finally computes the linear output weights **w** via least squares by $\mathbf{w} = \mathbf{H}^{\dagger}\mathbf{y}$ that best fit the target values **y**, where $\mathbf{H}^{\dagger} = (\mathbf{H}^{T}\mathbf{H})^{-1}\mathbf{H}^{T}$ denotes the Moore–Penrose generalized inverse. Therefore in the basic ELM training [7] [8] calculating the regression matrix **H** has cost $O(LNd)$, that is relatively small when d is much smaller than L, while multiplying **H** to compute the gram matrix $\mathbf{H}^{T}\mathbf{H}$, which has cost $O(L^2N)$, and solving the resulting linear least squares, which has cost $O(L^3)$, are the most demanding tasks. As a result, the existing parallelization studies on ELM rather focus on speeding up the matrix-matrix multiplications and the accompanied least squares solution that follows.

Hence, the parallel ELM study in [14] calculates the matrix-matrix multiplications on parallel machines using the Map Reduce framework that is applied for the Moore–Penrose generalized inverse. Based on the same fact, that is for this generalized inverse in basic ELM training the most expensive computation part is the matrix multiplication operator, other authors [15] also parallelize this part. Thus, as the matrix multiplication operator is decomposable, the work in [15] presents a distributed ELM training that is based on MapReduce framework, which can first calculate the matrix multiplication effectively with MapReduce in parallel, and then calculates the corresponding output weight vector with centralized computations.

Parallel ELM algorithms have been also proposed for ensembles of basic ELM. Works like the parallel implementations of an ensemble of GPU-accelerated ELMs for large regression tasks is presented in [16] [17]. Both regression approaches in [16] and [17] accelerate the least-squares basic ELM solution using GPU-based LAPACK solvers. The only parameter that needs to be determined for basic ELM is the number of hidden neurons. The work in [16] uses a leave-one-out model structure selection of ELM and demonstrates parallelization. In [17] the optimal number of hidden neurons in the ELMs of the ensemble was determined by performing model structure selection through the classical Bayesian information criterion. For classification tasks the work in [18] proposes a parallelized ELM ensemble, based on min–max modular network combiner, where the basic ELM modules were trained in parallel. The multi-class classification problem was divided into smaller binary sub-problems, usually through the one-versus-one strategy. The second step was to utilize ELMs to learn each sub-problem independently. This method also deals with imbalanced classification tasks.

Different from the existing ELM parallelization studies in [14][15][16][17][18] that use the basic ELM, in this work we study the Incremental versions of ELM. In contrast with the basic ELM where the training proceeds by randomly generating all hidden neurons, computing the regression matrix and solving the output weights using the generalized inverse, the Incremental versions of ELM have much lower computational cost. Incremental versions do not use direct matrix-matrix multiplications. The computational cost of training the incremental ELM versions is of the order of $O(LNd)$, and is linear in the number N of training examples. Since the process reduces to computing L vectors of size N, it has probably one of the lower costs we can expect from a learning algorithm. In addition the training phase by adding neurons one-by-one it uses one pass through the data per neuron and thus can be straightforwardly parallelized.

4.3 Training Incremental Extreme Learning Machines

4.3.1 I-ELM, Enhanced I-ELM and Convex I-ELM

With a given training set $\{\mathbf{x}_n, y_n\}_{n=1}^{N}$, the Extreme Learning Machine in fig. 4.2 searches for the best function that can approximate any target value y_n. For an unknown $\mathbf{x}$ the prediction output function $f_L(\mathbf{x})$ of the ELM with L hidden neurons is (eq. 4.1):

$$f_L(\mathbf{x}) = \sum_{i=1}^{L} w_i \cdot g_i(\mathbf{x}) \tag{4.1}$$

where $g_i(\mathbf{x}) = g_i(\mathbf{x}, \mathbf{a}_i, b_i)$ is the activation function of i^{th} hidden neuron, w_i is the linear output weight of the i^{th} hidden neuron and $\mathbf{a}_i$ and b_i are the parameters (input weight and bias) of the i^{th} hidden neuron. We use the sigmoid type activation function for the additive hidden neurons which is defined as $g(\mathbf{x}, \mathbf{a}, b) = 1/(1+\exp(-(\mathbf{a}\cdot\mathbf{x}+b)))$. The core of ELM training consists of randomly generating the hidden neuron parameters $\mathbf{a}_i$ and b_i (usually from the range [−1, 1]) and learning only the linear output weights w.

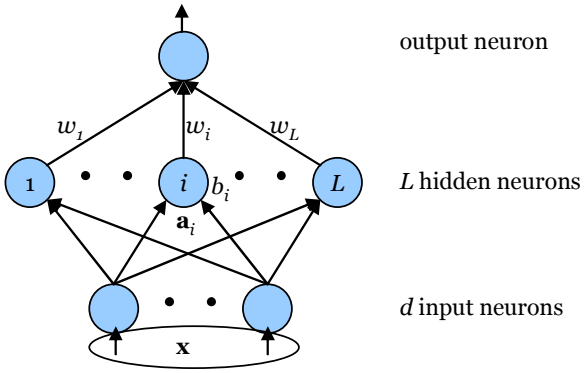


Figure 2. Extreme Learning Machine topology with L additive hidden nodes.

For the I-ELM [11] case it has been proven theoretically that one may freeze the weights of the existing hidden neurons and need not further adjust them when new neurons are added (see [11] for proofs and further references therein). In the I-ELM training when the L hidden neuron is added its output weight w_L is computed by [11]:

$$w_L = \sum_{n=1}^{N} e_{L-1}(\mathbf{x}_n) \cdot g_L(\mathbf{x}_n) \;/\; \sum_{n=1}^{N} (g_L(\mathbf{x}_n))^2 \tag{4.2}$$

where $g_L(\mathbf{x}_n) = g(\mathbf{a}_L \cdot \mathbf{x}_n + b_L)$ is the activation of the new L hidden neuron for each $\mathbf{x}_n$ training example and $e_{L-1}(\mathbf{x}_n) = y_n - f_{L-1}(\mathbf{x}_n)$ is the corresponding residual error of each example $\mathbf{x}_n$ before the new hidden neuron is added. An I-ELM adding step incrementally adds a new neuron and computes its new weight w_L using eq. 4.2.

The Enhanced I-ELM [12] tests in every adding step a small list of K new hidden neurons before adding the best one of them. Thus when K=1 the Enhanced I-ELM reduces to the I-ELM case. The K number is usually ranging from 10 to 20 (see also [12] for details). On initialization the residual vector error $\mathbf{e} = [e(\mathbf{x}_1),\ldots,e(\mathbf{x}_N)]^T$ is set equal to the expected target vector $\mathbf{y} = [y_1,\ldots,y_N]^T$ of the training data set.

Given a maximum number of neurons $L_{\max}$ and the expected learning rate ε as threshold, the Enhanced I-ELM algorithm is as follows [12]:

Enhanced I-ELM algorithm

Initialization Step): Let $L = 0$, set residual errors for each sample $\mathbf{x}_n$ by $e_L(\mathbf{x}_n) = y_n$, for all $\mathbf{y} = [y_1, y_2,\ldots, y_N]^T$ and find initial error norm $||\mathbf{e}|| = \sum_{n=1}^{N} (e_L(\mathbf{x}_n))^2$

Learning Step):

while $L<L_{max}$ and $||\mathbf{e}|| > \varepsilon$

a) increase by one the number of hidden neurons $L = L +1$

b) create a list of K temporary new randomly generated hidden neurons $\{\mathbf{a}_k, b_k\}_{k=1}^{K}$

c) **for** k=1 to K //test each one of the k^{th} neurons in the K-list

i) temporary set $\{\mathbf{a}_L, b_L\} = \{\mathbf{a}_k, b_k\}$ to form the activation function $g_L(\mathbf{x}_n)$

ii) temporary calculate the linear output weight w_L using eq. 4.2

iii)temporary calculate the new residual errors $e_L(\mathbf{x}_n) = e_{L-1}(\mathbf{x}_n) - w_L \cdot g_L(\mathbf{x}_n)$ for each $\mathbf{x}_n$ example, after the temporal addition of the k^{th} neuron in the network

iv) find the corresponding error norm $||\mathbf{e}_{(k)}|| = \sum_{n=1}^{N} (e_L(\mathbf{x}_n))^2$

endfor

d) find in the K-list the k^* neuron with minimum error $||\mathbf{e}_{(k^*)}||$, add this k^* neuron to the hidden layer, set residual vector $\mathbf{e}_L=\mathbf{e}_{(k^*)}$

endwhile

The Convex I-ELM [13] algorithm adds incrementally a new random hidden neuron, computes the new w_L weight and recalculates all the linear output weights w_i of the other existing hidden neurons, instead of freezing them. The Convex I-ELM at every adding step updates the new w_L weight as [13]:

$$w_L = \sum_{n=1}^{N} e_{L-1}(\mathbf{x}_n) \cdot (e_{L-1}(\mathbf{x}_n) - (y_n - g_L(\mathbf{x}_n))) \,/\, \sum_{n=1}^{N} (e_{L-1}(\mathbf{x}_n) - (y_n - g_L(\mathbf{x}_n)))^2 \quad (4.3)$$

calculates the new residual errors for each $\mathbf{x}_n$ example by using:

$$e_L(\mathbf{x}_n) = (1-w_L) \cdot e_{L-1}(\mathbf{x}_n) + w_L \cdot (y_n - g_L(\mathbf{x}_n)) \quad (4.4)$$

and recalculates the weights of all the other existing hidden neurons as:

$$w_i = (1-w_L) \cdot w_i, \qquad i=1,\ldots, L-1 \quad (4.5)$$

where again $g_L(\mathbf{x}_n) = g(\mathbf{a}_L \cdot \mathbf{x}_n + b_L)$ is the activation of the new L hidden neuron for each $\mathbf{x}_n$ example, y_n is the target of this example and $e_{L-1}(\mathbf{x}_n)$ is the corresponding residual error before this new hidden neuron is added.

4.3.2 Proposed Enhanced and Convex I-ELM

The proposed Enhanced Convex I-ELM (ECI-ELM) algorithm is a straightforward combination of EI-ELM and CI-ELM and has not been explored so far, although in [12] it was suggested that EI-ELM with convex optimization is worth investigating in the future. We present ECI-ELM here and further studying it in the experimental comparisons with the other incremental versions. ECI-ELM achieves more compact network architectures than I-ELM and further improves the convergence rate of I-ELM by recalculating the output weights of the existing nodes based on the Barron's convex optimization method. Hence we have the advantages of two algorithms, EI-ELM and CI-ELM, in one. In essence ECI-ELM replaces inside the while loop of the EI-ELM algorithm the steps c)ii and c)iii and uses eq. 4.3 and eq. 4.4 instead, whereas it adds an additional step e) for recalculating the weights of the other existing hidden neurons

instead of freezing them. The full ECI-ELM algorithm is given in detail in the following:

Enhanced Convex I-ELM algorithm

Initialization Step): Let $L = 0$, set residual errors for each sample $\mathbf{x}_n$ by $e_L(\mathbf{x}_n) = y_n$, for all $\mathbf{y} = [y_1, y_2,\ldots, y_N]^T$ and find initial norm $||\mathbf{e}|| = \sum_{n=1}^{N} e_L(\mathbf{x}_n) \cdot e_L(\mathbf{x}_n)$

Learning Step):

while $L<L_{max}$ and $||\mathbf{e}|| > \varepsilon$ ← **Incremental**

a) increase by one the number of hidden neurons $L = L + 1$

b) create a K-list of K temporal new random hidden neurons $\{\mathbf{a}_k, b_k\}_{k=1}^{K}$ ← **Enchanced**

c) **for** k=1 to K (test each one of the k^{th} neurons in the K-list)

i) temporary set $\{\mathbf{a}_L, b_L\} = \{\mathbf{a}_k, b_k\}$ to form the activation function $g_L(\mathbf{x}_n)$

ii) temporary calculate the linear output weight w_L using eq. 4.3 ← **Convex**

iii) temporary find the new residuals $e_L(\mathbf{x}_n)$ for each $\mathbf{x}_n$ using eq. 4,4

iv) find corresponding error norm $\|\mathbf{e}_{(k)}\| = \sum_{n=1}^{N} (e_L(\mathbf{x}_n))^2$

endfor

d) find in the K-list the k^* neuron with minimum error $\|\mathbf{e}_{(k^*)}\|$, ← **Enchanced**

add this k^* neuron to the hidden layer, set residual vector error $\mathbf{e}=\mathbf{e}_{(k^*)}$

e) recalculate the output weights w_i for all existing neurons using eq. 4.5 ← **Convex**

endwhile

Setting K=1 the proposed ECI-ELM algorithm reduces to the Convex I-ELM case. ECI-ELM is incremental and is based solely on simple and direct summations of data vectors. These summations can be mapped efficiently in a data parallel design.

4.4 Data Parallelism of Enhanced Convex I-ELM

In the data parallel approach the training data are partitioned and distributed to P Worker processors. In every iteration step the Master processor holds the current I-ELM structure, namely the input weights $\mathbf{a}_i$, biases b_i and output weights w_i. Each Worker processor p holds a partition of N_p training data ($N_p=N/P$). These training data vectors $\mathbf{x}_n$ along with their target values y_n are stored across the Workers which also store the corresponding residual errors $e_{L-1}(\mathbf{x}_n)$ by using the previous ELM predictions $f_{L-1}(\mathbf{x}_n)$. Hence, each Worker runs the same program code but has its own set of local variables and reads from a different subset of training data. The Master/Worker Data Parallel algorithm for the Enhanced Convex I-ELM is as follows (initialization step is the same as before and is omitted):

Master/Worker Data Parallelism of Enhanced Convex I-ELM

Learning Step):

while $L<L_{max}$ and $||\mathbf{e}|| > \varepsilon$

set $L = L + 1$

Master: create a K-list of K temporal new random hidden neurons $\{\mathbf{a}_k, b_k\}_{k=1}^{K}$

Broadcast the K-list $\{\mathbf{a}_k, b_k\}_{k=1}^{K}$ to workers

Worker p: Receive the K-list $\{\mathbf{a}_k, b_k\}_{k=1}^{K}$ from master

for k=1 **to** K //test each one of the neurons in the K-list

use $\{\mathbf{a}_k, b_k\}$ for the activation function $g_k(\mathbf{x}_n) = g(\mathbf{a}_k \cdot \mathbf{x}_n + b_k)$ and

Compute partial sum $\sum_{n=1}^{Np} (e_{L-1}(\mathbf{x}_n) \cdot (e_{L-1}(\mathbf{x}_n) - (y_n - g_k(\mathbf{x}_n))))$

Compute partial sum $\sum_{n=1}^{Np} (e_{L-1}(\mathbf{x}_n) - (y_n - g_k(\mathbf{x}_n)))^2$

endfor

Master: Reduce all partial sums and find K temporal weights w_k using eq. 4.3

Broadcast the list of K temporal weights $\{w_k\}_{k=1}^{K}$ to workers

Worker p: Receive the list of K temporal weights $\{w_k\}_{k=1}^{K}$

for k=1 **to** K //find K partial sums of residual errors (eq. 4.4)

Compute partial sums $\sum_{n=1}^{Np} ((1-w_k) \cdot e_{L-1}(\mathbf{x}_n) + w_k \cdot (y_n - g_k(\mathbf{x}_n)))^2$

endfor

Master: Reduce all partial sums and finds K error norms $||\mathbf{e}_{(k)}||$

find in the K-list the k^* neuron with minimum error $||\mathbf{e}_{(k^*)}||$

add this k^* neuron to the hidden layer , $w_L = w_{k^*}$, $\{\mathbf{a}_L, b_L\} = \{\mathbf{a}_{k^*}, b_{k^*}\}$

recalculate the weights w_i of all other existing neurons using eq. 4.5

Broadcast the k^* neuron and its weight w_L to workers

Worker p: Receive the k^* neuron and w_L

set new residual errors $e_L(\mathbf{x}_n)$ for each $\mathbf{x}_n$ example using eq. 4.4

endwhile

Setting K=1 the algorithm reduces to the Data Parallelism of Convex I-ELM case.

4.5 Experiments

4.5.1 Benchmark Datasets

In the experiments we use many benchmark datasets (listed in table 4.1) for regression in order to test the accuracy of the proposed Enhanced Convex I-ELM versus the other incremental variants. Rather large datasets have been considered together with some others with moderate or smaller size. Pokerhand, Year Prediction MSD, 3D road network, California Housing, Abalone, are from the UCI repository (http://www.ics.uci.edu/~mlearn/MLRepository.htm). Kinematics, Bank, Computer activity and Puma dynamic can be found in the Delve Repository (http://www.cs.toronto.edu/~delve). The Protein Structure dataset describes the folding structure of different amino acids and can be downloaded from the Infobiotics PSP repository (http://icos.cs.nott.ac.uk/datasets). The rest benchmark datasets like Ailerons, Delta Ailerons, Elevators, Delta Elevators, 2Dplanes, Friedman, Census can be found in the Torgo's site (http://www.dcc.fc.up.pt/~ltorgo/Regression).

Table 4.1. Benchmark datasets characteristics

dataset	instances	features
Pokerhand	1025010	10
Year Prediction MSD	515345	90
3D Road Network	434874	2
Protein Structure PSP	257560	20
2D planes	40768	10
Friedman	40768	10

Census (House8L)	22784	8
California Housing	20640	8
Elevators	16599	18
Ailerons	13750	39
Delta Elevators	9517	6
Puma dynamic	8193	8
Bank	8192	8
Kinematics	8192	8
Delta Ailerons	7129	5
White Wine Quality	4898	11
Abalone	4177	8
Red Wine quality	1599	11

4.5.2 Regression Performance comparisons

Following the classical ELM training guidelines, in the experiments all the input features (attributes) have been normalized into the range [-1, 1] while the outputs (targets) have been normalized into the range [0, 1]. The hidden neuron parameters **a** and b are randomly chosen from the range [-1, 1]. Each dataset is randomly split into a training set (80%) and a test set (20%). The sigmoid function is used for the hidden neuron activation. To compare the prediction accuracy all versions of incremental ELMs employ L = 1000 hidden neurons. The learning procedure is repeated 20 times for each dataset and the results are averaged. We measure the regression performance of each algorithm using the root mean squared error (RMSE). The comparison results are illustrated in table 4.2.

Table 4.2. Root Mean Squared Error (RMSE) of the test set, averaged for 20 runs, for the Incremental ELM (I-ELM), Convex I-ELM, Enhanced I-ELM and the proposed Enhanced Convex I-ELM. For each dataset the lowest RMSE is shown in boldface.

Dataset	I-ELM	Convex I-ELM	Enhanced I-ELM	Enhanced Convex I-ELM
Pokerhand	0.0859	0.0858	0.0846	**0.0841**
Year Prediction MSD	0.0294	0.0187	0.0176	**0.0169**
3D Road Network	0.1283	0.1278	0.1277	**0.1261**
Protein Structure PSP	0.1420	0.1398	0.1382	**0.1375**
2D planes	0.1006	0.0774	0.0607	**0.0504**
Friedman	0.0871	0.0821	0.0814	**0.0718**
Census (House8L)	0.0848	0.0836	0.0805	**0.0784**
California Housing	0.1517	0.1482	0.1436	**0.1421**
Elevators	0.0553	0.0473	0.0407	**0.0398**
Ailerons	0.0720	0.0466	0.0459	**0.0451**
Delta Ailerons	0.0407	0.0398	0.0397	**0.0393**
Puma Dynamic	0.1823	0.1783	0.1771	**0.1710**
Bank	0.0697	0.0717	0.0470	**0.0465**
Kinematics	0.1414	0.1324	0.1242	**0.1123**
Delta elevators	0.0539	0.0530	0.0531	**0.0528**
White Wine	0.1298	0.1273	0.1265	**0.1250**
Abalone	0.0853	0.0801	0.0794	**0.0775**
Red Wine quality	0.1319	0.1303	0.1301	**0.1290**

From table 4.2 the prediction performance comparison results show that the Enhanced versions outperform their I-ELM and CI-ELM counterparts. The lowest error is observed from the proposed Enhanced Convex I-ELM, which precedes the other algorithms in all the benchmark datasets and delivers the best results based on the prediction accuracy. An order of accuracy is also evident, which from the least accurate to the most accurate algorithm is I-ELM, CI-ELM, EI-ELM and ECI-ELM. The comparison of the standard deviations of the testing RMSE for each dataset is presented in table 4.3.

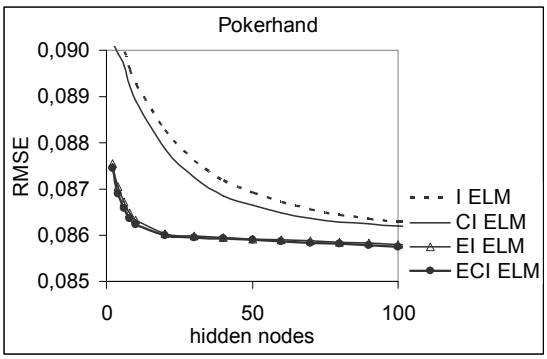
Table 4.3. Standard deviations of the errors between the Incremental ELM (I-ELM), Convex I-ELM, Enhanced I-ELM and the Enhanced Convex I-ELM.

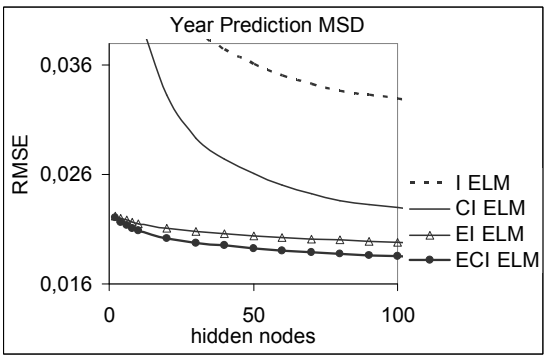
Dataset	I-ELM	Convex I-ELM	Enhanced I-ELM	Enhanced Convex I-ELM
Pokerhand	0.00023	0.00021	0.00021	0.00022
Year Prediction MSD	0.01575	0.00020	0.00031	0.00011
3D Road Network	0.00036	0.00044	0.00033	0.00033
Protein Structure PSP	0.00105	0.00042	0.00044	0.00042
2D planes	0.00490	0.00152	0.00322	0.00072
Friedman	0.00291	0.00065	0.00141	0.00102
Census (House8L)	0.00202	0.00215	0.00210	0.00234
California Housing	0.00301	0.00272	0.00193	0.00205
Elevators	0.00181	0.00223	0.00077	0.00051
Ailerons	0.00685	0.00096	0.00102	0.00092
Delta Ailerons	0.00210	0.00139	0.00145	0.00137
Puma Dynamic	0.00497	0.00377	0.00380	0.00540
Bank	0.00412	0.00494	0.00265	0.00227
Kinematics	0.00286	0.00274	0.00207	0.00267
Delta elevators	0.00079	0.00074	0.00070	0.00074
White Wine	0.00403	0.00367	0.00338	0.00322
Abalone	0.00279	0.00187	0.00203	0.00165
Red Wine quality	0.00626	0.00593	0.00620	0.00609

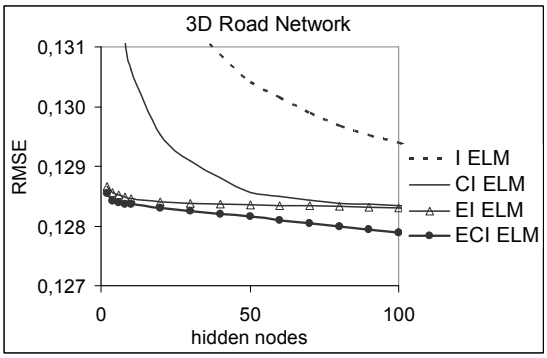
From table 4.3 it can be observed that the standard deviations for Convex I-ELM, Enhanced I-ELM and the Enhanced Convex I-ELM are similar.

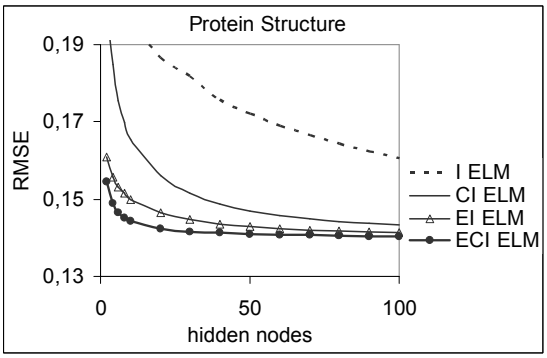
4.5.3 Convergence analysis

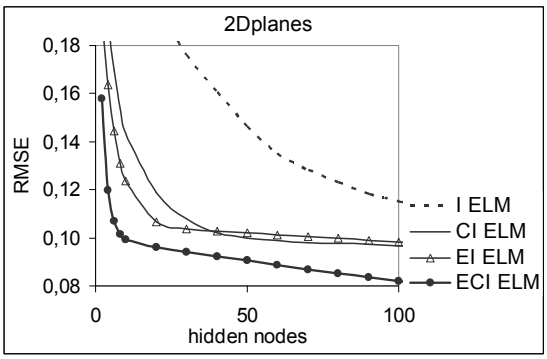
One key ingredient in the effectiveness of the proposed ECI-ELM algorithm lays in its convergence. Hence, in this section we study and compare the convergence rate and speed for each one of the incremental algorithms. Fig. 4.3 compares I-ELM, CI-ELM, EI-ELM and the proposed ECI-ELM with respect to their convergence and provides snapshots that visually show the intrinsic capability of each algorithm to minimize the error, as averaged over the 20 runs, showing also how stable the results are. For all the benchmark datasets the RMSE is measured as a function of the number of hidden neurons, when those are incrementally added in the network structure. In all the cases in fig. 4.3 we can observe that ECI-ELM has the lowest error rate curve and converges much faster than the other algorithms in all of the datasets. Hence, we can safely conclude that ECI-ELM quickly reaches a satisfactory solution and its power lays in both the more compact architecture and the higher convergence rate. Fig. 4.3 also confirms in a visual way that the order of accuracy from the least accurate to the most accurate algorithm is I-ELM, CI-ELM, EI-ELM and ECI-ELM.

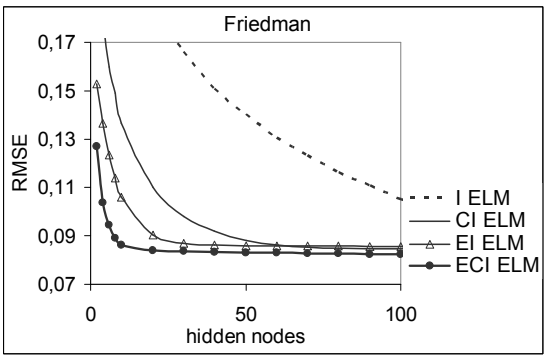

Pokerhand
RMSE
0,090
0,089
0,088
0,087
0,086
0,085
0
50
100
hidden nodes
I ELM
CI ELM
EI ELM
ECI ELM

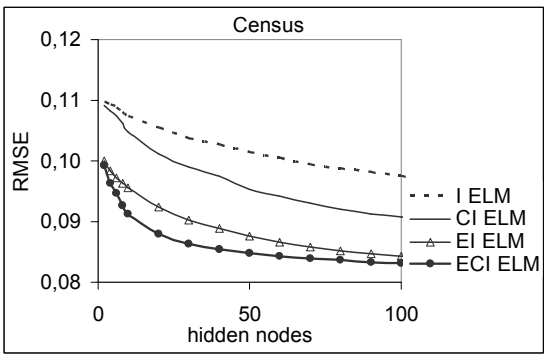

Year Prediction MSD
RMSE
0,036
0,026
0,016
0
50
100
hidden nodes
I ELM
CI ELM
EI ELM
ECI ELM

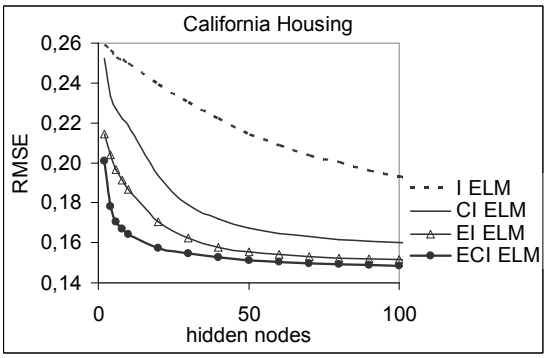

3D Road Network
RMSE
0,131
0,130
0,129
0,128
0,127
0
50
100
hidden nodes
I ELM
CI ELM
EI ELM
ECI ELM

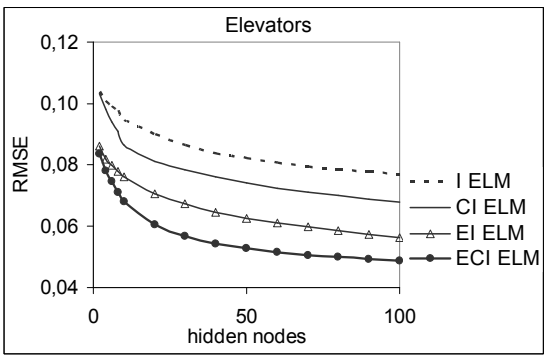

Protein Structure
RMSE
0,19
0,17
0,15
0,13
0
50
100
hidden nodes
I ELM
CI ELM
EI ELM
ECI ELM

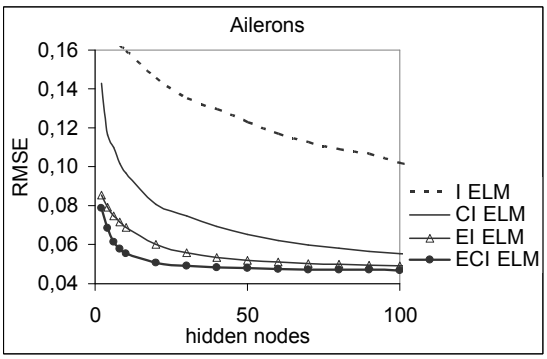

2Dplanes
RMSE
0,18
0,16
0,14
0,12
0,10
0,08
0
50
100
hidden nodes
I ELM
CI ELM
EI ELM
ECI ELM


Friedman
RMSE
0,17
0,15
0,13
0,11
0,09
0,07
0
50
100
hidden nodes
I ELM
CI ELM
EI ELM
ECI ELM


Census
RMSE
0,12
0,11
0,10
0,09
0,08
0
50
100
hidden nodes
I ELM
CI ELM
EI ELM
ECI ELM


California Housing
RMSE
0,26
0,24
0,22
0,20
0,18
0,16
0,14
0
50
100
hidden nodes
I ELM
CI ELM
EI ELM
ECI ELM


Elevators
RMSE
0,12
0,10
0,08
0,06
0,04
0
50
100
hidden nodes
I ELM
CI ELM
EI ELM
ECI ELM


Ailerons
RMSE
0,16
0,14
0,12
0,10
0,08
0,06
0,04
0
50
100
hidden nodes
I ELM
CI ELM
EI ELM
ECI ELM

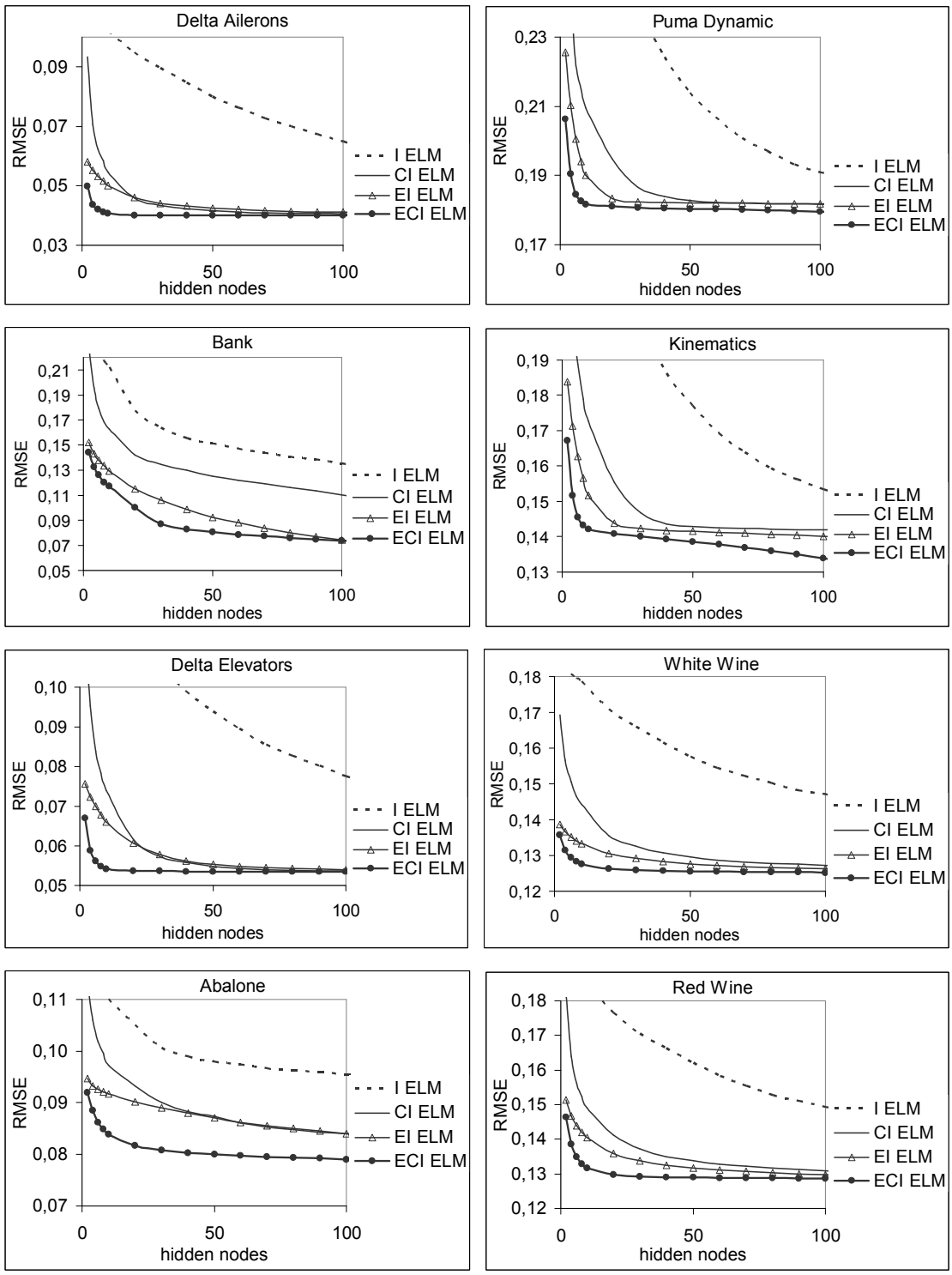


Figure 4.3. Convergence comparisons for every dataset present RMSE curves of the algorithms, as a function of the number of hidden ELM nodes (neurons). For clarity reasons only the first 100 neurons are shown, in order to make the curves more distinct. In all the datasets the lowest curve always corresponds to ECI-ELM algorithm.

The original I-ELM [11] incrementally adds random hidden neurons to the existing network one-by-one and due to the randomness and the freezing of the existing weights this process usually requires more hidden neurons that the conventional neural networks, a fact that is clearly illustrated in fig 4.3. That is, I-ELM suffers from low convergence and larger network sizes that are required in order to achieve

matched performance. To improve the network convergence the Convex I-ELM [13], by using eq. 4.5, readjusts all the linear output weights of the other existing hidden neurons, after each incremental neuron addition and computation of its new weight. This is also demonstrated in fig. 4.3 and that is why the CI-ELM curves in fig. 4.3 are lower than those of the I-ELM. The Enhanced I-ELM (EI-ELM) [12] improves the I-ELM compactness by first generating K random hidden neurons, at every neuron addition step, and then by selecting from these K trials only the one neuron that produces the minimum training error. In this way, the EI-ELM hidden layer requires much less neurons than I-ELM and CI-ELM in order to achieve the same performance and that is why the EI-ELM curves are lower that those of CI-ELM and I-ELM. While EI-ELM freezes the output weights of the existing hidden neurons when a new hidden neuron is added, the proposed ECI-ELM, like its CI-ELM counterpart, readjusts all the output weights of the other existing hidden neurons, after selecting from the K temporary generated random neurons the one neuron that produces the minimum residual error, the addition of this new neuron to the hidden layer and the computation of its weight. That is why all the ECI-ELM curves are the lowest for all the datasets in fig. 4.3.

4.5.4 Data Parallelism performance metrics

The master/worker parallel ELM implementations for the parallel system of distributed memory workstations are written in C using the Message Passing Interface (MPI) library [19]. In the parallel system each processor has a clock speed of 2.5 GHz, memory 2GB and Linux operating system. The machines are interconnected with 1000 Mbps Ethernet. The training times are determined for 1 up to P=50 processors.

In essence, parallel programs are governed by laws/metrics that are trying to explain and assert the potential performance of a parallel application [3][4]. The two most basic such lows are the low of Amdahl (Amdahl's speedup) and the low of Gustafson (Gustafson's isoefficiency). Usually given a problem D and P processors we study four kinds of scalability measurements: speedup, efficiency and scaleup.

4.5.4.1 Speedup and efficiency

Speedup finds out sequential bottlenecks by fixing the size of the problem D and increasing the number P of processors (Amdahl). Speedup measures the ratio between the sequential execution time $T(1, \mathrm{D})$ and the parallel execution time $T(P, \mathrm{D})$ and is given by:

$$Speedup(P) = T(1, \mathrm{D})/T(P, \mathrm{D})$$

Efficiency measures the usage of the computational resources. This can sometimes provide a more convenient measure of parallel algorithm quality. Efficiency computes the fraction of time between performance and the resources used to achieve that performance given by:

$$Efficiency(P) = T(1, \mathrm{D})/(P \times T(P, \mathrm{D})) = Speedup(P) \ / \ P$$

An ideal parallel implementation exhibits linear speedup, meaning that with p processors the speedup is p. In practice, with an increasing number of processors the communication cost increases and the related latencies render a linear speedup very hard to accomplish, and the results are sub-linear. On the other hand, if the communication cost is kept low as it is the case for Incremental ELMs the speedup is

progressively improved on increasing the dataset size. The speedup ratio is illustrated in fig. 4.4.

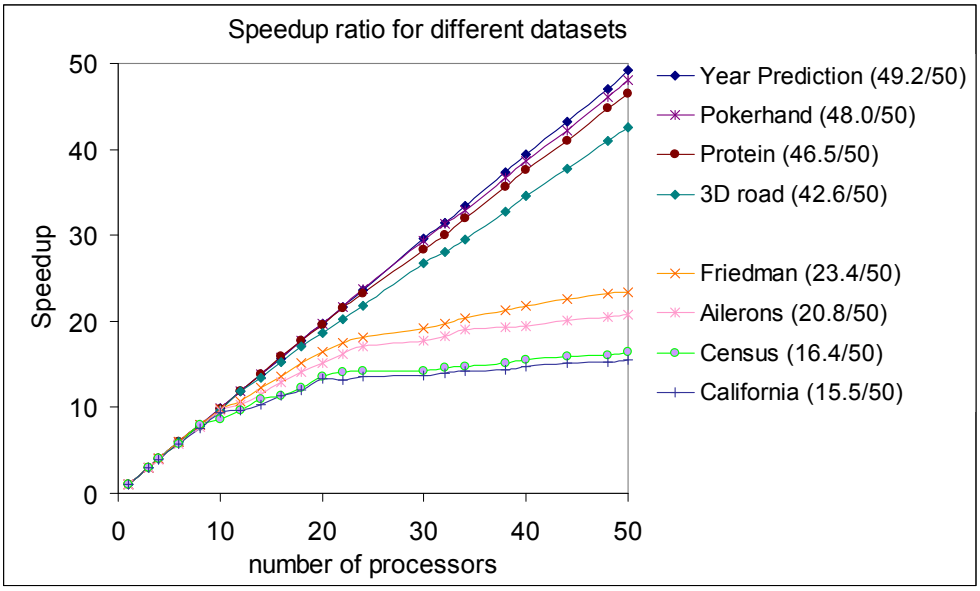


Figure 4.4. Speedup ratio in the data-parallel training of Enhanced Convex I-ELM. For each dataset the corresponding efficiency value for P=50 processors is indicated in the parenthesis. A noticeable almost linear speedup is observed for the larger size datasets.

For moderate size datasets like Friedman, Ailerons, Census and California in fig. 4.4 a sub-linear speedup is noticeable. However this situation is well improved on increasing the data size. For the large in size datasets like Year Prediction, Pokerhand, Protein, and 3D road there are significant improvements of speedup. Also one can observe that for the larger size datasets like Year Prediction, Pokerhand, Protein, and 3D road the efficiencies range from 0.85 (42.6/50) to 0.98 (49.2/50) and they approach the ideal case as many of them are very close to one. It can be seen from fig. 4.4 that the larger a data set is, the higher the speedup ratio is. The highest speedup ratio occurs for the Year Prediction dataset with an efficiency value of 49.2/50.

Data parallelism of the proposed Enhanced and Convex I-ELM is promising in a distributed memory multiprocessor system and becomes more efficient as the volume of the data increases.

4.5.4.2 Scaleup (isoefficiency)

Scaleup measures the scalability with respect to efficiency. The scalability of a parallel program reflects its capacity in making use of available resources effectively. A parallel program is considered scalable if its efficiency is maintained when we increase proportionally the number of processors and the size of the problem. Hence, scaleup reveals how many p processors are required to solve a problem P-times larger than the problem D by keeping the performance steady (Gustafson's isoefficiency). Scaleup is expressed by the ratio:

$Scaleup(p) = T(1, \mathrm{D}) \; / \; T(P, P\mathrm{D})$

In an ideal scaleup curve a constant response time must be maintained as both the size of the dataset and the number of processors and disks in the parallel system grow proportionally. However the scalability of the parallel solution may be limited owning to larger communication costs and idle times. In the performed scaleup experiments

the size of each dataset D is increased in direct proportion to the number of computers in the system.

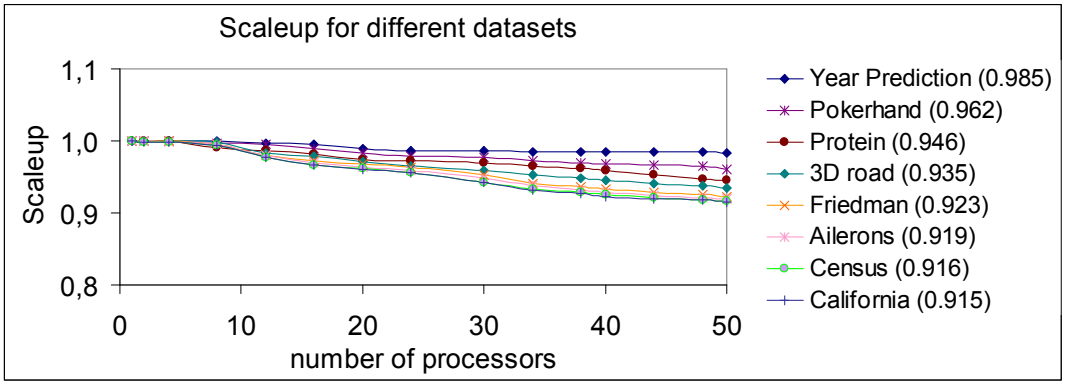


Figure 4.5. Scaleup results. For each dataset the corresponding scaleup value for P=50 processors is indicated in the parenthesis. Almost linear scaleup is observed for the larger size datasets.

Fig. 4.5 shows the scaleup results. For all the datasets the scaleup measurements range from 0.98 to 0.92 while for the larger datasets the scaleup values are very close to 1, the ideal case. From the scaleup values in the parenthesis that correspond to P=50 processors it can be seen that the parallel ECI-ELM algorithm scales well. The previous performance metrics demonstrate that the proposed Enhanced Convex I-ELM offers almost linear speedup for the larger datasets and as the scaleup metric reveal it can scales up on the available resources.

The suggestion of ECI-ELM algorithm as a promising candidate for big data regression can be further reinforced by the following observations. Notice first of all that practically in the largest scaleup experiment, the pokerhand dataset, 17.5 Gigabyte of data were easily processed from 50 processors. Furthermore, the ECI-ELM algorithm needs one data scan per iteration $L = L + 1$, and trial k in the K-list of, in order to construct the one activation vector $\mathbf{g}_L = [g_L(\mathbf{x}_1) \ \ldots \ g_L(\mathbf{x}_N)]^T$ of size N, that needs. Thus, owning to the incremental nature of ECI-ELM a major computational advantage comes from the processing requirements and the fact that only one random neuron is added incrementally per iteration step. This means that the learning process requires only one particular activation vector $\mathbf{g}_L$ to be involved in the calculations at a time. Another advantage lays in the memory needs of ECI-ELM since, practically, beyond the hidden neuron set $\{\mathbf{a}_l, b_l\}_{l=1}^{L}$, the algorithm at each iteration $L = L + 1$ requires in main memory few main vectors, namely the activation vector $\mathbf{g}_L$, the weights vector $\mathbf{w} = [w_1 \ldots w_L]^T$ and the residual vector $\mathbf{e}_{L-1} = [e_{L-1}(\mathbf{x}_1) \ \ldots \ e_{L-1}(\mathbf{x}_N)]^T$. In the data parallel experiments the activation vector and the residual vector were partitioned across the processors. Note also that there are no restrictions for the training dataset, which can either be loaded in main memory for fast accessing, when the available resources permitted, or might remain in fast secondary memory storage, in which case the data files can be directly scanned, in chunks, at every calculation step.

4.6 Summary

For solving large regression problems we explore scalable Incremental Extreme Learning Machine (I-ELM) algorithms that can be straightforwardly trained in distributed memory multiprocessors by using data parallel mappings. We introduce the Enhanced Convex Incremental ELM (ECI-ELM) algorithm which is a combination

of the Enhanced I-ELM and the Convex I-ELM. In Enhanced I-ELM, the newly added hidden neurons per iteration step are selected from a candidate pool, and only appropriate neurons are added into the network. Consequently, the Enhanced I-ELM network becomes more compact than I-ELM, by getting rid of unimportant neurons. In the Convex I-ELM the convergence rate of I-ELM is further improved by recalculating the output weights of the existing nodes based on the Barron's convex optimization method. In the presented ECI-ELM the advantages of the two previous algorithms, either Enhanced or Convex, are fused in one. The main findings of the simulations are that the proposed Enhanced Convex Incremental ELM (ECI-ELM) is a fast algorithm for accurate regression, with high convergence rate, and it is also fully parallelizable.

Specifically, the measurements of the accuracy and performance on various benchmark datasets demonstrate that the ECI-ELM is the most accurate among the other existing incremental ELM versions. From the least accurate to the most accurate algorithm the ranking is I-ELM, CI-ELM, EI-ELM and ECI-ELM. This same ranking appears in the detailed experiments we conduct for the convergence analysis, where comparisons with all the existing incremental ELM versions have also been made. The RMSE is measured as a function of the number of hidden neurons in order to show that ECI-ELM has the lowest error rate curve and converges much faster than the other algorithms in all of the datasets.

We concentrate on the incremental ELM versions because they are based solely on simple direct summations which by their part can be most efficiently mapped on parallel environments and can potentially scale up to several processors. To that end, extensive experimental simulations on the data parallel design of the ECI-ELM reveal that, while for the smaller datasets the overhead of the parallel algorithms overshadows the parallelisation benefits, for the larger datasets, which is the focus of the current study, the parallel training of ECI-ELM is able to demonstrate almost linear speedups. By using 50 processors in the larger datasets the highest speedup ratio that we measure has an efficiency value of 0.98. In addition, the scaleup parallel simulations for all the datasets reveal scaleup values that range from 0.98 to 0.92, from which we can safely conclude that scaling up of the algorithm can be assured on increasing the available resources.

References for chapter 4

[1] Chen M., Mao S. and Liu Y. (2014) Big Data: A Survey. Mobile Networks and Applications, 19, 171–209.

[2] Wu X., Zhu X., Wu G.Q. and Ding W. (2014) Data mining with big data. IEEE Transactions on Knowledge and Data Engineering, 26(1), 97–107.

[3] Wilkinson B. and Allen M. (2000) Parallel Programming: Techniques and Applications Using Networked Workstations and Parallel Computers. Prentice Hall.

[4] Grama A., Gupta A., Karypis G. and Kumar V. (2003) Introduction to Parallel Computing. Pearson Education Limited.

[5] Upadhyaya S.R. (2013) Parallel approaches to machine learning-A comprehensive survey. Journal of Parallel Distributed Computing, 73, 284–292.

[6] Halevy A., Norvig P. and Pereira F. (2009) The Unreasonable Effectiveness of Data. IEEE Intelligent Systems, 24(2), 8–12.

[7] Huang G.-B., Zhu Q.-Y. and Siew C.-K. (2006) Extreme learning machine: theory and applications. Neurocomputing, 70, 489–501.

[8] Huang G.-B., Wang D.H. and Lan Y. (2011) Extreme learning machines: a survey. International Journal of Machine Learning & Cybernetics 2, 107–122.

[9] Cambria E., and Huang G.-B. (2013) Extreme Learning Machines. IEEE Intelligent Systems, 30–59.

[10] Huang G., Huang G.B., Song S. and You K. (2015) Trends in extreme learning machines: A review. Neural Networks, 61, 32–48.

[11] Huang G.B., Chen L. and Siew C.-K. (2006) Universal approximation using incremental constructive feedforward networks with random hidden nodes. IEEE Transactions on Neural Networks, 17(4), 879–892.

[12] Huang G.B. and Chen L. (2008) Enhanced random search based incremental extreme learning machine. Neurocomputing, 71, 3460–3468.

[13] Huang G.B. and Chen L. (2007) Convex incremental extreme learning machine. Neurocomputing, 70, 3056–3062.

[14] He Q., Shang T., Zhuang F. and Shi Z. (2013) Parallel extreme learning machine for regression based on MapReduce. Neurocomputing 102, 52–58.

[15] Xin J., Wang Z., Chen C., Ding L., Wang G. and Zhao Y. (2014) ELM*: distributed extreme learning machine with MapReduce. World Wide Web, 17(5), 1189-1204.

[16] van Heeswijk M., Miche Y., Oja E. and Lendasse A. (2011) GPU-accelerated and parallelized ELM ensembles for large-scale regression. Neurocomputing, 74. 2430–2437.

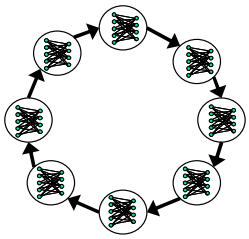
[17] van Heeswijk M., Miche Y., Oja E. and Lendasse A. (2010) Solving large regression problems using an ensemble of GPU-accelerated ELMs. In: Proceedings of the 18th European Symposium on Artificial Neural Networks (ESANN), Bruges, Belgium, April 2010, pp. 309–314.

[18] Wang X.-L., Chen Y.-Y., Zhao H. and Lu B.-L. (2014) Parallelized extreme learning machine ensemble based on min–max modular network. Neurocomputing, 128, 31–41.

[19] Pacheco P. (1997) Parallel Programming with MPI. Morgan Kaufmann, San Francisco, CA.

5 Ring Pipeline parallel algorithms for Scalable reduced Probabilistic Neural Networks

This chapter demonstrates that a parallel parsimonious Probabilistic Neural Network (PNN) can be efficiently designed for fast and scalable training by using the combined force of the proposed leave-one-out kernel averaged gradient descent algorithm in conjunction with Subtractive Clustering algorithm and Expectation Maximization algorithm in a ring pipeline parallel scheme suitable for distributed memory machines. Pipeline is known for minimizing the communication delays which otherwise hinder the classical parallel algorithms from scaling up. Although pipeline can produce linear speedups using thousands of processors few training algorithms can be entirely implemented in a pipeline. The essence of the proposed solution relies on a ring pipeline architecture where each processor holds a portion of data and a portion of neurons.



For scalable PNN training we map the proposed leave-one-out gradient descent algorithm together with Subtractive Clustering and Expectation Maximization in the proposed hybrid ring pipeline parallel architecture. First leave-one-out kernel gradient descent extracts suitable kernel bandwidth parameters from every class of the data. Then Subtractive Clustering uses these bandwidth parameters to automatically select exemplar centers for the PNN kernels of every class and Expectation Maximization refines these centers and computes their mixing coefficients and widths. Unlike many existing methods, in our scheme no user-defined parameters are required and the PNN model is created automatically. In addition the produced parsimonious PNN model is compared with PNNs produced from other *k*-center clustering algorithms. Experimental simulations on classification accuracy performance on various benchmark datasets and related comparisons are also demonstrated.

5.1 Introduction

The continued growing of many applications that collect and store their data across distributed processors interconnected with each other, give rise to the need of machine learning algorithms that work in parallel in order to analyse data or to perform data mining tasks. Parallel processing methods [1] [2] [3] [4] [5] can in principal cop with common scalability issues arise in large data collections when one tries to discover and exploit their hidden patterns. The Probabilistic Neural Network (PNN) [6][7][8][9][10]

[11][12][13][14][15][16][17][18][19][20][21][22][23][24][25] is well established for classification tasks and applications that also need confidence levels. PNN derives patterns directly from the data and represent them in the form of simple well understood Bayesian models. Such Bayesian classifier methods can, in a strict mathematical sense, work under uncertainty and produce confidence levels together with their predictions. In this chapter we present new scalable parallel training algorithms for parsimonious Probabilistic Neural Networks in a ring pipeline architecture which is suitable for distributed memory machines.

The common problem of Probabilistic Neural Network is that the number of hidden neurons in the PNN pattern layer is usually of the order of the whole dataset size. Thus the PNN is rather slow and quite demanding in memory and CPU resources and for large scale systems the PNN usage is hindered. This encourages more research into the scalability of these techniques. As a result, during the last years, various works have been presented for mapping a Probabilistic Neural Network operation in parallel processing systems. Existing parallel solutions include the parallel PNN in Clusters of workstations [26] using MPI, in Grid mining with Map/Reduce [27], and in Graphic Processing Units [28]. These studies mainly focus on splitting the data-neuron matrix to speed up the slow execution times due to PNNs quadratic computational complexity. In this way they demonstrate that the PNN operation can be efficiently parallelized.

However, parallelism is only one path towards speeding up the PNN. When all the N training examples are used as kernel centers in the hidden PNN pattern layer, the typical PNN operation has O(N) cost to classify an unknown pattern. This cost is quite large. For large scale applications the increased number of pattern layer neurons in the classical PNN renders them impractical for use even in a parallelized algorithm running in multiple processors. Parsimonious PNNs are more desirable. The smaller is the better. The computational complexity of PNN can be reduced only by selecting the most important pattern neurons of PNN and try to place them optimally.

Hence existing neuron selection techniques try to extract a list of k-centers which are close to any given training example. For the PNN structure several clustering methods like learning vector quantization, k-means clustering, agglomerative hierarchical clustering, Orthogonal Forward Selection, Gaussian ARTMAP clustering, K-Medoids clustering, Subtractive Clustering are well documented (see section 2 for more details). These algorithms require user-defined adjustable parameters, specifically in most cases the number k of the centers. From the aforementioned algorithms Subtractive Clustering is the one that can be efficiently implemented in a scalable parallel ring pipeline. The problem is that Subtractive Clustering also requires user-defined parameters, like the bandwidth, and we have recently found [29] that through a gradient descent learning procedure of the leave-one-out kernel averaged regression function [29] such parameters can be estimated automatically. This process can thus be applied for PNN construction and training without human intervention.

Hence, here we use the recently proposed leave-one-out kernel averaged gradient descent Subtractive Clustering (KG-SC) [29] for selecting the most representative kernel centers within classes and manage to automatically find their number as well. After that, we use Expectation Maximization algorithm [30] for refining the resulting PNN learning parameters and finalizing the training. In addition, efficient ring pipeline parallel mappings are presented in the next sections for the entire learning scheme.

The essence of the proposed parallel solution for the scalable parsimonious PNN is based on a ring pipeline parallel architecture which is known for not suffering from common scalability problems, communication delays and bottleneck effects. By mapping on the ring pipeline the recently proposed leave-one-out kernel averaged gradient descent together with subtractive clustering and Expectation-Maximization we finally end up with a compact and scalable parallel training scheme that automatically determines the PNN structure and optimizes all the parameters efficiently and without any user input or intervention. The proposed solution is extremely parallelizable and produces quite small PNNs in the same time.

The main contributions of this chapter can be summed up as follows: 1) It employs the recently proposed KG-SC algorithm [29] for automatic selection of the most important PNN centers within the training data. 2) It presents this combination for Probabilistic Neural Network construction and compare it side-by-side with many existing samplers like Orthogonal Forward Selection for PNN [8], K-Medoids for PNN [12], Farthest Point Clustering [33] and Affinity Propagation [34]. 3) It presents the ring pipeline parallel algorithms for the leave-one-out kernel averaged gradient descent, subtractive clustering, Expectation step and Maximization step for the parsimonious PNN training. Hence the same ring pipeline scheme is used for the parallel mappings of all these algorithms on distributed memory machines. The presented ring pipeline parallel version of Expectation-Maximization is different from existing ones in the literature since it works without maintaining into main memory any large matrix for the Expectation step weights (hence it is scalable for any large size of training dataset) and it has both Expectation as well as Maximization steps consisted of the same parts of the sequence {Send-Compute-Receive} that provides efficient overlapping of computation delays with communication delays.

We explicitly employ the ring pipeline parallel architecture. In contrast with master/worker or simple pipeline architectures where either data or neurons are partitioned across the processors, in the ring pipeline both neurons and data can be partitioned. In the ring pipeline the message passing uses only point-to-point communication messages where each processor node receives a message only from its previous and sends a message only to its next. Thus the communication delays between the processors are minimized. A similar ring pipeline of processing elements with a broadcast mechanism (messages send to all processors) is a general architecture [1] for both clusters of workstations [4] as well as neuro-chips.

The rest of the chapter is structured as follows. Section 5.2 provides related works for parallelising Probabilistic Neural Networks and existing training methods for neuron reduction. Section 5.3 introduces the proposed PNN training scheme. Section 5.4 presents the basics of the PNN Parallelization mappings. Section 5.5 gives the proposed data-neuron PNN parallel training in a Ring pipeline. It also presents the ring pipeline mini-batch kernel averaged gradient descent and the ring pipeline parallel subtractive clustering algorithms. Section 5.6 provides details for the data distributed Ring parallel Expectation Maximization. Section 5.7 presents experimental results and comparisons and Section 5.8 concludes the chapter.

5.2 Related Work

5.2.1 Parallelising Probabilistic Neural Networks

The Parallel operation of Probabilistic Neural Networks so far has been implemented in Beowulf Clusters [26], in Grid mining with Map/Reduce [27], and in Graphic Processing Units [28]. These works focus on splitting the pattern layer neurons of PNN to many processors which then work either in a neuron parallel master-worker or in a neuron parallel pipeline fashion. The last architecture does not suffer from bottleneck effects and can guarantee linear speedup that can scale well. Since the previous works use all the training instances in the pattern layer the parallel training of PNN for parsimonious network models has not been considered yet. Differently from the existing works we study how to parallelize the reduction of pattern neurons for efficient structure determination and training of PNN.

5.2.2 Reducing the PNN pattern neurons

Reducing the PNN pattern neurons can increase the speed and generalization capability. Parsimonious models are better. Various approaches have been proposed for decreasing the PNN pattern neurons and to find cluster centers, like learning vector quantization in [35], k-means clustering [36], restricted Coulomb energy clustering [7], agglomerative hierarchical clustering [37], the Dynamic Decay Adjustment (DDA) algorithm [38], Orthogonal Forward Selection (OFS) [8], Gaussian ARTMAP clustering in [39] with gap-based estimation, global k-means clustering in [40] and K-Medoids clustering algorithm in [12]. Genetic algorithms are used in [14] and [15] for reducing the number of hidden pattern neurons in the PNN model and for optimizing the smoothing parameters. The studies in [18] and [41] examine the application of plain Subtractive Clustering, with user-defined parameters, for constructing a reduced PNN [18] and a reduced RBPNN [41] respectively. The aforementioned methods require some user-defined adjustable parameters, specifically in most cases the number k of the kernel centers, or a kernel bandwidth which are usually found by trial-and-error. Since for large datasets the trials are very costly we should avoid using any free-parameter and the whole process must be automatic.

In this work we propose to use the KG-SC algorithm [29] that can automatically determine both the centers and their number. This algorithm can be also implemented in a ring pipeline. We further combine the reduction of PNN pattern neurons with Expectation Maximization (EM) algorithm which refines the final positions of the centers as well as the mixing coefficients and the covariances of the kernels. Expectation Maximization has been used previously for PNN training [30] [31] [32] and we have found that it can be also efficiently implemented in the proposed ring pipeline parallel architecture.

We also make experimental comparisons with Orthogonal Forward Selection for PNN [8], K-Medoids clustering for PNN [12], Farthest Point Clustering [33] and Affinity Propagation clustering [34]. Farthest Point Clustering is an efficient algorithm for the *k*-centers problem and is considered fast. Affinity Propagation clustering algorithm finds the most important *k*-centers without their *k* number to be given in advance and is considered the current state-of-the-art.

5.3 Proposed PNN training

5.3.1 The basics of PNN Architecture

A Probabilistic Neural Network (PNN) [6] is a supervised neural network classifier that is based on the Bayes rule which minimizes the potential risk. In the next mathematical formulas an uppercase bold letter will symbolize a matrix, while a lowercase bold letter will symbolize a vector, and an italic letter will denote a scalar.

With a given training dataset $\{\mathbf{x}_n, y_n\}_{n=1}^{N}$, where each label y_n represents the class of $\mathbf{x}_n$, the basic idea of PNN is to learn a probability distribution function for each class. The number of classes is K and each class ω has N_ω training examples (ω=1,…,K). From Bayes theorem the posterior probability $p(\omega \mid \mathbf{x})$ that a given $\mathbf{x}$ belongs to class ω is:

$$p(\omega \mid \mathbf{x}) = p(\mathbf{x} \mid \omega) \cdot p(\omega) \;/\; p(\mathbf{x}) \tag{5.1}$$

where $p(\mathbf{x} \mid \omega)$ is the class conditional probability that ω hypothesis can be satisfied by $\mathbf{x}$. The normalization factor $p(\mathbf{x})$ is the prior probability of $\mathbf{x}$ occurrence and is given by summing the contributions from all K classes as:

$$p(\mathbf{x}) = \sum_{\omega}^{K} p(\mathbf{x} \mid \omega) \cdot p(\omega) \tag{5.2}$$

The prior probabilities $p(\omega)$ for each class ω are usually constant. Thus only the class conditional probabilities $p(\mathbf{x} \mid \omega)$ must be found, namely the true probability density functions for each class. These are computed via non-parametric density estimation using the summations of the Parzen kernels which are implemented via multivariate Gaussian probability density functions. This defines the optimum Bayes rule.

The architecture of PNN illustrated in fig. 5.1 has four layers, namely input, pattern, summation and output. The PNN pattern layer is where pattern neurons are divided into class groups. Each class ω has N_ω training data, M_ω pattern neurons and a single summation neuron in the summation layer.

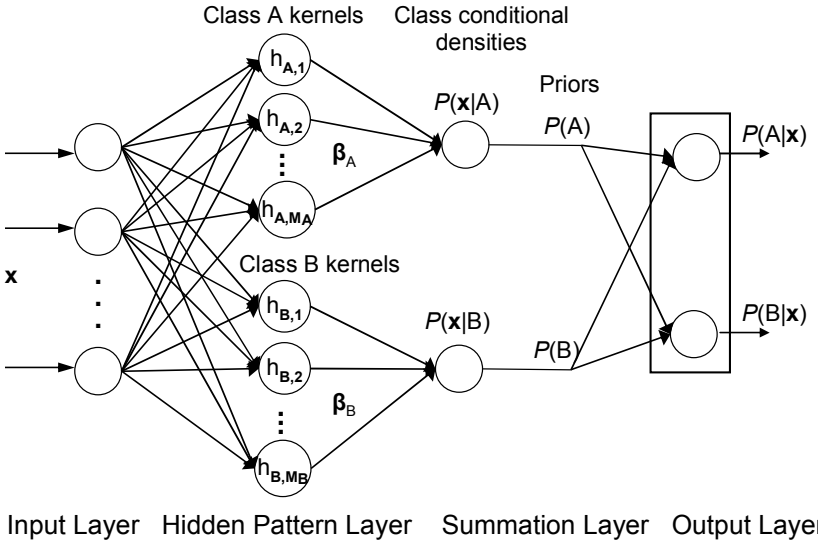


Figure 5.1. The Probabilistic Neural Network architecture with Bayesian probability terms for a two class problem. The outputs provide the confidences for each class.

For an unknown sample $\mathbf{x}$ the class conditional probability $p(\mathbf{x}\,|\,\omega)$ for a class ω is given by its summation neuron $f_\omega(\mathbf{x})$ as:

$$f_\omega(\mathbf{x}) = \sum_m^{M_\omega} \beta_{m,\omega} \cdot h(\mathbf{x}, \mathbf{c}_{m,\omega}, \mathbf{R}_\omega) \tag{5.3}$$

where m is the kernel index, ω is the class index, $\beta_{m,\omega}$ is the positive mixing coefficient of m^{th} Parzen kernel in ω^{th} class with $\sum_m^{M_\omega} \beta_{m,\omega} = 1$, $\mathbf{R}_\omega$ is the covariance matrix in ω^{th} class, $\mathbf{c}_{m,\omega}$ is the m^{th} center of the Parzen kernel in ω^{th} class, and $h(\mathbf{x}, \mathbf{c}_{m,\omega}, \mathbf{R}_\omega)$ is given by a multivariate Gaussian probability density function:

$$h(\mathbf{x}, \mathbf{c}, \mathbf{R}) = \frac{1}{(2\pi)^{d/2}\sqrt{|R|}} \exp\Big(-\frac{1}{2}(\mathbf{x}-\mathbf{c})^{\mathrm{T}}\,\mathbf{R}^{-1}(\mathbf{x}-\mathbf{c})\Big) \tag{5.4}$$

If all the training data are used as centers for the kernels then $M_\omega = N_\omega$. The summation layer computes the class conditionals $p(\mathbf{x}\,|\,\omega) = f_\omega(\mathbf{x})$ and finally the output layer classifies the unknown $\mathbf{x}$ in the class that has maximum output. As usual in PNNs we assume that the covariance matrices are diagonal. The class priors $p(\omega)$ are problem dependent and it is a common tactic to assume that every class is equiprobable. The posterior probability $p(\omega\,|\,\mathbf{x})$ for a class ω, gives the confidence levels for this class. A PNN with a global covariance matrix for all classes is called homoscedastic, while if kernels are allowed to have different covariances per class then this PNN is called heteroscedastic. We consider the heteroscedastic PNN case.

5.3.2 Proposed Kernel Gradient Subtractive Clustering

This section presents the recently proposed Kernel Gradient Subtractive Clustering (KG-SC) algorithm [29] that automatically selects most representative centers inside each class. Before the descriptions of the parallel steps an introduction for the sequential algorithm is presented next.

5.3.2.1 Traditional Subtractive clustering

The traditional Subtractive clustering algorithm [42] [43] [44] selects a set of exemplar centers, which are data points themselves, from the most representative ones. The algorithm computes the potentials of the points based on their density and then gradually subtracts points one by one. Every point $\mathbf{x}_i$ is a candidate center and is ranked based on a potential $P(i)$ defined as a sum of Gaussian kernels over all N data points as:

$$P(i) = \sum_{n=1}^{N} exp(-a\,||\,\mathbf{x}_i - \mathbf{x}_n\,||^2) \tag{5.5}$$

where $a = (2/\sigma_a)^2$ and σ_a represents a bandwidth (neighbourhood radius).

After finding all the potentials the algorithm repeatedly executes the following cycle:

Step 1) Find data point $\mathbf{x}^*$ (center) with the highest potential value P^*

Step 2) Revise the potential of all other points using:

$$P(i) = P(i) - P^* \exp(-b\,||\,\mathbf{x}_i - \mathbf{x}^*\,||^2) \tag{5.6}$$

In step 2 the highest potential $P^*$of the selected point $\mathbf{x}^*$ will substantially affect all the potentials of the points near by. The update of the potentials have $b=(2/\sigma_b)^2$ where σ_b defines another bandwidth. The σ_b value is taken to be a factor of σ_a. This factor must

be larger than 1 and usually $\sigma_b = 1.5\ \sigma_a$, in order to avoid the selection of closely located centers. The algorithm terminates if $P_j^* <= \varepsilon P_1^*$ [43] [44], that is if the max potential P_j^* at j-th iteration drops below some fraction of the potential of the first center.

The choice of an appropriate bandwidth σ_a is the main problem, since choosing very small or very large values will result in a poor accuracy. This choice is usually done via extensive experimentation and trial-and-error. Using a very small radius will practically neglect the contributions of neighbouring data points in each potential $P(i)$. Using a large radius will increase the contributions of all the data points in each potential $P(i)$, not only the neighbour ones, thus cancelling the effect of dense clusters. Moreover the bandwidth is dataset dependent and the previous limits depend on the formation of a given dataset. An automatic or semi-automatic process is essential in order to avoid many user-defined parameters. In our scheme the leave-one-out kernel averaged gradient descent [29] provides an appropriate value.

5.3.2.2 The leave-one-out Kernel Averaged Gradient descent

We have recently proposed gradient descent learning of the leave-one-out kernel averaged regression function [29] to automatically estimate a bandwidth parameter for subtractive clustering. Here we repeat the main derivation steps. Given a dataset $\{\mathbf{x}_i, y_i\}_{i=1}^N$ where $\mathbf{x}_i$ represents points and y_j the desired labels (which will be defined later in eq. 5.9), the conventional kernel averaged (or weighted average) regression function $f(\mathbf{x})$ for an input $\mathbf{x}$ is:

$$f(\mathbf{x}) = \sum_k^N g_k(\mathbf{x})\, y_k \qquad \text{with } g_k(\mathbf{x}) = \varphi_k(\mathbf{x}) \Big/ \Big(\sum_j^N \varphi_j(\mathbf{x}) \Big) \tag{5.7}$$

where $\varphi_k(\mathbf{x}) = \exp(-\delta_k(\mathbf{x})/\sigma^2)$ is the Gaussian kernel, with bandwidth σ, and $\delta_k(\mathbf{x}) = \|\mathbf{x}_k - \mathbf{x}\|^2$ is the squared Euclidean distance between point $\mathbf{x}_k$ and $\mathbf{x}$.

When the input $\mathbf{x}$ is one of the $\mathbf{x}_i$ points in the dataset $\{\mathbf{x}_i\}_{i=1}^N$ the leave-one-out kernel averaged regression function [29] can be defined by leaving out from the sum a percentage γ of the self-contribution of $\mathbf{x}_i$ as:

$$f_{loo}(\mathbf{x}_i,\gamma) = \sum_k^N g_k(\mathbf{x}_i)\, y_k - \gamma\, g_i(\mathbf{x}_i)\, y_i \tag{5.8}$$

where γ is the small leave-one-out parameter. The proposed method needs the desired labels y_i for the points $\mathbf{x}_i$. We define them as [29]:

$$y_i = (1/N) \sum_{j=1}^N \| \mathbf{x}_j - \mathbf{x}_i \|^2 \tag{5.9}$$

Thus, each desired label y_i is considered as the variance of the corresponding $\mathbf{x}_i$, if this $\mathbf{x}_i$ was the center of the training set.

The squared error $E^i(\sigma, \mathbf{x}_i)$ for each $\mathbf{x}_i$ in terms of the prediction function $f_{loo}(\mathbf{x}_i,\gamma)$ is:

$$E^i(\sigma, \mathbf{x}_i) = (1/2)\, (f_{loo}(\mathbf{x}_i,\gamma) - y_i)^2 \tag{5.10}$$

The gradient descent update $\Delta\sigma = \sigma^{new} - \sigma^{old}$ for the bandwidth σ can be defined in terms of the gradient of the squared error $\partial\, E^i(\sigma,\mathbf{x}_i)/\partial\sigma$, with respect to bandwidth σ, as:

$$\Delta\sigma = -\,\xi\, \partial E^i(\sigma,\mathbf{x}_i)/\partial\sigma \tag{5.11}$$

where ξ is a learning rate.

The chain rule for the gradient $\partial E^i(\sigma,\mathbf{x}_i)/\partial\sigma$ gives:

$$\partial E^i(\sigma,\mathbf{x}_i)/\partial\sigma = (\partial E^i(\sigma,\mathbf{x}_i)/\partial f_{loo}(\mathbf{x}_i,\gamma))\ (\partial f_{loo}(\mathbf{x}_i,\gamma)/\partial\sigma)$$
$$= (f_{loo}(\mathbf{x}_i,\gamma) - y_i)\ (\partial f_{loo}(\mathbf{x}_i,\gamma)\ /\partial\sigma) \tag{5.12}$$

where the derivate $(\partial f_{loo}(\mathbf{x}_i,\gamma)\ /\partial\sigma)$ is:

$$\frac{\partial}{\partial\sigma} f_{loo}(\mathbf{x}_i,\gamma) = \sum_k^N \left(\frac{\partial}{\partial\sigma} g_k(\mathbf{x}_i)\ y_k\right) - \gamma \frac{\partial}{\partial\sigma} g_i(\mathbf{x}_i)\ y_i \tag{5.13}$$

where we only need to find the derivate $\partial g_k(\mathbf{x}_i)/\partial\sigma$ given by:

$$\frac{\partial}{\partial\sigma} g_k(\mathbf{x}_i) = \frac{\partial}{\partial\sigma}\ \left[\varphi_k(\mathbf{x}_i)\cdot\left(\sum_j^N \varphi_j(\mathbf{x}_i)\right)^{-1}\right] =$$

$$= \left(\frac{\partial}{\partial\sigma}\varphi_k(\mathbf{x}_i)\right)\cdot\left(\sum_j^N \varphi_j(\mathbf{x}_i)\right)^{-1} - \varphi_k(\mathbf{x}_i)\cdot\left(\sum_j^N \varphi_j(\mathbf{x}_i)\right)^{-2}\left(\sum_j^N \frac{\partial}{\partial\sigma}\varphi_j(\mathbf{x}_i)\right) \tag{5.14}$$

This equation by using $\frac{\partial}{\partial\sigma}\varphi_k(\mathbf{x}_i) = \varphi_k(\mathbf{x}_i)\ \delta_k(\mathbf{x}_i)\ /\sigma^3$ becomes:

$$\frac{\partial}{\partial\sigma} g_k(\mathbf{x}_i) = \left(\varphi_k(\mathbf{x}_i)\delta_k(\mathbf{x}_i)/\sigma^3\right)\cdot\left(\sum_j^N \varphi_j(\mathbf{x}_i)\right)^{-1} - \varphi_k(\mathbf{x}_i)\cdot\left(\sum_j^N \varphi_j(\mathbf{x}_i)\right)^{-2}\left(\sum_j^N (\varphi_j(\mathbf{x}_i)\ \delta_j(\mathbf{x}_i)/\sigma^3)\right)$$

$$= (1/\sigma^3)\ \varphi_k(\mathbf{x}_i)\left(\sum_j^N \varphi_j(\mathbf{x}_i)\right)^{-1}\left[\delta_k(\mathbf{x}_i) - \left(\sum_j^N \varphi_j(\mathbf{x}_i)\right)^{-1}\left(\sum_j^N (\varphi_j(\mathbf{x}_i)\ \delta_j(\mathbf{x}_i))\right)\right] \tag{5.15}$$

Eq. 5.15 by replacing each term $\varphi_k(\mathbf{x}_i)\left(\sum_j^N \varphi_j(\mathbf{x}_i)\right)^{-1}$ with $g_k(\mathbf{x}_i)$ gives the general derivate for any function $g_k(\mathbf{x}_i)$ as:

$$\frac{\partial}{\partial\sigma} g_k(\mathbf{x}_i) = (1/\sigma^3)\ g_k(\mathbf{x}_i)\ \left[\delta_k(\mathbf{x}_i) - \left(\sum_j^N (g_j(\mathbf{x}_i)\ \delta_j(\mathbf{x}_i))\right)\right] \tag{5.16}$$

The derivate $\partial g_i(\mathbf{x}_i)/\partial\sigma$ (of the contribution of $\mathbf{x}_i$ to itself) has a shorter expression produced by eq. 5.15 which after simplifications (by setting $\delta_i(\mathbf{x}_i) = 0$ and $\varphi_i(\mathbf{x}_i) = 1$) is:

$$\frac{\partial}{\partial\sigma} g_i(\mathbf{x}_i) = -(1/\sigma^3)\left(\sum_j^N (\varphi_j(\mathbf{x}_i)\ \delta_j(\mathbf{x}_i))\right)\left(\sum_j^N \varphi_j(\mathbf{x}_i)\right)^{-2} \tag{5.17}$$

Finally by substituting eq. 5.16 and eq. 5.17 into eq. 5.13 we can compute the derivate $(\partial f_{loo}(\mathbf{x}_i,\gamma)/\partial\sigma$. In a more shorthanded notation it gives:

$$\frac{\partial}{\partial\sigma} f_{loo}(\mathbf{x}_i,\gamma) = \frac{1}{\sigma^3}\left\{\sum_k^N\left(\left(\delta_k - \frac{\Sigma(\varphi_j\delta_j)}{\Sigma\varphi_j}\right)\cdot\frac{\varphi_k}{\Sigma\varphi_j}\ y_k\right) + \gamma\frac{\Sigma(\varphi_j\delta_j)}{(\Sigma\varphi_j)^2}\ y_i\right\} \tag{5.18}$$

The small leave-one-out parameter $\gamma \in [0, 1]$ prevents the gradient from converging into tiny values of the bandwidth σ. There exists a trade-off between γ=1 which gives large bandwidths and γ=0 which gives tiny bandwidths.

A naive implementation of eq. 5.18 (either sequential or parallel) will compute in the first round the partial sum variables $\Sigma\varphi_j$, $\Sigma\varphi_j\delta_j$ and in the second round will compute the outer sum $\sum_k^N$ and finally will add the leave-one-out term $\gamma\cdot y_i\cdot\Sigma\varphi_j\delta_j/(\Sigma\varphi_j)^2$. This strategy needs two passes over the N examples and raises the computational cost to

O(2*N*). However there is a more efficient implementation. If we decompose eq. 5.18 then we get:

$$\sum_{k}^{N}\frac{\delta_k\cdot\varphi_k\cdot y_k}{\Sigma\varphi_j}-\sum_{k}^{N}\left(\frac{\Sigma(\varphi_j\delta_j)}{\Sigma\varphi_j}\cdot\frac{\varphi_k\cdot y_k}{\Sigma\varphi_j}\right)+\gamma\frac{\Sigma(\varphi_j\delta_j)}{(\Sigma\varphi_j)^2}y_i=$$
$$=\frac{1}{\Sigma\varphi_j}\sum_{k}^{N}(\varphi_k\delta_k y_k)-\frac{\Sigma(\varphi_j\delta_j)}{(\Sigma\varphi_j)^2}\sum_{k}^{N}(\varphi_k y_k)+\gamma\frac{\Sigma(\varphi_j\delta_j)}{(\Sigma\varphi_j)^2}y_i \tag{5.19}$$

Eq. 5.19 needs only one pass over the *N* examples where for each $\mathbf{x}_i$ it computes:

$$\Sigma\varphi_j=\sum_{j}^{N}\varphi_j(\mathbf{x}_i) \tag{5.20a}$$

$$\Sigma\varphi_j\delta_j=\sum_{j}^{N}(\varphi_j(\mathbf{x}_i)\delta_j(\mathbf{x}_i)) \tag{5.20b}$$

$$\Sigma\varphi_k y_k=\sum_{k}^{N}(\varphi_k(\mathbf{x}_i)y_k) \tag{5.20c}$$

$$\Sigma\varphi_k\delta_k y_k=\sum_{k}^{N}(\varphi_k(\mathbf{x}_i)\delta_k(\mathbf{x}_i)\ y_k) \tag{5.20d}$$

Eq. 5.19, eq. 5.20 and eq. 5.7 reveal the partial sum variables $\Sigma\varphi_j$, $\Sigma\varphi_j\delta_j$, $\Sigma\varphi_k y_k$ (or $\Sigma\varphi_j y_j$) and $\Sigma\varphi_k\delta_k y_k$ we will need to circulate in the parallel ring pipeline. By avoiding the second round using the trick in eq. 5.19 the parallel computation cost is reduced from O(2*N*) to O(*N*).

5.3.2.3 Initializations and settings for the KG-SC algorithm

All initializations and settings for the Kernel Gradient Subtractive Clustering algorithm are the same as in [29]. For initializing the bandwidth σ in the beginning of the gradient descent (first epoch) there is a simple automatic way that avoids the need to search for different initial values of σ for different datasets. As in [29] we set the initial bandwidth equal to the trace of covariance matrix **R**. Hence, given *N* points $\mathbf{x}_n$ each one in *d* dimension, with their mean vector $\boldsymbol{\mu} = (1/N)\sum_n^N \mathbf{x}_n$ the covariance matrix can be calculated as $\mathbf{R} = (1/N)\sum_n^N(\boldsymbol{\mu} - \mathbf{x}_n)\ (\boldsymbol{\mu} - \mathbf{x}_n)^{\mathrm{T}}$ and the initial value of σ is $(1/d)\sum_i^d \sqrt{r_{ii}}$, where r_{ii} are the diagonal elements of **R**. Thus, the gradient descent starts with a relative large bandwidth σ which decreases immediately after the first epoch, until it converges.

The data features are scaled into the range [0, 1]. Without scaling the gradient might not converge, since the learning rate ξ depends on the scale of the feature space. By scaling the data features first, we can then use a fixed value for ξ for all datasets and thus avoiding the search for suitable learning rates each time we use a different dataset. Such scaling also avoids over-fitting. For the gradient descent we set a fixed learning rate $\xi = 0.2$ and maximum epochs = 10. Usually it converges after the first epoch if the dataset size is larger than 10000. Therefore, for the larger datasets we set maximum epochs = 2. For the leave-one-out kernel averaged regression function we set the leave-one-out parameter $\gamma = 0.1$ which is always sufficient enough to prevent bandwidth from converging into tiny values, so as to provide a stable solution.

For the settings in Subtractive Clustering (SC) we apply two minor modifications (see also [29]). First, instead of using a fixed $\sigma_b = 1.5\sigma_a$, we prefer to use a variable one that

equals to $\sigma_b = \sigma_a + 0.5\,(1.0 - k_{\text{sofar}}/N)\,\sigma_a$, which starts from $\sigma_b = 1.5\sigma_a$ and decays. As k_{sofar} (the number of selected exemplars so far) increases one-by-one during the potential $P(i)$ updating cycle of SC, the parameter σ_b gradually decreases and in the theoretical limit $k=N$ the value σ_b becomes equal to σ_a. This strategy works around the problem in higher dimensions where the fixed 1.5 percentage influences more strongly the nearby points. The second modified setting is in the termination of the potential updating cycle which terminates if the current max potential P^* become less that a threshold ($P^* < e\,P_1^*$). If e is selected to be very small, a large number of exemplars will be selected. On the contrary, a large value of e will lead to a small exemplar set. In order to avoid any other user-defined parameter we set $e = 1/P_1^*$. That is, Subtractive Clustering terminates at j-th iteration when $P_j^* < 1$. Thus, every point starts with potential $P(i) >= 1$ and finally ends up with potential $P(i) < 1$. There is a theoretical justification for this limit since $P(i)=1$ is the self-contribution of every i-th point to itself.

With these simple default settings, which are the same for all datasets, no adjustable user-defined parameters are required and the whole process is automatic.

5.3.3 Expectation-Maximization basics

Expectation-Maximization [30] [31] [32] is a classical approach for finalizing the PNN training. Given a set of problem parameters and a set of observed data the Expectation Maximization (EM) algorithm is an iterative method that converges to the maximum likelihood estimate of those problem parameters using the data. An iteration in the EM algorithm consists of an Expectation step (E-step) followed by a Maximization step (M-step). An Expectation step (E-step) computes the expected values of some missing or hidden data, using the current parameter estimate and the observed data. Then a Maximization step (M-step) uses the missing data measurements previously found in the E-step to form the log likelihood and compute the maximum likelihood estimate of the problem parameters.

The EM steps [30] are executed separately for each class ω until the maximum likelihood for all classes is reached. For a given class ω the PNN parameters are the mixing coefficients, centers, and covariances ($\beta_{m,\omega}$, $\mathbf{c}_{m,\omega}$, $\mathbf{R}_\omega$). The set of observed data is the class subset $\{\mathbf{x}_{n,\omega}, y_{n,\omega}\}_{n=1}^{N\omega}$ composed of N_ω data examples. Then the missing data are the 'unobserved' weights $w_{m,\omega}(\mathbf{x}_{n,\omega})$ that virtually relates components $\mathbf{c}_{m,\omega}$ with examples $\mathbf{x}_{n,\omega}$.

The E-step for the ω^{th} class computes the 'unobserved' weights $w_{m,\omega}(\mathbf{x}_{n,\omega})$ that relates every kernel $\{m=1,\ldots,M_\omega\}$ with every data point $\mathbf{x}_{n,\omega}$ $\{n=1,\ldots,N_\omega\}$ using:

$$w_{m,\omega}(\mathbf{x}_{n,\omega}) = \beta_{m,\omega} \cdot h(\mathbf{x}_{n,\omega}, \mathbf{c}_{m,\omega}, \mathbf{R}_\omega) \;/\; \Big(\sum_{j=1}^{M_\omega} \beta_{j,\omega} \cdot h(\mathbf{x}_{n,\omega}, \mathbf{c}_{j,\omega}, \mathbf{R}_\omega) \Big) \tag{5.21}$$

The M-step updates the new parameters by:

$$\beta_{m,\omega} = (1/N_\omega) \sum_{n=1}^{N_\omega} w_{m,\omega}(\mathbf{x}_{n,\omega}) \tag{5.22a}$$

$$\mathbf{c}_{m,\omega} = \Big(\sum_{n=1}^{N_\omega} w_{m,\omega}(\mathbf{x}_{n,\omega}) \cdot \mathbf{x}_{n,\omega} \Big) \;/\; \Big(\sum_{n=1}^{N_\omega} w_{m,\omega}(\mathbf{x}_{n,\omega}) \Big) \tag{5.22b}$$

$$\mathbf{R}_\omega = \Big(\sum_{n=1}^{N_\omega} \sum_{m=1}^{M_\omega} (\mathbf{x}_{n,\omega} - \mathbf{c}_{m,\omega}) \cdot w_{m,\omega}(\mathbf{x}_{n,\omega}) \cdot (\mathbf{x}_{n,\omega} - \mathbf{c}_{m,\omega})^{\mathrm{T}} \Big) \;/\; N_\omega \tag{5.22c}$$

The denominator N_ω in eq. 22c results from the sum of all $w_{m,\omega}(\mathbf{x}_{n,\omega})$ values. The iterative EM algorithm terminates when it reaches the maximum of the log-likelihood:

$$\log L_f = \sum_{\omega=1}^{K} \sum_{n}^{N_\omega} \log f_\omega(\mathbf{x}_{n,\omega}) \quad (5.23)$$

where $f_\omega(\mathbf{x})$ is given by eq. 5.3.

In the usual heteroscedastic PNN case, we consider here, each class covariance matrix $\mathbf{R}_\omega$ is diagonal, thus a different covariance value $r_{\omega,d}$ is used in the kernels for a different data feature d and a different class ω.

From Bayes theorem a weight $w_m(\mathbf{x}_n)$ expresses the posterior probability $p(\mathbf{c}_m \mid \mathbf{x}_n)=p(\mathbf{x}_n \mid \mathbf{c}_m)p(\mathbf{c}_m)/p(\mathbf{x}_n)$ that a given example $\mathbf{x}_n$ belongs to $\mathbf{c}_m$ cluster group. So the denominator in the $w_{m,\omega}(\mathbf{x}_{n,\omega})$ expression in eq. 5.21 is the prior probability $p(\mathbf{x}_n)$ of $\mathbf{x}_n$ occurrence. The priors of $p(\mathbf{c}_m)$ are the mixing coefficients β_m, and the conditional probability $p(\mathbf{x}_n \mid \mathbf{c}_m)$ of $\mathbf{x}_n$ given a $\mathbf{c}_m$ cluster group is the Gaussian probability density function $h(\mathbf{x}_n, \mathbf{c}_m, \mathbf{R})$, which sometimes is also denoted as $p(\mathbf{x}_n \mid \mathbf{c}_m, \mathbf{R})$.

5.3.4 The final PNN training scheme

The final PNN training scheme is:

1) use the leave-one-out kernel averaged gradient descent with Subtractive Clustering to find the PNN centers $\mathbf{c}_{n,\omega}$ for each ω^{th} class

2) use the previously extracted center set to train a PNN via Expectation-Maximization that refines the centers, covariances and mixing coefficients.

We demonstrate that all these algorithms can be performed in a parallel ring pipeline.

5.4 The basics of PNN Parallelization mappings

Concurrency exists in a computational problem when the problem can be decomposed into sub-problems that can be safely executed at the same time in parallel. Basically there are two decomposition patterns, neuron (task) decomposition and data decomposition. In the next we are going to see the data parallelism that learns the model by placing different training data partitions to each processor, and the neuron parallelism that splits the neurons to different processors.

5.4.1 Data Parallel training in Master/Worker

For data parallel training in the classical Master/Worker architecture (see fig 5.2) each Worker processor holds and operates on a different partition of data. The PNN implementation uses a Master processor for control (node 0). The Master sends the same copy of tasks (instruction stream) to all Workers. The Worker processors by receiving the copy of instructions, in our case the model parameters (neuron centers, weights, widths), they apply them to their own data partition. Then the Master accumulates the partial results of the Workers.

Each iteration step consists of a computation phase and a synchronous communication phase. This is a classical parallel architecture. However, because the Master node has the control, many synchronization points are applied in the parallel computation and a bottleneck effect might appear in this architecture which can slightly hinder the parallel execution times.

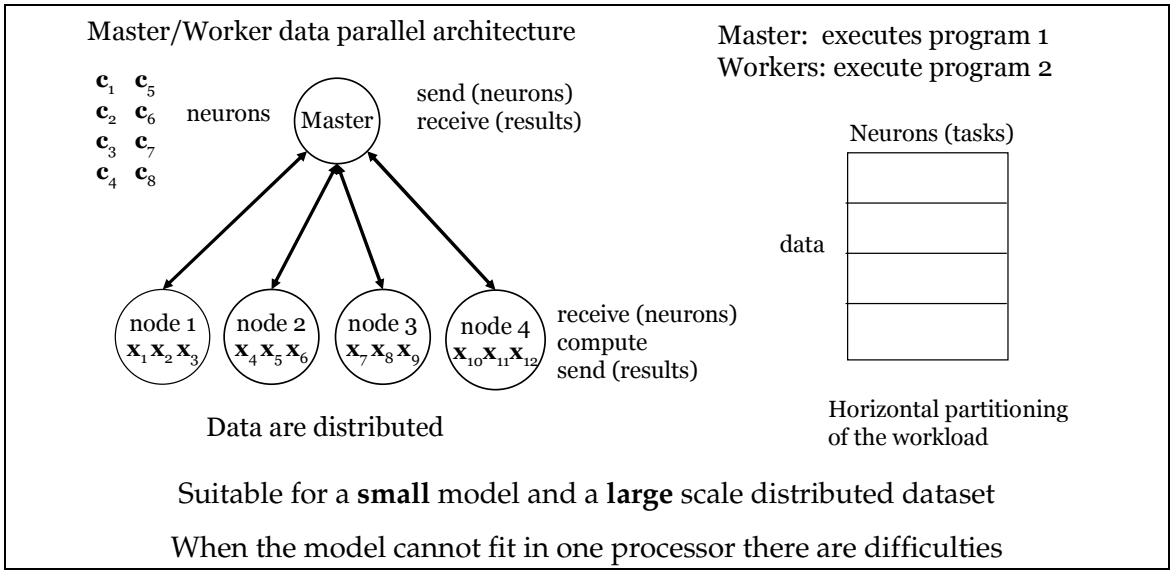


Figure 5.2. In Master/Worker data parallel architecture, the master processor holds the pattern neurons and the worker processors hold the data partitions.

5.4.2 Neuron parallel training in a pipeline

Figure 5.3 shows the neuron parallel training in the pipeline architecture. Here the pattern neuron parameters ($\mathbf{c}_m$, β_m, $\mathbf{R}$) are partitioned and distributed in different processors. The PNN neuron pipeline works like the instruction pipeline in parallel systems. The pipeline has three kinds of nodes, Source (first node), Seil (internal nodes) and Sink (last node). Every $\mathbf{x}$ is submitted for processing in parallel. The Source sends an instance $\mathbf{x}$ (or a block) of data to the next node. Each Seil node receives $\mathbf{x}$ and the list of partial sums from the previous node, then it adds the kernel function sums to the partial sum of the class conditionals of $\mathbf{x}$ and finally sends $\mathbf{x}$ and its list of partial sums to the next node. The Sink finalizes the computations.

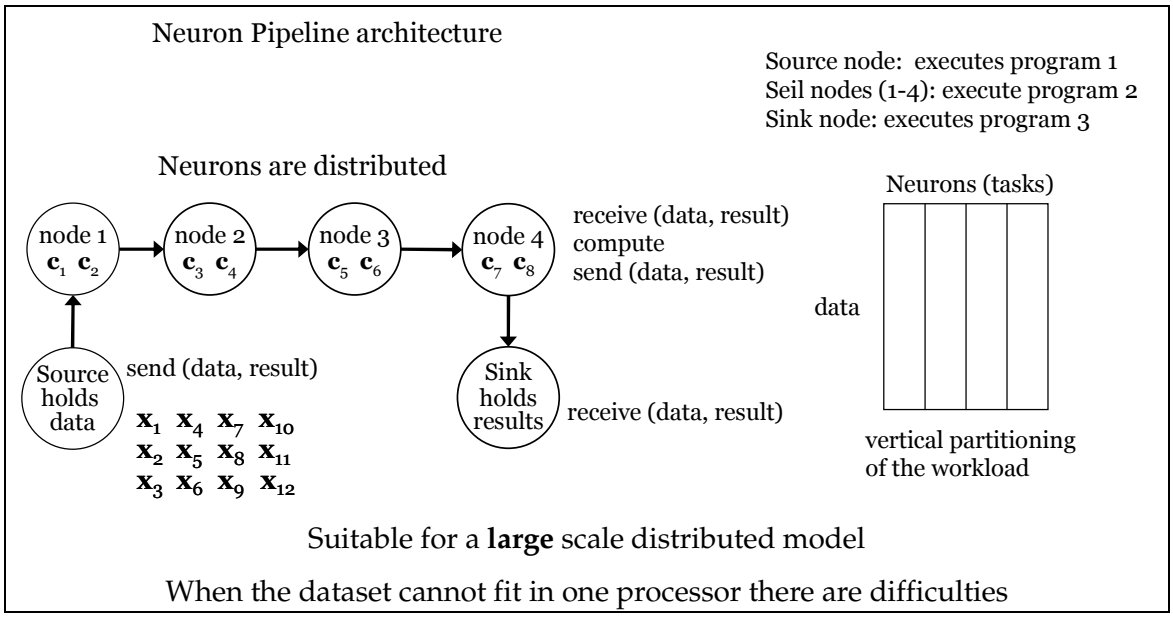


Figure 5.3. In Neuron parallel Pipeline the source holds the data and the other nodes hold the pattern neurons.

The idea behind pipeline topology [5] in processing elements (workstations, cores, chips, procedures) is to reduce as much as possible their idle time. An efficient parallel algorithm must try to minimize the communication, and to maximize the overlapping between communications and computations. The Pipeline parallel architecture reduces communications to point-to-point only. A processor receives messages only from its previous and sends messages only to its next processor. Overlapping communications with computations can be achieved if asynchronous (non-blocking) messages for send and receive commands are used and more importantly if a {Send-Compute-Receive} pattern is adopted. Each processor sends a message to the next, continues the computations with the available data, while the send/receive communications complete in the background using the buffer, and receives a message from the previous processor via the buffer.

In large-scale applications one can try to mix data and task parallelism simultaneously. Pipeline parallelism is a programming pattern mainly for task parallelism where the tasks are partitioned in a sequence of stages. Parallelism is accomplished by running stages concurrently on subsequent data vectors. A possible bottleneck might occur at the two ends of the pipeline, namely the Source which can delay to process the input stream and the Sink which can delay to process the output stream. Hence the number of processors is bounded by the input and output speed. On the other hand in the ring pipeline presented next all nodes participate in the same I/O.

5.5 Proposed Data-neuron PNN parallel training in a Ring pipeline

5.5.1 Ring pipeline topology basics

In a ring pipeline topology (see chapter 9.2 in [1] for data-instruction ring pipeline problems) the data are partitioned and equally distributed into the processor nodes which by their part are organized in a virtual ring pipeline. The ring structure that connects the processor nodes in a cluster is well known in neural networks [45] and ensembles [46]. This is the paradigm we employ for designing the PNN data-neuron ring pipeline in fig. 5.4. In our case each node holds a data portion as well as a neuron portion. The data and neuron distribution in the ring gives more flexibility. In the pipeline we can circulate on demand either the data or the neurons.

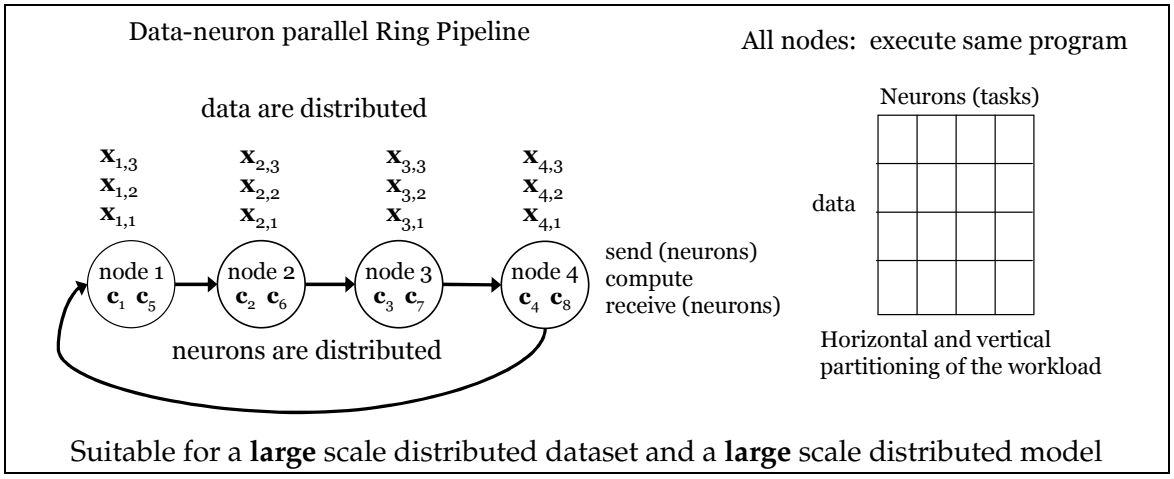


Figure 5.4. In PNN data-neuron ring pipeline (like data-task ring pipeline [1]) all nodes have both data partitions as well as neuron partitions. They can send either neurons or data. The pattern {Send-Compute-Receive} overlaps computations with communications.

In fig. 5.4 the {Send-Compute-Receive} pattern is adopted. Periodically every node sends a data-block to its next node, then uses this data-block for internal computations and then receives a data-block from its previous node. In this way all blocks are gradually propagated around the ring. We explicitly use the order {Send-Compute-Receive} for overlapping computations with communications and thus avoiding delays and bottlenecks. There is an additional advantage for data storage since data are distributed in all the processors which contribute equally. The problem is that not many algorithms can be mapped in the ring pipeline topology. We demonstrate that the proposed PNN training scheme can be executed entirely in this topology.

5.5.2 Overlapping delays

Ring Pipeline uses point-to-point communications. Thus a particular advantage is that we can use overlapping the communications delays with the computations delays (chapter 9.2 in [1]) so as to minimize the idle time. In Fig. 5.4 there are {Send-Compute-Receive} commands that have a communication delay and a computation delay. If the Compute command was the first and the Send command was second, then the two delays would have been cumulated in the ring. Instead, now they are overlapped.

1. Send (block) to next node.
2. Compute (block). During the data transfer communication the node perform partial computations with the available data in this block.
3. Receive (block) from previous node via the buffer.

The combined data partitioning and neuron partitioning permits available data in every node and available neurons in every node. Every process by finishing its computation step can see that a new block has already arrived from its previous node and thus can read it without further delay. In this way the effects from communication delays are eliminated in the ring pipeline, providing that computation time is larger than communication time, as it is in the PNN case.

5.5.3 Loops of {Send-Compute-Receive} commands

In the {Send-Compute-Receive} sequence, that manages to overlap delays, the Send and Compute commands use the same block. The main program consists of such commands that circulate the blocks (data or neurons) through the Ring pipeline and has two loops one inside the other. The same program is executed from all nodes as illustrated in fig. 5.5.

```
Every node:      //2 loops
  for cycle = 1 to B                    //for my data blocks
       Send (myBlock_cycle)
       Compute Partial (myBlock_cycle)
       Receive (prevBlock)
       for node = 2 to L                //for other data blocks
           Send (prevBlock)
           Compute Partial (prevBlock)
           Receive (prevBlock)
       end for
       Compute Final (prevBlock)        //finalize computations
  end for                               //my block list of size B becomes empty
```

Figure 5.5. A Ring pipeline program paradigm that circulates the data-blocks through the ring

Fig. 5.5 has two loops of many {Send-Compute-Receive} sequences. The first loop (outer) is for my data blocks, the second loop (inner) is for the blocks of the other nodes. In fig. 5.5 the function Compute Partial can be any function that either uses the processor's data partition and adds to the partial computations of the circulating block or uses the circulating block and adds to the partial computations of the processor's data partition.

This paradigm in fig. 5.5 is used in this work for all the ring pipeline training algorithms: 1) Leave-one-out kernel averaged gradient descent, 2) the Subtractive clustering, 3) the Expectation step and 4) the Maximization step.

5.5.4 Load balancing and program termination

In order for this PNN data-neuron ring pipeline to work efficiently a load balancing is required for the training data for each class as well as the pattern neurons which are implemented via the Gaussian kernels. For a given class ω a dataset of size N_ω is equally divided into the L nodes of the pipeline. By dividing N_ω/L there could be a remainder. That is if the modulo $N_\omega\%L$ isn't zero then the remainder examples will be distributed one-by-one into the nodes. So some nodes will have one more example. The same holds for the pattern neurons.

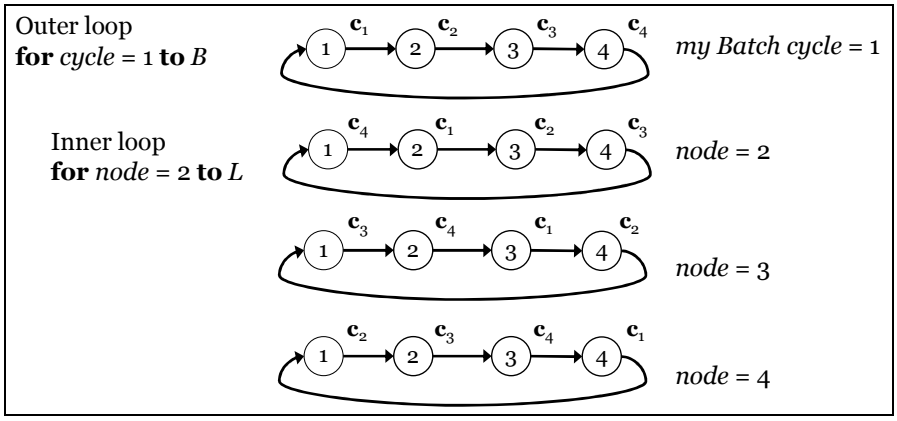


Figure 5.6. The stages in one PNN data-neuron ring pipeline cycle. All nodes send the center to the next node. The pipeline is always full.

A mechanism based on tokens is possible for cases where some nodes must continue to send data while others don't. Note however that this is unnecessary here since all the nodes will group their examples into blocks. Some nodes will have one more example only in their last block. Hence, in all the following ring pipeline parallel algorithms the load balancing is further ensured since every data partition is divided into a predefined number B of blocks, same to all nodes. The program in fig. 5.5 (the outer loop for cycle…) terminates when the block list is empty. The important observation is that the block lists in all nodes will have the same number B of blocks. This fact permits the nodes to end simultaneously when they receive their own last block. Fig. 5.6 illustrates the stages for the first cycle.

For the implementation of parallel PNN mappings in Cluster of workstations, with distributed memory, we have coded all parallel programs in C language using the Message Passing Interface (MPI) library [3] for the communications between processors. MPI is designed to be used with computer clusters and complements standard computer languages. MPI does not depend from the hardware that supports

and is available on almost all platforms. Thus we can use the parallel programs even in heterogeneous architectures.

5.5.5 Proposed Ring pipeline mini-batch kernel averaged gradient descent

There are many forms of gradient descent learning like online (or stochastic), batch or mini-batch. Online mode uses a single example at a time to compute the gradient. Batch mode computes the gradient by averaging the contribution of all examples. Mini-batch mode uses a mini-batch $\{\mathbf{x}_b, y_b\}_{b=1}^{L}$ of examples to compute the averaged gradient at a time. In this way the gradient is averaged over the $\mathbf{x}_b$ examples in the mini-batch. An epoch ends after all examples are introduced in a random order. The learning rate ξ can be constant or can vary at each epoch. For one epoch step the sequential mini-batch *leave-one-out kernel averaged gradient descent* is:

for $t = 1$ **to** B data blocks

form a mini-batch $\{\mathbf{x}_b, y_b\}_{b=1}^{L}$ of $\mathbf{x}_b$ examples (one from each node)

update the parameter σ by using $\sigma^{(t+1)} = \sigma^{(t)} - \xi^{(t)} \frac{1}{L} \sum_{b}^{L} \partial E(\sigma^{(t)}, \mathbf{x}_b)/\partial\sigma$

end for

Only mini-batch gradient descent is appropriate for parallel mapping in a ring pipeline.

Algorithm 1 shows one epoch of the Ring pipeline parallel mini-batch leave-one-out kernel averaged gradient descent. It adapts the {Send-Compute-Receive} pattern. There are two loops (outer and inner). The outer loop iterates through the number B of data blocks each node holds and sends through the pipeline. An epoch terminates when this data block list is empty. The inner loop iterates through the number L of nodes.

Every node sends a small block of examples $\mathbf{x}_i$ and their parameters into its next node. For the sake of clarity in algorithm 1 the blocks of data in each node contain one example $\mathbf{x}_i$. The same holds for the *send/receive* messages. The generalization to blocks of many examples is straightforward. At any cycle, the circulating mini-batch is composed from all the examples in the messages that circulate in the pipeline.

There is a mini batch of examples $\mathbf{x}_b$ with their partial sums $\Sigma\varphi_j$, $\Sigma\varphi_j\delta_j$, $\Sigma\varphi_j y_j$ and $\Sigma\varphi_k\delta_k y_k$ that circulates in the ring pipeline. The mini-batch size is a multiple of the number L of processors, which equally supply their examples. During passing throughout the nodes each $\mathbf{x}_b$ continues to sum up the local kernels contributions by using $\Sigma\varphi_j = \Sigma\varphi_j + \varphi_j(\mathbf{x}_b)$, $\Sigma\varphi_j\delta_j = \Sigma\varphi_j\delta_j + \varphi_j(\mathbf{x}_b) \cdot \delta_j(\mathbf{x}_b)$, $\Sigma\varphi_j y_j = \Sigma\varphi_j y_j + \varphi_j(\mathbf{x}_b) \cdot y_j$ and $\Sigma\varphi_k\delta_k y_k = \Sigma\varphi_k\delta_k y_k + \varphi_k(\mathbf{x}_b) \cdot \delta_k(\mathbf{x}_b) \cdot y_k$ until it computes the total sums, and finally arrives at its origin processor from which it had began circulating. Then the node that holds $\mathbf{x}_b$ will hold the final sums $\Sigma\varphi_j$, $\Sigma\varphi_j\delta_j$, $\Sigma\varphi_j y_j$, $\Sigma\varphi_k\delta_k y_k$ for this example and will compute the leave-one-out kernel averaged regression function [29] $f_{loo}(\mathbf{x}_b,\gamma) = \Sigma\varphi_j y_j / \Sigma\varphi_j - \gamma \cdot y_b / \Sigma\varphi_j$, the residual error $e_b = (f_{loo}(\mathbf{x}_b,\gamma) - y_b)$, the derivate $f_{loo}(\mathbf{x}_b,\gamma)/\partial\sigma = \Sigma\varphi_j\delta_j y_j / \Sigma\varphi_j - \Sigma\varphi_j\delta_j \cdot \Sigma\varphi_j y_j / (\Sigma\varphi_j)^2 + \gamma \cdot y_b \cdot (\Sigma\varphi_j\delta_j)/(\Sigma\varphi_j \cdot \Sigma\varphi_j)$ and the gradient of the squared error $\partial E_{\text{local}} = \partial E(\sigma^{(t)}, \mathbf{x}_b)/\partial\sigma$ for each $\mathbf{x}_b$ in the mini-batch. Then all nodes merge those $\partial E_{\text{local}}$ to form the sum ∂E_{global} for the mini-batch.

Algorithm 1: Ring pipeline parallel mini-batch gradient descent of the leave-one-out kernel averaged (one epoch). The number of processor nodes is L. Each node has N_{local} data. *my_rank* is the number of the node that runs the algorithm in the ring pipeline.

Every Node: //computes $\sigma^{(t+1)} = \sigma^{(t)} - \xi^{(t)} \; (1/L) \sum_b^L \; \partial E(\sigma^{(t)}, \mathbf{x}_b)/\partial\sigma$

next = Next(*my_rank*), *cycle* = 1 , $t = 1$

Random Shuffle the order of my local data list of $\{\mathbf{x}_i, y_i\}_{i=1}^{N_{local}}$

repeat

$\Sigma\varphi^{next} = 0$, $\Sigma\varphi\delta^{next} = 0$, $\Sigma\varphi y^{next} = 0$, $\Sigma\varphi\delta y^{next} = 0$, $i = cycle$

Send ($\mathbf{x}_i$, *my_rank*, $\Sigma\varphi^{next}$, $\Sigma\varphi\delta^{next}$, $\Sigma\varphi y^{next}$, $\Sigma\varphi\delta y^{next}$) //send my data point $\mathbf{x}_i$

Compute $s\varphi_{\text{local}}$, $s\varphi\delta_{\text{local}}$, $s\varphi y_{\text{local}}$, $s\varphi\delta y_{\text{local}}$ using ($\mathbf{x}_i$, $\sigma^{(t)}$))

Receive ($\mathbf{x}$, *rank*, $\Sigma\varphi$, $\Sigma\varphi\delta$, $\Sigma\varphi y$, $\Sigma\varphi\delta y$)

for *node* = 2 **to** L //for each one of the other nodes

$\Sigma\varphi=\Sigma\varphi+s\varphi_{\text{local}}$, $\Sigma\varphi\delta=\Sigma\varphi\delta+s\varphi\delta_{\text{local}}$, $\Sigma\varphi y=\Sigma\varphi y+s\varphi y_{\text{local}}$, $\Sigma\varphi\delta y=\Sigma\varphi\delta y+s\varphi\delta y_{\text{local}}$

Send ($\mathbf{x}$, *rank*, $\Sigma\varphi$, $\Sigma\varphi\delta$, $\Sigma\varphi y$, $\Sigma\varphi\delta y$)

Compute $s\varphi_{\text{local}}$, $s\varphi\delta_{\text{local}}$, $s\varphi y_{\text{local}}$, $s\varphi\delta y_{\text{local}}$ using ($\mathbf{x}$, $\sigma^{(t)}$))

Receive ($\mathbf{x}$, *rank*, $\Sigma\varphi$, $\Sigma\varphi\delta$, $\Sigma\varphi y$, $\Sigma\varphi\delta y$)

end for

$\Sigma\varphi=\Sigma\varphi+s\varphi_{\text{local}}$, $\Sigma\varphi\delta=\Sigma\varphi\delta+s\varphi\delta_{\text{local}}$, $\Sigma\varphi y=\Sigma\varphi y+s\varphi y_{\text{local}}$, $\Sigma\varphi\delta y=\Sigma\varphi\delta y+s\varphi\delta y_{\text{local}}$

//Close the loop

Send ($\Sigma\varphi$, $\Sigma\varphi\delta$, $\Sigma\varphi y$, $\Sigma\varphi\delta y$)

Receive ($\Sigma\varphi$, $\Sigma\varphi\delta$, $\Sigma\varphi y$, $\Sigma\varphi\delta y$)

//Find the leave-one-out kernel averaged regression function (see eq. 5.8)

$f_{loo}(\mathbf{x}_i,\gamma) = \Sigma\varphi y/\Sigma\varphi - \gamma \cdot y_i/\Sigma\varphi$

//Find the derivate of $f_{loo}(\mathbf{x}_i,\gamma)$ with respect to σ (see eq. 5.18 and eq. 5.19)

$f_{loo}(\mathbf{x}_i,\gamma)/\partial\sigma = \left(\Sigma\varphi\delta y/\Sigma\varphi - \Sigma\varphi\delta \cdot \Sigma\varphi y/(\Sigma\varphi)^2 + \gamma \cdot y_i \cdot (\Sigma\varphi\delta)/ (\Sigma\varphi)^2 \right) / (\sigma^{(t)})^3$

// Find the gradient of the squared error (see eq. 5.12)

$\partial E_{\text{local}} = \partial E(\sigma^{(t)}, \mathbf{x}_i)/\partial\sigma = (f_{loo}(\mathbf{x}_i,\gamma) - y_i) \; (\partial f_{loo}(\mathbf{x}_i,\gamma) \; /\partial\sigma)$

//here $\partial E_{\text{local}} = \frac{1}{\sigma^3}\left\{\sum_k^N\left(\left(\delta_k - \frac{\Sigma(\varphi_j\delta_j)}{\Sigma\varphi_j}\right)\cdot\frac{\varphi_k}{\Sigma\varphi_j}y_k\right) + \gamma\frac{\Sigma(\varphi_j\delta_j)}{(\Sigma\varphi_j)^2}y_i\right\}$ for my local point $\mathbf{x}_i$

Reduce $\partial E_{\text{local}}$ into $\partial E_{\text{global}}$ // $\partial E_{\text{global}}$ is the sum of all local $\partial E_{\text{local}}$

Broadcast $\partial E_{\text{global}}$

$\sigma^{(t+1)} = \sigma^{(t)} - \xi^{(t)} \; \partial E_{\text{global}} \; / \; L$ //find the new σ

cycle = *cycle* + 1 , $t = t + 1$

until (my local data list is empty)

Function Next(*node*) : **if** (*node*==L) **return** 1 **else return** *node*+1 **end if**.

Function Compute $s\varphi_{\text{local}}$, $s\varphi\delta_{\text{local}}$, $s\varphi y_{\text{local}}$, $s\varphi\delta y_{\text{local}}$ using ($\mathbf{x}$, $\sigma^{(t)}$) :

first compute the vector $\boldsymbol{\delta}(\mathbf{x}) = [\delta_1(\mathbf{x}) \ldots \delta_n(\mathbf{x}) \ldots \delta_{N_{local}}(\mathbf{x})]$

$s\varphi_{\text{local}} = \sum_{n=1}^{N_{local}} \varphi_n(\mathbf{x})$ where $\delta_n(\mathbf{x}) = ||\mathbf{x}_n - \mathbf{x}||^2$

$s\varphi\delta_{\text{local}} = \sum_{n=1}^{N_{local}} \varphi_n(\mathbf{x}) \; \delta_n(\mathbf{x})$ $\varphi_n(\mathbf{x}) = \exp(-\delta_n(\mathbf{x}) \; / \; (\sigma^{(t)})^2)$

$s\varphi y_{\text{local}} = \sum_{n=1}^{N_{local}} \varphi_n(\mathbf{x}) \; y_n$

$s\varphi\delta y_{\text{local}} = \sum_{n=1}^{N_{local}} \varphi_n(\mathbf{x}) \; \delta_n(\mathbf{x}) \; y_n$

The circulation is illustrated in fig. 5.7

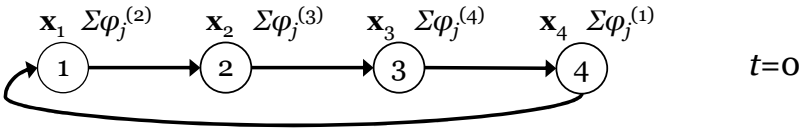


Figure 5.7. Circulating a mini-batch of points $\mathbf{x}_i$ and sums $\Sigma\varphi_j$, $\Sigma\varphi_j\delta_j$, $\Sigma\varphi_j y_j$, $\Sigma\varphi_j\delta_j y_j$. Each such sum accumulates the results of the point that precedes it in a round robin fashion.

5.5.6 Ring pipeline parallel subtractive clustering

After automatically finding the neighbourhood radius σ_a via the proposed leave-one-out kernel averaged gradient descent, and setting the parameter $a = (2/\sigma_a)^2$ then subtractive clustering can use this value for computing the potentials. The ring pipeline data-task distributed algorithm for Subtractive Clustering works as follows. The data points are partitioned and loaded into the processors, which must find and store the potentials $P(i)$ for their local data points $\mathbf{x}_i$. At any cycle every processor node Sends() a small block of data points into its next node, then Compute PartialPotentials() with the data in this block, and then Receive() from the queue buffer the block that has been arrived from the previous node. The whole process is presented in algorithm 2. For clarity reasons in the computations the send/receive messages in algorithm 2 use one data point $\mathbf{x}$. The generalization to messages that have blocks of several points is straightforward.

Algorithm 2: Ring pipeline in parallel Subtractive clustering for computing the potentials $P(i)$:

Every Node:
Input: number B of data blocks, my local data, parameter $a = (2/\sigma_a)^2$, number L of processors
Output: potentials $P(i)$ of my local data

```
for cycle = 1 to B                    //for each one of my data blocks
    Send (x_cycle)                    // here each cycle sends one point in the message
    Compute PartialPotentials (x_cycle)
    Receive (x_prev)
    for node = 2 to L                 //for each one of the other nodes
        Send (x_prev)
        Compute PartialPotentials (x_prev)
        Receive (x_prev)              // the last received message contains my data block
    end for
end for                               //my block list of size B becomes empty

Function Compute PartialPotentials (x) :
    for each x_i in my local Data Partition do
        P(i) = P(i) + exp(−a ||x_i − x||^2)
    end for
```

At the end of algorithm 2 every node in the ring pipeline will hold the potentials $P(i)$ for the $\mathbf{x}_i$ points of its own data partition.

The data for each class are distributed across the processor nodes, and the parallel algorithm is performed for each class separately. The algorithm is suitable for time-consuming large datasets.

After finding all the potentials the algorithm 3 presents the revision of the potentials and the selection of centers with higher potentials. These potential revision computations are fast and simple and are performed in an independently parallel fashion at each processor. Algorithm 3 proceeds by repeatedly execute the following cycle: (1) every node finds its best local point $\mathbf{x}^*_{local}$ with the highest local potential value P^*_{local}, (2) These pairs $\{\mathbf{x}^*_{local}, P^*_{local}\}$ are communicated through the ring pipeline. (3) every node finds among these locally best pairs the pair $\{\mathbf{x}^*, P^*\}$ with the highest potential $\{\mathbf{x}^*, P^*\}$. (4) every node performs local computations to revise the potentials of its own local points by using $P(i) = P(i) - P^*\exp(-b||\mathbf{x}^* - \mathbf{x}_i||^2)$. The costly computations performed here are those of the 4th step which are concurrently executed in parallel.

Algorithm 3: Ring pipeline in parallel Subtractive clustering for revising the potentials

Every Node:
Input: potentials $P(i)$, my local data partition, parameter b
Output: local centers

Find my local $\mathbf{x}^*_{local}$ with the highest local potential P^*_{local}
repeat
Send $(\mathbf{x}^*_{local}, P^*_{local})$
Set $P^* = P^*_{local}$, $\mathbf{x}^* = \mathbf{x}^*_{local}$
Receive $(\mathbf{x}^*_{prev}, P^*_{prev})$
for *node* = 2 **to** L //for each one of the other nodes
Send $(\mathbf{x}^*_{prev}, P^*_{prev})$
if $P^*_{prev} > P^*$ **then** set $P^* = P^*_{prev}$, $\mathbf{x}^* = \mathbf{x}^*_{prev}$ //update current max P^*
endif
Receive $(\mathbf{x}^*_{prev}, P^*_{prev})$
end for
Revise potentials $P(i)$ using $(\mathbf{x}^*, P^*)$ //finalize computations
Find my next local $\mathbf{x}^*_{local}$ with the highest local potential P^*_{local}
until $(P^* <= 1)$

Function Revise potentials $P(i)$ using $(\mathbf{x}^*, P^*)$:
for each $\mathbf{x}_i$ in my data partition **do**
$P(i) = P(i) - P^*\exp(-b||\mathbf{x}_i - \mathbf{x}^*||^2)$
end for

After performing the algorithms 1, 2 and 3 in the Ring pipeline for finding the centers and their number, then the EM algorithm refines all the PNN parameters.

5.6 Data distributed Ring parallel Expectation Maximization

The ring pipeline architecture, via the combined data and task partitioning strategy, is most suitable for parallelising the Expectation Maximization (EM) algorithm. If both the PNN parameters and the data are partitioned across the processors in the ring pipeline, we can straightforwardly apply the {Send-Compute-Receive} sequence of commands that avoids delays.

5.6.1 Related work on data distributed EM

In a distributed environment the data are partitioned by definition. Since the cost of computation at each node is much less than the cost of communication between nodes, avoiding transmitting the data (or centralizing them) is the only choice. Hence, the Gaussian component parameters (or tasks), which are much less than the data in the EM case, must be communicated between nodes. Existing studies explore strategies for training distributed EM [47] [48] [49] when the data are remain distributed in every processor location.

The work in [47] proposes a master-worker programming framework for a data parallel EM algorithm for computer vision applications. [47] uses Message Passing Interface (MPI) and stores a full matrix of size $N \times M$ for the E-step weights $w_m(\mathbf{x}_n)$. The E-step distributes the data in this matrix and each processor node computes and stores its own local set of $w_m(\mathbf{x}_n)$. The M-step distributes the computing by components, and uses a master-worker with all-gather operation. Different from [47] our implementation avoids storing the full matrix, achieving thus scalability, and uses a ring pipeline framework, minimizing thus communication bottlenecks, that distributes both data and components.

The work in [48] explores distributed EM algorithms for sensor networks. This algorithm performs local computations on the sensor data at each node and passes a small set of sufficient statistics from node-to-node. To this end, [48] adapts to the distributed environment the gradient based incremental EM [50], an alternative to the standard EM, where the unobserved variables can be represented by sums of component sufficient (locally or globally) statistics [50] [48]. These works [50] [48] also give the basic equations that express the learning parameters (mixing coefficients β_m, centers $\mathbf{c}_m$, and covariances $\mathbf{R}$ for the Gaussian mixture) in terms of the component sufficient statistics $\{\Sigma w_m, \mathbf{\Sigma c}_m, \mathbf{\Sigma c x}_m\}_{m=1}^{M}$. Because [48] explores a gradient-based EM it uses a full vector of size M for the component sufficient statistics $\{\Sigma w_m, \mathbf{\Sigma c}_m, \mathbf{\Sigma c x}_m\}_{m=1}^{M}$ that is passed from one node to another. Each node updates this vector using the local data and the local gradient. Then in a sequential way, it propagates the updated vector to the next node. However, at one cycle only one node at a time updates the statistics using its own local observations at each time step. Other nodes do nothing. That is because the gradient updating notion of the algorithm is based on a sequential propagation, and not parallel. [48] states in the conclusions that non-sequential updating strategies, like parallel and/or asynchronous schemes could be applied. Different from [48] we use the traditional EM and a ring pipeline parallel updating where all nodes simultaneously update the circulating statistics.

The work in [49] considers another gradient based distributed EM for sensor networks, in a very promising way. The learning parameters were estimated at each node locally, in parallel, using the local data. Each node holds a portion of data and the entire vector of all the learning parameters $\{\beta_m, \mathbf{c}_m, \mathbf{R}_m\}_{m=1}^{M}$. [49] uses the idea of the average consensus filter where each node holds the local summary quantities and the estimated global summary quantities from its neighbour nodes only. Using these values the consensus filter in each node outputs the updated estimated global summary quantities, which are also sent to the neighbours to implement the M-step. Eventually, the learning parameter set in each node will asymptotically converge to the true one. In essence, the low-pass consensus filter can smooth away the high-frequency noise

caused by the estimation of the global summary quantities. Each iteration diffuses the local sufficient statistics to the neighbour nodes and estimates the global sufficient statistics in each node. In the M-step of this algorithm [49], each sensor node uses its own estimated global sufficient statistics to update model parameters of the Gaussian mixtures. This distributed EM algorithm is a Robbins–Monro stochastic approximation to the gradient–ascent based EM algorithm as the filter states asymptotically converges to a local maximum of the log-likelihood. That is why [49] is suitable for sparsely connected graphs of sensor networks. However, it might require synchronization points during the communication of each node with its neighbour nodes (as it is local master-worker). Different from [49] we use the traditional EM, without the need of a gradient, and without synchronization points, and the EM learning parameters in our ring pipeline are distributed, while in [49] they existed (the same model is duplicated) in all nodes.

It is interesting to note here that the ring pipeline distributed EM can be employed to any kind of distributed data sources. For example, if the nodes in a sensor network can form a ring (as suggested in [48]) then the presented ring pipeline EM can also be straightforwardly applied.

5.6.2 Component statistics for EM

Similar to [50] [48] we proceed by defining component statistics for EM, or summary quantities (see also [49] [51] that use similar component statistics), which are simple sums for each m-th Gaussian component given by:

$$\Sigma w_m = \sum_{i=1}^{N_\omega} w_m(\mathbf{x}_i) \tag{5.24a}$$

$$\mathbf{\Sigma c}_m = \sum_{i=1}^{N_\omega} w_m(\mathbf{x}_i) \cdot \mathbf{x}_i \tag{5.24b}$$

$$\mathbf{\Sigma cx}_m = \sum_{i=1}^{N_\omega} w_m(\mathbf{x}_i) \cdot \mathbf{x}_i \cdot \mathbf{x}_i \tag{5.24c}$$

With these component statistics the M-step calculations for the component parameters $\{\mathbf{c}_m, \beta_m, \mathbf{R}\}$ can be written as:

$$\mathbf{c}_m^{new} = \mathbf{\Sigma c}_m \,/\, \Sigma w_m \tag{5.25a}$$

$$\beta_m^{new} = \Sigma w_m \,/\, N_\omega \tag{5.25b}$$

$$\mathbf{r}_\omega^{new} = (1/\, N_\omega) \sum_m^M (\mathbf{\Sigma cx}_m - (\mathbf{\Sigma c}_m \cdot \mathbf{\Sigma c}_m)/\, \Sigma w_m) \tag{5.25c}$$

Although [50] [48] did not provide further details on the simple decomposition of the covariance in terms of these component sufficient statistics, it can be easily deduced from the equality $(\mathbf{x} - \mathbf{c})^2 = \mathbf{x}^2 + \mathbf{c}^2 - 2\mathbf{xc}$.

[48] propagates the component sufficient statistics (for the center parameters) from the first node to the last node until they finally end up with the total sum. The data remain distributed across the nodes. [51] propagates the data vectors through the pipeline, from the first node to the last, while the component statistics are those that remain distributed across the processing nodes. In our version, the proposed ring pipeline, both data and component statistics are distributed. In our version all nodes operate simultaneously in parallel and each node propagates its own local vector of component statistics for its own EM parameters (the goal is to render every node source-seil-sink).

5.6.3 Covariance decomposition in terms of component statistics for EM

For a given class the decomposition for one diagonal covariance value r_d in the d^{th} dimension in terms of component statistics is given for N training examples $\mathbf{x}_n$ and M kernel centers $\mathbf{c}_m$ as follows:

$$r_d = (1/N)\sum_{n=1}^{N}\sum_{m=1}^{M} w_{n,m}(x_{n,d} - c_{m,d}^{new})^2 = (1/N)\sum_{m=1}^{M}\sum_{n=1}^{N} w_{n,m}(x_{n,d} - c_{m,d}^{new})^2$$

$$= (1/N)\sum_{m=1}^{M}\sum_{n=1}^{N} w_{n,m}(x_{n,d}^2 + (c_{m,d}^{new})^2 - 2c_{m,d}^{new}x_{n,d})$$

$$= (1/N)\left[\sum_{m=1}^{M}\sum_{n=1}^{N} w_{n,m}x_{n,d}^2 + \sum_{m=1}^{M}\sum_{n=1}^{N} w_{n,m}(c_{m,d}^{new})^2 - 2\sum_{m=1}^{M}\sum_{n=1}^{N} w_{n,m}c_{m,d}^{new}x_{n,d}\right] \quad (5.26)$$

where $w_{n,m}$ is the E-step weight $w_m(\mathbf{x}_n)$ that relates the m^{th} kernel with the n^{th} data point $\mathbf{x}_n$.

The second term is:

$$\sum_{m=1}^{M}\sum_{n=1}^{N} w_{n,m}(c_{m,d}^{new})^2 = \sum_{m=1}^{M}(c_{m,d}^{new})^2\sum_{n=1}^{N} w_{n,m} \quad (5.27)$$

The third term is:

$$2\sum_{m=1}^{M}\sum_{n=1}^{N} w_{n,m}c_{m,d}^{new}x_{n,d} = 2\sum_{m=1}^{M} c_{m,d}^{new}\sum_{n=1}^{N} w_{n,m}x_{n,d} = 2\sum_{m=1}^{M} c_{m,d}^{new}c_{m,d}^{new}\sum_{n=1}^{N} w_{n,m} \quad (5.28)$$

Thus the value r_d in d^{th} dimension becomes:

$$r_d = (1/N)\left[\sum_{m=1}^{M}\sum_{n=1}^{N} w_{n,m}x_{n,d}^2 - \sum_{m=1}^{M} c_{m,d}^{new}\sum_{n=1}^{N} w_{n,m}x_{n,d}\right] \quad (5.29)$$

Substituting the $c_{m,d}^{new}$ term gives:

$$r_d = (1/N)\left[\sum_{m=1}^{M}\sum_{n=1}^{N} w_{n,m}x_{n,d}^2 - \sum_{m=1}^{M}\frac{\sum_{n=1}^{N} w_{n,m}x_{n,d}\sum_{n=1}^{N} w_{n,m}x_{n,d}}{\sum_{n=1}^{N} w_{n,m}}\right] \quad (5.30)$$

Hence, in shorthand notation (that reveals the variables we will need to circulate in the ring pipeline) a covariance element is given in terms of component statistics by:

$$r_d = (1/N)\left[\sum_{m=1}^{M} scx_{m,d} - \sum_{m=1}^{M}\frac{sc_{m,d}\cdot sc_{m,d}}{sw_m}\right] \quad (5.31)$$

$$sw_m = \sum_{n=1}^{N} w_{n,m} \quad , \quad sc_{m,d} = \sum_{n=1}^{N} w_{n,m}x_{n,d} \quad , \quad scx_{m,d} = \sum_{n=1}^{N} w_{n,m}x_{n,d}^2$$

The variable sw_m is a scalar. The variable $\mathbf{sc}_m = [sc_{m,1}\ sc_{m,2}\ \ldots.\ sc_{m,d}]^T$ is a vector. The same holds for the variable $\mathbf{scx}_m = [scx_{m,1}\ scx_{m,2}\ \ldots.\ scx_{m,d}]^T$.

5.6.4 Proposed parallel Expectation-Maximization in data distributed Ring pipeline

It is important to avoid having the matrix of all the weights $w_{m,\omega}(\mathbf{x}_{n,\omega})$ in main memory, since there is a prohibited large storage O($M \cdot N$) cost. Traditionally to complete the E-step one must compute the weights $w_{m,\omega}(\mathbf{x}_{n,\omega})$. First, there is no need to actually store the matrix of the weights $w_{m,\omega}(\mathbf{x}_{n,\omega})$ which has O($M \cdot N$) storage cost. We need only to store the priors $p(\mathbf{x}_{n,\omega}) = \sum_{j=1}^{M_\omega} \beta_{j,\omega} \cdot h(\mathbf{x}_{n,\omega}, \mathbf{c}_{j,\omega}, \mathbf{R}_\omega)$ for every data point $\mathbf{x}_{n,\omega}$ and leaving the actual normalization of the weights for the M-step.

In the ring pipeline the centers (with their mixing coefficients) and the data are partitioned and distributed locally across the processors. Each processor has a local center partition and a local data partition. Load balancing is assured if we sub-divide each center partition into a predefined number B of blocks. B is the same for all processors which finish up their computing cycle simultaneously at the time they have sent all their blocks.

The following algorithm presents the ring pipeline E-step for one class by using the {Send-Compute-Receive} sequence. In the beginning all the prior probabilities $p(\mathbf{x}_i)$ are initialized to zero. For the sake of clarity the send/receive commands assume that every message has one center. The generalization to messages that contain a block of several centers is straightforward.

Algorithm 4: Ring pipeline Expectation-step

```
Every Node:                                   //finds prior probabilities p(x_i)
Input: number B of center blocks, my local data partition {x_i}_{i=1}^{Nlocal}
       my local centers c_m with their β_m mixing coefficients
Output: prior probabilities p(x_i) of my local data partition

  for cycle = 1 to B                          //for each one of my center blocks
     m = cycle                                //here each cycle sends one center
     Send (c_m, β_m)                          //send my center parameters {c_m, β_m }
     AddToMyPriors(c_m, β_m)
     Receive (c, β)                           //parameter blocks of the other nodes
     for node = 2 to L                        //for each one of the other nodes
        Send (c, β)
        AddToMyPriors(c, β)
        if (node != L) Receive (c, β) end if
     end for
  end for                                     //my local center list becomes empty

Function AddToMyPriors(c, β) :
  for each x_i in my local data partition do
     p(x_i) = p(x_i) + β · h(x_i, c, R)
  end for
```

At the end of algorithm 4 every node will hold the prior probabilities $p(\mathbf{x}_i)$ for the $\mathbf{x}_i$ points in its own data partition.

The Maximization-step-1 must compute for each center $\mathbf{c}_m$ the partial sums:

$$\Sigma w_m = \sum_{i=1}^{N_\omega} w_m(\mathbf{x}_i) = \sum_{i=1}^{N_\omega} \beta_m \cdot h(\mathbf{x}_i, \mathbf{c}_m, \mathbf{R}) \,/\, p(\mathbf{x}_i) \tag{5.32a}$$

$$\mathbf{\Sigma c}_m = \sum_{i=1}^{N_\omega} w_m(\mathbf{x}_i) \cdot \mathbf{x}_i = \sum_{i=1}^{N_\omega} \beta_m \cdot h(\mathbf{x}_i, \mathbf{c}_m, \mathbf{R}) \,/\, p(\mathbf{x}_i) \cdot \mathbf{x}_i \tag{5.32b}$$

$$\mathbf{\Sigma cx}_m = \sum_{i=1}^{N_\omega} w_m(\mathbf{x}_i) \cdot \mathbf{x}_i \cdot \mathbf{x}_i = \sum_{i=1}^{N_\omega} \beta_m \cdot h(\mathbf{x}_i, \mathbf{c}_m, \mathbf{R}) \,/\, p(\mathbf{x}_i) \cdot \mathbf{x}_i \cdot \mathbf{x}_i \tag{5.32c}$$

Algorithm 5 illustrates the Ring pipeline Maximization-step 1 for one class by using the {Send-Compute-Receive} sequence. It assumes that every send/receive message has one center in order to show the computations. The generalization to messages that contain a block of centers is straightforward. Σw_m , $\mathbf{\Sigma c}_m$ and $\mathbf{\Sigma cx}_m$ are partial sums. The number of processor nodes is L.

Algorithm 5: Ring pipeline Maximization-step 1

Every Node:

Input: number B of center blocks, my local dataset $\{\mathbf{x}_i\}_{i=1}^{Nlocal}$ with their priors $p(\mathbf{x}_i)$
my local centers $\{\mathbf{c}_m\}$ with their β_m mixing coefficients

Output: the sums Σw_m^{new}, $\mathbf{\Sigma c}_m^{new}$, $\mathbf{\Sigma cx}_m^{new}$ for each of my local centers

```
next = Next(my_rank)
cycle = 1
for cycle = 1 to B                          //for each one of my center blocks
    m = cycle                               //here each cycle sends one center
    Send (c_m, β_m, Σw^next, Σc^next, Σcx^next)   //send my parameter block
    Compute sw_local, sc_local , scx_local using (c_m, β_m)
    Receive (c, β, Σw, Σc, Σcx)             //receive parameter blocks of the other nodes
    for node = 2 to L                       //for each one of the other nodes
        Σw = Σw + sw_local                  //add the previously computed value sw_local
        Σc = Σc + sc_local                  //add the previously computed vector sc_local
        Σcx = Σcx + scx_local               //add the previously computed vector scx_local
        Send (c, β, Σw, Σc, Σcx)
        Compute sw_local, sc_local , scx_local using (c, β)
        Receive (c, β, Σw, Σc, Σcx)
    end for
    Σw = Σw + sw_local ,  Σc = Σc + sc_local,  Σcx = Σcx + scx_local
    Send (Σw, Σc, Σcx)
    Receive (Σw, Σc, Σcx)
    Σw_m^new = Σw ,  Σc_m^new = Σc,  Σcx_m^new = Σcx
end for                                     //my local center list becomes empty

Function Next(node) : if (node==L) return 1 else return node+1 end if
Function Compute sw_local , sc_local, scx_local using (c_m, β_m) :
    sw_local = 0,  sc_local = 0,  scx_local = 0
    for each local x_i in my data list do
        w_m(x_i) = β_m · h(x_i, c_m, R)/ p(x_i)
        sw_local = sw_local + w_m(x_i)
        sc_local = sc_local + w_m(x_i) · x_i
        scx_local = scx_local + w_m(x_i) · x_i · x_i
    end for
```

At the end of the algorithm 5 each node holds the new Σw_m^{new}, $\mathbf{\Sigma c}_m^{new}$, $\mathbf{\Sigma cx}_m^{new}$ for its own local centers.

The circulation in the Ring pipeline Maximization-step 1 is illustrated in Fig. 5.8.

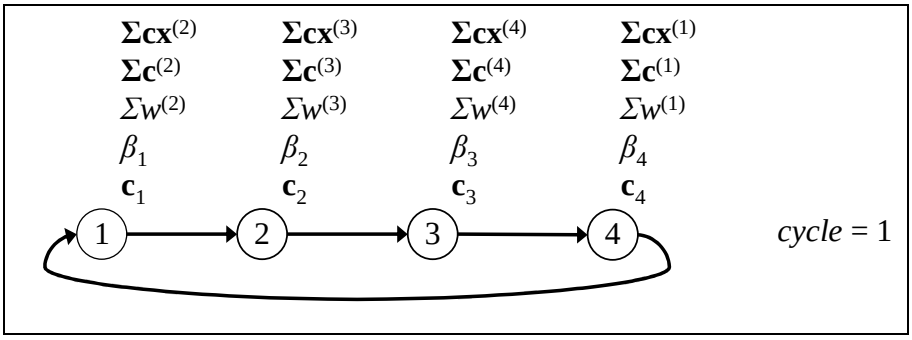


Figure 5.8. Circulating centers $\mathbf{c}_m$ and component statistics (the sums Σw_m, $\mathbf{\Sigma c}_m$ and $\mathbf{\Sigma cx}_m$) in the ring pipeline. Each sum accumulates the results of the center that precedes it in a round robin fashion.

Immediately after the Ring pipeline Maximization-step 1 it follows the Ring pipeline Maximization-step 2. This step is illustrated in algorithm 6, which simply sets up the new parameters and finds the new covariances.

Algorithm 6: Ring pipeline Maximization-step 2

Every Node: // just updates the new parameters in one last round

$\mathbf{R}_{\text{local}} = \sum_{m=1}^{M_{local}} (\mathbf{\Sigma cx}_m^{new} - \mathbf{\Sigma c}_m^{new} \cdot \mathbf{\Sigma c}_m^{new} / \Sigma w_m^{new})$

// compute $\mathbf{R}^{new}$ as the sum of all $\mathbf{R}_{\text{local}}$

Send ($\mathbf{R}_{\text{local}}$)

$\mathbf{R}^{new} = \mathbf{R}^{new} + \mathbf{R}_{\text{local}}$

Receive ($\mathbf{R}_{\text{prev}}$)

for *node* = 2 **to** L

Send ($\mathbf{R}_{\text{prev}}$)

$\mathbf{R}^{new} = \mathbf{R}^{new} + \mathbf{R}_{\text{prev}}$

Receive ($\mathbf{R}_{\text{prev}}$) //the last R_{prev} is mine that signals the loop termination

end for

//set up the new parameters

$\mathbf{R} = \mathbf{R}^{new}/N_\omega$

for each of my local centers $\mathbf{c}_m$

$\mathbf{c}_m = \mathbf{c}_m^{new} = \mathbf{\Sigma c}_m^{new} / \Sigma w_m^{new}$

$\beta_m = \beta_m^{new} = \Sigma w_m^{new} / N_\omega$

At the end of the algorithm 6 every node has set its own local centers, mixing coefficients and covariance to their new values.

The first advantage of this ring pipeline is the simplicity. The parallel algorithms are like a sequential algorithm, that additionally needs a send function and a receive function. No special synchronization points, or locking mechanisms or any other complex function is required. Such minimization of point-to-point communication delays is the most efficient among the other types in parallel processing.

The next advantage is that in every step the communication buffer in each processor that receives the messages from the previous node has all the information data available before the processor requests them (strategy of overlapping communication delays with computation delays).

5.7 Experimental Results

This section presents experimental simulations for accuracy and speed of the proposed method.

5.7.1 Benchmark Datasets

Experiments are performed on publicly available real-world benchmark datasets from the UCI data repository (http://www.ics.uci.edu/~mlearn/MLRepository.htm), the KEEL data repository (http://sci2s.ugr.es/keel) and the Infobiotics PSP repository (http://icos.cs.nott.ac.uk/datasets). Datasets are representatives of small scale, medium scale and large scale. The details of the datasets are illustrated in table 5.1.

Table 5.1. Benchmark datasets

Num	dataset	instances	features	classes
1	Iris	150	4	3
2	Wine	178	13	3
3	Ecoli	336	7	5
4	Ionosphere	351	33	2
5	Wisconsin	683	9	2
6	Diabetes	768	8	2
7	Vehicle Silhouettes	846	18	4
8	German Credit	1000	24	2
9	Banknote	1372	4	2
10	Yeast	1479	8	9
11	Spambase	4601	57	2
12	Waveform	5000	21	3
13	Banana	5300	2	2
14	Phoneme	5404	5	2
15	Page Blocks	5473	10	5
16	Texture	5500	40	11
17	Optical Digits	5620	64	10
18	Satellite Image	6435	36	6
19	Ring	7400	20	2
20	Pen Digits	10992	16	10
21	Magic telescope	19020	10	2
22	Letter	20000	16	26
23	Shuttle	58000	9	7
24	Skin Segmentation	245057	3	2
25	PSP protein	257560	20	2

5.7.2 Performance comparisons

We compare the classification error of all the training methods. In the experiments all the input features (attributes) have been normalized into the range [0, 1]. Each dataset is randomly split with stratification into a training set (70%) and a test set (30%). After the training phase of all algorithms is finished their classification error is measured on the test set. The experimental procedure is repeated 30 times for each dataset and the results are averaged. The comparison results are illustrated in table 5.2. PNN KG-SC in the last column is the proposed method of automatically selecting the k_ω centers for each class ω of the parsimonious PNN via the proposed leave-one-out kernel averaged gradient descent and subtractive clustering (KG-SC) and learning the remaining parameters via the classical EM. In PNN AP the k_ω centers for each class are automatically selected via Affinity Propagation. In PNN CV the PNN is trained via cross-validation using all the N data vectors. In PNN FPC the k_ω centers for each class (k_ω=10%N_ω) are found by the Farthest Point clustering and the training proceeds via EM. In PNN OFS the PNN is trained via OFS and EM. Here the k_ω centers for each class (k_ω=10%N_ω) are found via Orthogonal Forward Selection. In PNN KM the k_ω centers for each class (k_ω=10%N_ω) are initialized using K-Medoids algorithm and the remaining learning is again conducted via EM. In PNN RS the centers are randomly selected and their k_ω number for each class is the one found by the proposed PNN KG-SC. Random selection of centers shows that the number of centers is as important as their locations are, before the EM training. The last two columns of table 5.2 present the two methods, PNN AP and the proposed PNN KG-SC that are free of any user-defined parameters, like the k number which they found automatically.

Table 5.2. Classification Errors averaged over 30 runs. The proposed method kernel averaged gradient descent and subtractive clustering (KG-SC) is compared against Farthest Point Clustering (FPC), Orthogonal Forward Selection (OFS), K-Medoids (KM), Random Selection (RS) and Affinity Propagation (AP).

		PNN clustered inside classes & Expectation Maximization					
		Require the k number of centers				Automatically determine k centers	
Dataset	**PNN CV**	**PNN FPC**	**PNN OFS**	**PNN KM**	**PNN RS**	**PNN AP**	**PNN KG-SC**
Iris	3.0 ±1.76	2.1 ±1.68	**2.1** ±1.87	2.2 ±1.65	2.2 ±1.73	**2.1** ±1.56	**2.1** ±1.51
Wine	3.7 ±2.20	1.9 ±1.72	1.4 ±1.80	1.5 ±1.89	2.0 ±3.23	1.6 ±1.76	**1.1** ±1.64
Ecoli	**16.5** ±2.58	17.1 ±3.42	18.2 ±3.63	16.9 ±2.21	16.7 ±2.64	**16.5** ±2.79	**16.5** ±2.79
Ionosphere	10.3 ±1.95	8.0 ±1.89	7.1 ±1.61	6.9 ±1.85	6.7 ±1.96	**6.4** ±1.96	6.7 ±1.83
Wisconsin	**4.0** ±1.12	4.2 ±1.19	4.3 ±1.44	4.2 ±1.30	4.3 ±1.51	4.2 ±1.20	**4.0** ±1.49
Diabetes	26.7 ±2.13	27.2 ±2.39	27.0 ±2.54	26.9 ±2.65	27.0 ±2.65	26.8 ±2.49	**26.5** ±2.40
Vehicle	31.0 ±1.88	31.6 ±2.48	31.8 ±2.56	31.5 ±2.86	31.6 ±3.05	**30.2** ±2.19	**30.2** ±2.57

German Credit	34.3 ±2.03	31.4 ±2.05	33.1 ±3.45	32.1 ±3.15	33.7 ±3.93	32.8 ±3.25	**30.4** ±2.08
Banknote	0.16 ±0.09	0.09 ±0.05	0.08 ±0.05	0.08 ±0.05	**0.06** ±0.06	0.10 ±0.05	**0.06** ±0.05
Yeast	**43.7** ±2.20	49.3 ±2.26	49.5 ±2.30	49.4 ±2.13	50.8 ±2.76	49.6 ±2.94	49.8 ±2.52
Spambase	**9.5** ±0.80	**9.5** ±0.80	**9.5** ±0.82	**9.5** ±0.82	9.6 ±0.82	**9.5** ±0.82	**9.5** ±0.82
Waveform	14.9 ±0.78	18.1 ±0.92	18.6 ±0.87	18.0 ±0.76	14.7 ±0.79	18.3 ±0.68	**14.5** ±0.67
Banana	9.9 ±0.56	**9.8** ±0.62	**9.8** ±0.64	9.9 ±0.58	**9.8** ±0.60	**9.8** ±0.68	**9.8** ±0.56
Phoneme	**11.1** ±0.64	15.8 ±0.87	15.7 ±0.93	15.9 ±0.85	16.0 ±0.86	16.8 ±0.97	15.9 ±0.86
Page Blocks	**5.6** ±0.58	**5.6** ±0.58	**5.6** ±0.58	**5.6** ±0.58	**5.6** ±0.58	**5.6** ±0.58	**5.6** ±0.58
Texture	1.4 ±0.27	1.4 ±0.51	1.5 ±0.68	1.4 ±0.75	**1.3** ±0.62	1.5 ±0.78	**1.3** ±0.39
Optical Digits	**1.2** ±0.24	1.4 ±0.85	1.4 ±0.92	1.4 ±0.92	1.4 ±0.88	1.5 ±0.90	**1.2** ±0.29
Satellite Image	**10.9** ±0.49	11.0 ±0.71	11.0 ±0.81	11.0 ±0.80	11.2 ±0.79	**10.9** ±0.77	**10.9** ±0.74
Ring	24.3 ±0.72	4.0 ±0.41	4.9 ±0.46	4.5 ±0.45	2.2 ±0.29	6.5 ±0.41	**2.1** ±0.24
Pen Digits	**0.7** ±0.16	0.8 ±0.23	0.8 ±0.25	0.8 ±0.25	0.8 ±0.31	0.9 ±0.34	**0.7** ±0.28
Magic telescope	16.8 ±0.45	16.9 ±0.55	16.8 ±0.56	16.8 ±0.56	16.8 ±0.67	**16.7** ±0.54	**16.7** ±0.51
Letter	**5.7** ±0.21	6.3 ±0.23	6.8 ±0.25	6.3 ±0.25	6.7 ±0.28	6.5 ±0.23	6.3 ±0.25
Shuttle	2.0 ±0.8	1.4 ±0.5	--	--	**1.2** ±0.6	--	**1.2** ±0.6
Skin Segmentation	**0.05** ±0.02	**0.05** ±0.02	--	--	0.06 ±0.02	--	**0.05** ±0.02
PSP protein	**28.0** ±1.2	29.9 ±1.5	--	--	29.3 ±1.4	--	29.2 ±1.4

As can be seen from table 5.2, the proposed PNN KG-SC in the last column outperforms the others in most of the cases, while in the remaining cases it is as accurate as the other methods are. All the groups of experiments show that the number of the selected centers via the proposed KG-SC is sufficient to guarantee an accurate PNN. In addition fig. 5.9 and fig. 5.10 reveal that the proposed PNN KG-SC is the fastest among all the other center selection methods and in the same time it manages to select much fewer centers than them, thus producing parsimonious PNNs.

The last three datasets in table 5.2 are quite large and since K-medoids (PNN-KM), Orthogonal Forward Selection (PNN OFS) and Affinity Propagation (PNN AP) all have quadratic $O(N^2)$ memory requirements they run out of memory. For the Shuttle dataset KM and OFS need 8GB while AP needs 24GB. The Shuttle dataset has highly irregular class distributions and we use class prior probabilities $p(\omega)$ for each class ω equal to N_ω/N. These class priors gave much better results for all PNN versions. For the Skin dataset KM and OFS need 142GB while AP needs 443GB. The PSP protein has similar memory requirements. So for the last three datasets only the other algorithms (PNN CV, PNN FPC and PNN RS) are included in the comparisons of table 5.2.

For each dataset the number of selected PNN centers is given in table 5.3. In addition figure 5.9 illustrates a visual comparison of the number of selected centers between PNN AP, PNN KG-SC and k=10%N of the data. The Expectation Maximization training cost is proportional to the number of selected centers and the PNN classifier operation cost is also proportional to this number.

The proposed PNN KG-SC with EM scheme manages to use much fewer centers, thus increasing the classification speed while the computational cost is reduced. The KG-SC selects much fewer centers than 10%N. It is worth noting that the computational gains of the proposed approach are higher for the larger datasets.

Table 3. Number of selected centers for PNN AP and PNN KG-SC.

Dataset	Iris	Wine	Ecoli	Ionosphere	Wisconsin	Diabetes	Vehicle	German Credit	Banknote	Yeast	Spambase	Waveform	Banana
10% N	10	12	23	24	48	53	59	69	95	103	322	349	370
PNN AP	**20**	**38**	**46**	**50**	**95**	**75**	**65**	**160**	**61**	**152**	**419**	**396**	**211**
PNN KG-SC	**20**	**19**	**47**	**19**	**52**	**64**	**68**	**31**	**104**	**159**	**287**	**51**	**240**

Dataset	Phoneme	Page blocks	Texture	Optical Digits	Satellite Image	Ring	Pen Digits	Magic telescope	Letter	Shuttle	Skin Segment	PSP protein
10% N	378	382	385	392	450	517	769	1331	1399	4059	17153	18028
PNN AP	**170**	**145**	**320**	**652**	**325**	**911**	**513**	**428**	**1208**	**--**	**--**	**--**
PNN KG-SC	**350**	**185**	**333**	**631**	**216**	**22**	**468**	**446**	**1078**	**910**	**1514**	**627**

In several datasets like Iris, Ecoli, Australian Credit, Diabetes, Vehicle, Yeast, Banana, Page blocks, Thyroid, Texture, Optical Digits, Pen Digits, Magic telescope, both PNN AP and PNN KG-SC have automatically found almost the same number of centers.

The Ring dataset (no 19) was produced from two multivariate Gaussian distributions and PNN KG-SC can intrinsically detect those kinds of patterns. For the Shuttle dataset the proposed method PNN KG-SC by using 910 centers achieves lower error rate than the full PNN-CV which uses 40,590 centers, while PNN-FPC uses 4059 centers. For the Skin Segmentation dataset the proposed method PNN KG-SC uses 1514 centers and delivers the same accuracy with the full PNN-CV which uses 171,530 centers, while PNN-FPC uses 17,153 centers which are 11 times more than the proposed. This practically means that, since the EM training time is proportional to the number of

centers, PNN KG-SC EM training is 11 times faster than PNN-FPC EM. In the PSP protein dataset the proposed method PNN KG-SC uses an extremely small set of 627 centers in order to achieve the same accuracy with the full PNN-CV which uses 180,291 centers (287 times more). Actually PNN KG-SC reveals that in many cases much less centers are required for delivering the best performance.

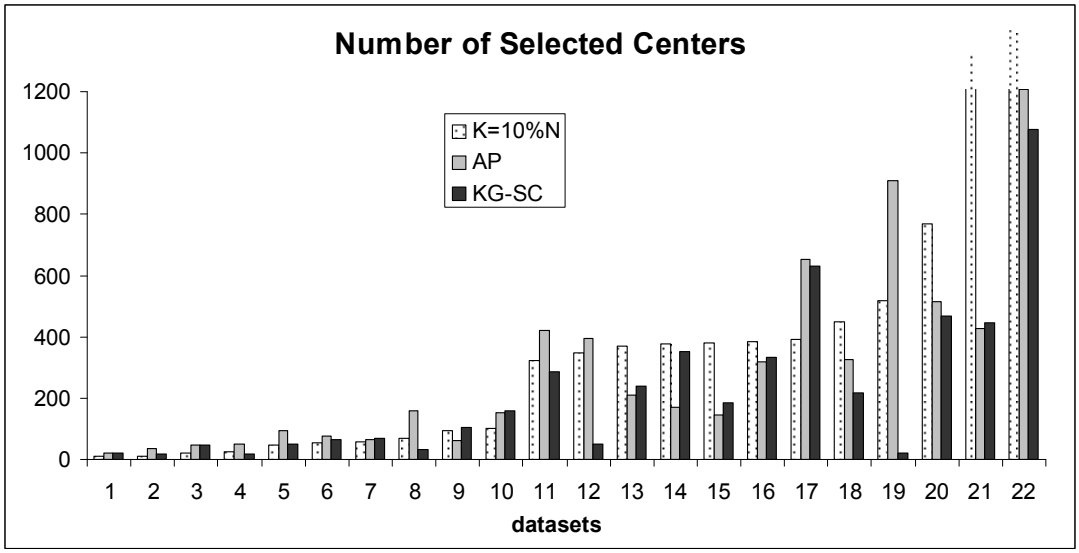


Figure 5.9. Comparison of the number of selected centers between AP and KG-SC. The 10% of the initial size N of the training data is also illustrated.

5.7.3 Complexity of the clustering phase

Fig. 5.10 shows the run times of all the sequential algorithms when trying to typically select k=10%N of centers. OFS has the highest run time and KG-SC the lowest run time.

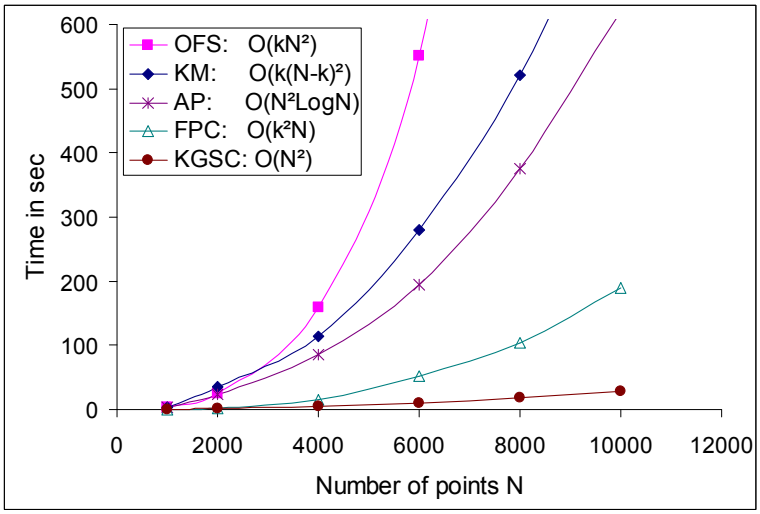


Figure 5.10. Run times for selecting k=10%N centers from N samples. From top to bottom the curves correspond to Orthogonal Forward Selection (OFS), K-Medoids (KM), Affinity Propagation (AP), Farthest Point Clustering (FPC) and Kernel average Gradient descent Subtractive Clustering (KG-SC).

Apart from KG-SC and FPC all other k-center samplers used in the comparisons were slow. Owning to their design few algorithms permit the pipeline implementation. Although the training via subtractive clustering algorithm can be pipelined in parallel,

the other k-center samplers like orthogonal forward selection, k-medoids, farthest point clustering and affinity propagation cannot.

5.7.4 Parallel performance metrics

In this section we present the performance results of the parallel implementations. A parallel computing system of distributed memory workstations is used, where each processor has a clock speed of 2.5 GHz, memory 2GB and Linux operating system. The machines are interconnected with 1000 Mbps Ethernet network. All parallel implementations are written in C using the Message Passing Interface (MPI) library [3] for the collective communication operations. During all experiments, we measured the training times for P=1 up to P=52 processors. For the evaluation of the parallel implementation we used three performance metrics [5] [4]: Amdahl's *Speedup* and *Efficiency*.

Speedup evaluates the strong scaling and finds out sequential bottlenecks by fixing the number N of data points and increasing the number P of processors (Amdahl). Let $T(1, N)$ be the execution time for the sequential algorithm to solve a dataset of size N on one processor and $T(P, N)$ be the execution time for a given parallel algorithm to solve the same problem of size N on P processors. Speedup is defined by the ratio $Speedup(P) = T(1, N)/T(P, N)$ where normally, $Speedup(P) \leq P$ and ideally $Speedup(P) = P$.

Efficiency measures the usage of the computational resources and provides a more convenient measure of the quality in the parallel algorithm. Efficiency computes the fraction of time between performance and the resources used to achieve that performance defined by $Efficiency(P) = T(1, N)/(P \times T(P, N)) = Speedup(P) / P$. Normally the $Efficiency(P) \leq 1$, while ideally $Efficiency(P) = 1$.

Figs. 5.11, 5.12, 5.13 and 5.14 present the speedup results of the ring pipeline parallel PNN training for its four steps respectively. We use five representative benchmark datasets with sizes ranging from medium, moderate and large. Based on the speedup results, it is clear that the speedup is progressively improved on increasing the dataset size.

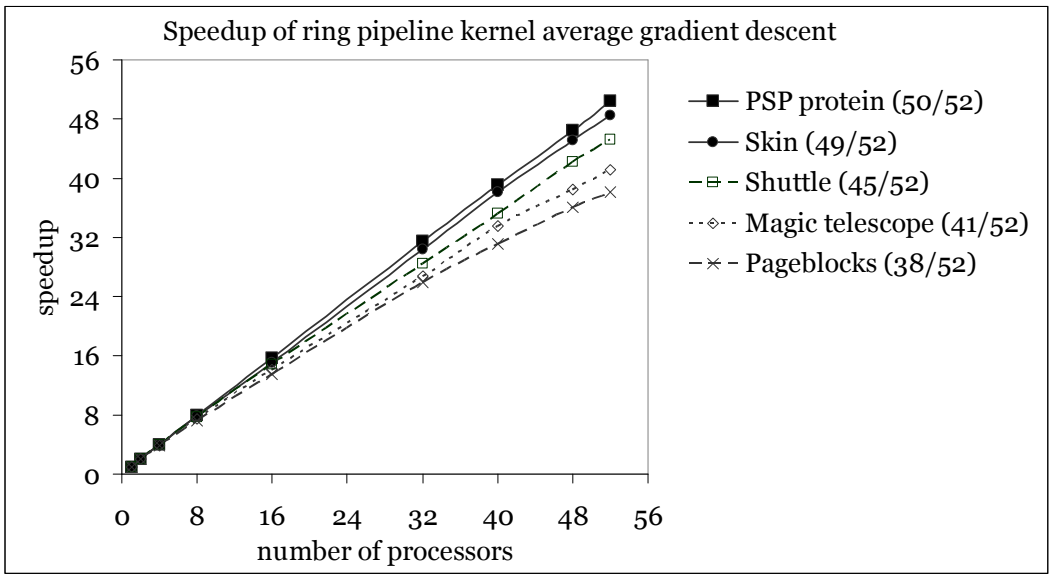


Figure 5.11. Speedup of the ring pipeline parallel Kernel Average Gradient Descent. For each dataset the corresponding *Efficiency* value for P=52 processors is indicated in the parenthesis. Almost ideal *Speedup* is observed for the larger size datasets.

Fig. 5.11 illustrates the Speedup curves with their Efficiency values obtained for the ring pipeline parallel Kernel Average Gradient Descent. The datasets are ranked based on their efficiency. For the larger datasets PSPprotein, Skin, Shuttle the speedup curves are very close to linear and they approach the ideal case, while for the moderate size datasets like Magic Telescope and Pageblocks the curves are clearly sub-linear.

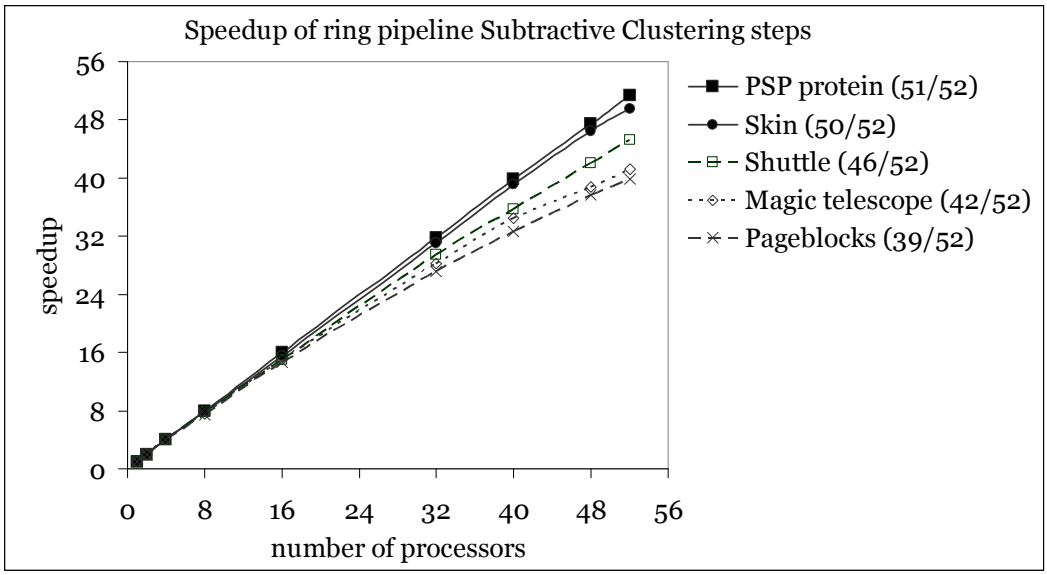


Figure 5.12. Speedup of the ring pipeline parallel Subtractive Clustering. For each dataset the corresponding *Efficiency* value for P=52 processors is indicated in the parenthesis. A noticeable almost linear *Speedup* is observed for the larger size datasets.

For ring pipeline parallel Subtractive Clustering the speedup curves in fig. 5.12 are sub-linear only for the moderate size datasets like Pageblocks and Magic Telescope. However, as illustrated in fig. 5.12 there are significant improvements for the larger datasets PSP protein, Skin and Shuttle, for which the speedup curves are very close to linear and the efficiencies are very close to one. The highest speedup occurs for the PSP protein dataset whose efficiency is (51/52).

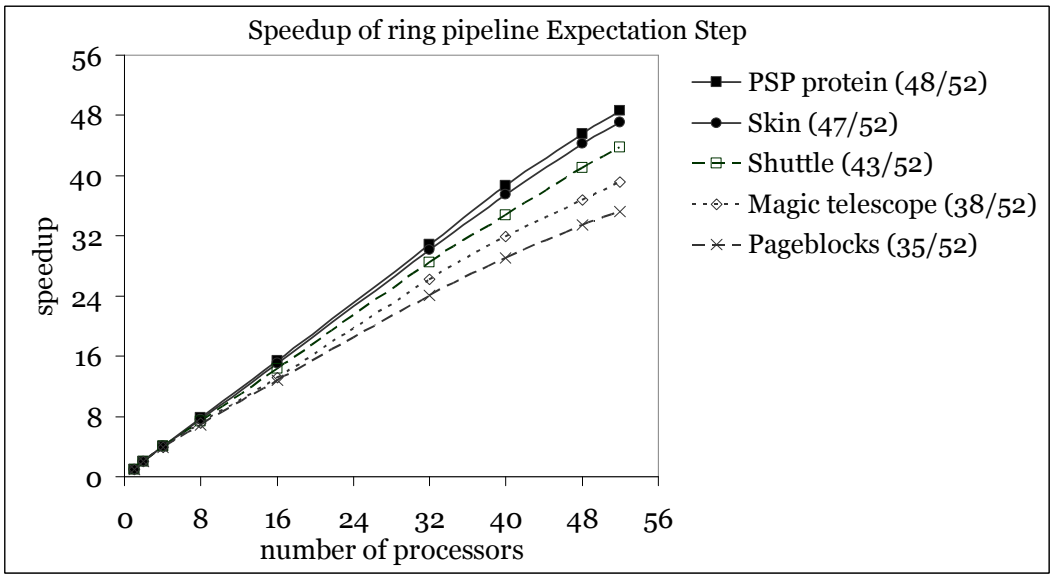


Figure 5.13. Speedup of the ring pipeline parallel Expectation Step. For each dataset the corresponding *Efficiency* value for P=52 processors is indicated in the parenthesis. For the larger size datasets an almost linear *Speedup* is observed.

Fig. 5.13 shows the ring pipeline parallel Expectation Step simulation results. Again the datasets are ranked based on their efficiency. The speedup curves are very close to linear for the larger datasets like PSP protein, Skin and Shuttle while for the moderate size datasets like Magic Telescope and Pageblocks the curves are evidently sub-linear. The efficiencies approach the ideal case as the data size increases.

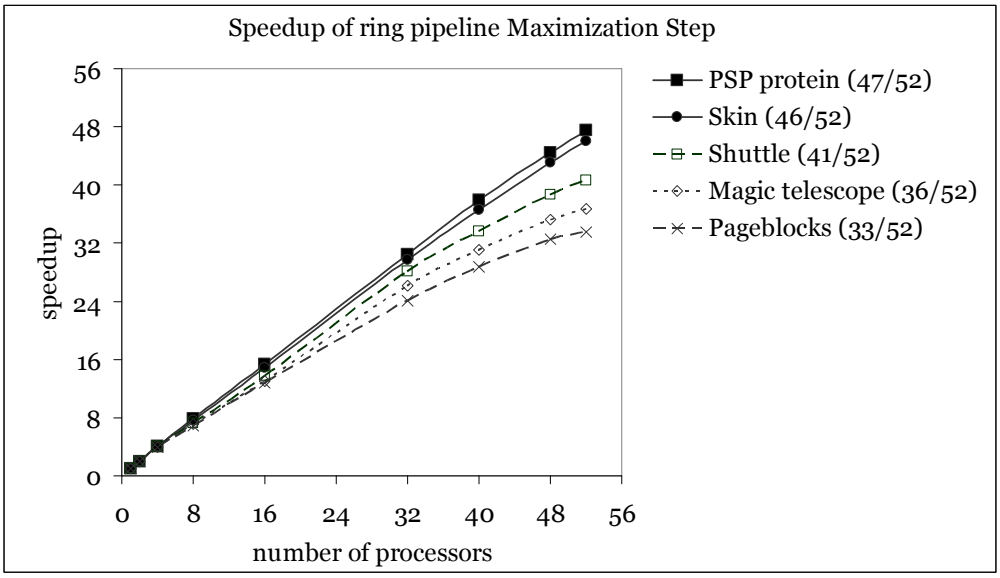


Figure 5.14. Speedup of the ring pipeline parallel Maximization Step. For each dataset the corresponding *Efficiency* value for P=52 processors is indicated in the parenthesis.

Fig. 5.14 gives the speedup curves for the ring pipeline parallel Maximization Step. The larger datasets PSP protein, Skin and Shuttle, have speedup curves close to linear and efficiencies close to one. The moderate size datasets Magic Telescope and Pageblocks have speedup curves sub-linear. The closer to ideal speedup occurs for the PSP protein and the Skin datasets.

5.8 Summary

We proposed a new learning scheme for training parsimonious Probabilistic Neural Networks, and show how this scheme can be implemented in a ring pipeline parallel architecture. Pipeline is a most efficient architecture for parallel computations since it does not suffer from common scalability problems, frequent communication delays and bottleneck effects, which are usually encountered in typical parallel systems. In order to automatically find the most representative centers from the data the recently proposed KG-SC algorithm is used. In consequence Expectation Maximization refines those centers and finds their mixing coefficients and covariances. All the algorithms in the presented PNN learning scheme are almost solely based on Gaussian summations which can by their part be efficiently mapped in parallel on distributed memory machines. The same ring pipeline scheme is used for: 1) the leave-one-out kernel averaged gradient descent, 2) the Subtractive clustering, 3) the Expectation step, 4) the Maximization step. The ring pipeline parallel implementations reveal nearly linear speedups on increasing the number of processors, and they are also scalable with increasing data points.

The produced parsimonious PNN model is compared with PNNs produced from other *k*-center clustering algorithms. The experiments show that the number of the selected centers via the proposed KG-SC is sufficient to guarantee an accurate PNN as

compared with the other existing methods like K-medoids (PNN-KM), Orthogonal Forward Selection (PNN OFS) and Affinity Propagation (PNN AP). The proposed scheme is promising since it has much lower computational complexity and memory requirements that the other methods in the comparisons. It is interesting that all the parallel implementations presented here consist of simple iterative algorithms which partition their executions in specific blocks for the ring pipeline. In theory, this design could be also suitable for large scale arrays of processing elements which have limited memory but a very large number, like neuro-chips or FPGAs. Future experiments could explore this possibility.

References for chapter 5

[1] Lester B.P. (1993) The art of Parallel programming. Prentice Hall.

[2] Margaritis K.G. and Evans D.J. (1992) Systolic implementation of neural networks. Parallel Computing 18, 325–334.

[3] Pacheco P. (1997) Parallel Programming with MPI. Morgan Kaufmann Publishers Inc, San Francisco.

[4] Grama A., Gupta A., Karypis G., and Kumar V. (2003) Introduction to Parallel Computing. Pearson Education Limited.

[5] Wilkinson B., Allen M., (2004) Parallel Programming: Techniques and Applications Using Networked Workstations and Parallel Computers (2nd Edition) Prentice Hall.

[6] Specht D.F. (1990) Probabilistic neural networks. Neural Networks 3, 109–118.

[7] Specht D.F. (1992) Enhancements to probabilistic neural networks. In Proc. IEEE Int. Joint Conf. Neural Networks, Baltimore, MD, 761–768.

[8] Mao K.Z., Tan K.-C., and Ser W. (2000) Probabilistic neural-network structure determination for pattern classification. IEEE Transactions on Neural Networks 11, 1009–1016.

[9] Georgiou V.L., Pavlidis N.G., Parsopoulos K.E., Alevizos P.D., Vrahatis M.N. (2006) New self-adaptive probabilistic neural networks in bioinformatic and medical tasks. International Journal of Artificial Intelligence Tools 15(3), 371–396.

[10] Grim J., Hora J. (2008) Iterative principles of recognition in probabilistic neural networks. Neural Networks 21, 838–846.

[11] Ubeyli E.D. (2008) Implementing eigenvector methods/probabilistic neural networks for analysis of EEG signals. Neural Networks 21(9), 1410–1417.

[12] Georgiou V.L., Alevizos P.D. and Vrahatis M.N. (2008) Novel approaches to probabilistic neural networks through bagging and evolutionary estimating of prior probabilities. Neural Processing Letters 27, 153–162.

[13] Adeli H., Panakkat A. (2009) A probabilistic neural network for earthquake magnitude prediction. Neural Networks 22, 1018–1024.

[14] Miguez R., Georgiopoulos M., Kaylani A. (2010) G-PNN: A genetically engineered probabilistic neural network. Nonlinear Analysis 73, 1783–1791.

[15] Mantzaris D., Anastassopoulos G., Adamopoulos A. (2011) Genetic algorithm pruning of probabilistic neural networks in medical disease estimation. Neural Networks 24(8), 831–835.

[16] Venkatesh S., Gopal S. (2011) Robust Heteroscedastic Probabilistic Neural Network for multiple source partial discharge pattern recognition - Significance of outliers on classification capability. Expert Systems with Applications 38(9), 11501–11514.

[17] Huang N., Xu D., Liu X., Lin L. (2012) Power quality disturbances classification based on S-transform and probabilistic neural network. Neurocomputing 98, 12–23.

[18] Kokkinos Y., Margaritis K.G. (2012) Parallelism, localization and chain gradient tuning combinations for fast scalable Probabilistic Neural Networks in data mining applications. In (SETN 2012), Springer Lecture Notes in Artificial Intelligence 7297, pp. 41–48.

[19] Khashei M., Bijari M. (2012) Hybridization of the probabilistic neural networks with feed-forward neural networks for forecasting. Engineering Applications of Artificial Intelligence, 25(6), 1277–1288.

[20] Savchenko A.V. (2013) Probabilistic neural network with homogeneity testing in recognition of discrete patterns set. Neural Networks 46, 227–241.

[21] Kokkinos Y. and Margaritis K.G. (2013) Distributed privacy-preserving P2P data mining via probabilistic neural network committee machines. IISA 2013: 1-4

[22] Kusy M., Zajdel R. (2014) Probabilistic neural network training procedure based on Q(0)-learning algorithm in medical data classification. Applied Intelligence 41(3), 837–854.

[23] Chen B.-H., Huang S.-C. (2015) Probabilistic neural networks based moving vehicles extraction algorithm for intelligent traffic surveillance systems. Information Sciences 299, 283–295.

[24] Kusy M., Zajdel R. (2015) Application of reinforcement learning algorithms for the adaptive computation of the smoothing parameter for probabilistic neural network. IEEE Transactions on Neural Networks and Learning Systems 26 (9), 2163-2175.

[25] Zhai J., Zhao W. (2016) Ensemble of multiresolution probabilistic neural network classifiers with fuzzy integral for face recognition. Journal of Intelligent and Fuzzy Systems 31(1), 405–414.

[26] Secretan J., Georgiopoulos M., Maidhof I., Shibly P., and Hecker J. (2006) Methods for Parallelizing the Probabilistic Neural Network on a Beowulf Cluster Computer. In International Joint Conference on Neural Networks, IJCNN 06, Vancouver pp. 2378–2385.

[27] Cardona K., Secretan J., Georgiopoulos M., and Anagnostopoulos G. (2007) A Grid Based System for Data Mining Using MapReduce. Technical Report, TR-2007-02

[28] Bastke S., Deml M., and Schmidt S. (2009) Combining statistical network data, probabilistic neural networks and the computational power of GPUs for anomaly detection in computer networks. In 19th International Conference on Automated Planning and Scheduling, Workshop on Intelligent Security (SecArt 2009), Thessaloniki, Greece,

[29] Kokkinos Y. and Margaritis K. G. (2016) Exemplar selection via leave-one-out kernel averaged gradient descent and Subtractive Clustering. In 12th International Conference on Artificial Intelligence Applications and Innovations (AIAI 2016), IFIP AICT 475, pp. 1–13.

[30] Streit R.L. and Luginbuhl T.E. (1994) Maximum likelihood training of probabilistic neural network. IEEE Transactions on Neural Networks, 5, 764–783.

[31] Yang Z.R., and Chen S. (1998) Robust maximum likelihood training of heteroscedastic probabilistic neural networks. Neural Networks, 11, 739–747.

[32] Tian B., and Azimi-Sadjadi M.R. (2001) Comparison of Two Different PNN Training Approaches for Satellite Cloud Data Classification. IEEE Transactions on Neural Networks, 12(1), 164–168.

[33] Gonzalez T.F. (1985) Clustering to minimize the maximum intercluster distance. Theoretical Computer Science, 38(2-3), 293–306.

[34] Frey B.J., Dueck, D. (2007) Clustering by passing messages between data points. Science, 315, 972–976

[35] Burrascano P. (1991) Learning vector quantization for the probabilistic neural network. IEEE Transactions on Neural Networks 2, 458−461.

[36] Traven H.G.C. (1991) A neural network approach to statistical pattern classification by "semi-parametric" estimation of probability density functions. IEEE Transactions on Neural Networks 2, 366–377.

[37] Babich G.A., Camps O.I. (1996) Weighted Parzen windows for pattern classification. IEEE Transactions on Pattern Analysis and Machine Intelligence, 18(5) 567–570.

[38] Berthold M., and Diamond J. (1998) Constructive training of probabilistic neural networks. Neurocomputing, 19, 167–183.

[39] Zhong M., Coggeshall D., Ghaneie E., Pope T., Rivera M., Georgiopoulos M., Anagnostopoulos G., Mollaghasemi M. and Richie S. (2007) Gap-Based Estimation: Choosing the Smoothing Parameters for Probabilistic and General Regression Neural Networks. Neural Computation 19, 2840–2864.

[40] Chang R.K.Y., Loo C.K. and Rao M.V.C. (2008) A Global k-means Approach for Autonomous Cluster Initialization of Probabilistic Neural Network. Informatica 32, 219−225

[41] Kokkinos Y. and Margaritis K.G. (2012) A Parallel Radial Basis Probabilistic Neural Network for scalable data mining in distributed memory machines. In 24th IEEE International Conference on Tools with Artificial Intelligence, (ICTAI 2012), 1094–1099

[42] Chiu, S.L. (1994). Fuzzy Model Identification Based on Cluster Estimation. Journal of Intelligent and Fuzzy Systems, 2, 267-278.

[43] Kothari, R., Pittas, D. (1999) On finding the number of clusters, Pattern Recognition Letters.

[44] Sarimveis H., Alexandridis A. and Bafas G., (2003) A fast training algorithm for RBF networks based on subtractive clustering, Neurocomputing, 51, 501−505.

[45] Parallel algorithms for Digital Image Processing, Computer Vision and Neural Networks. (I. Pitas eds.) WILEY 1993, pages 316−317

[46] Fernández C, Valle C, Saravia F, Allende H (2012) Behavior analysis of neural network ensemble algorithm on a virtual machine cluster. Neural Computing and Applications 21, 535–542.

[47] López de Teruel P., García J., and Acacio M. (1999) The parallel EM algorithm and its applications in computer vision. In International Conference on Parallel and Distributed Processing Techniques and Applications, pp. 571–578.

[48] Nowak R.D. (2003) Distributed EM algorithms for density estimation and clustering in sensor networks. IEEE Transactions on Signal Processing, 51(8), 2245–2253.

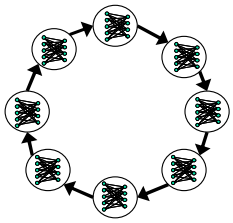
[49] Gu D. (2008) Distributed EM Algorithm for Gaussian Mixtures in Sensor Networks. IEEE Transactions on Neural Networks, 19(7), 1154−1166

[50] Neal R.M. and Hinton G.E. (1998) A view of the EM algorithm that justifies incremental, sparse, and other variants. In: Learning in Graphical Models, M. I. Jordan, Ed. Boston, MA: Kluwer, pp. 355–368.

[51] Guo C., Fu H., and Luk W. (2012) A fully-pipelined Expectation-Maximization engine for Gaussian Mixture models. In Proceedings of the International Conference on Field-Programmable Technology, (FPT 2012), pp. 182–189.

6 Parallel RBF Neural Networks in a ring pipeline for scalable regression

In this chapter we present a new effective, scalable and parallelized construction scheme for Radial Basis Function Neural Network (RBFNN). The proposed leave-one-out Kernel Averaged Gradient Descent algorithm with Subtractive Clustering algorithm is employed for automatically finding the RBFNN center locations and their number and mini-batch gradient descent learning is used for refining all the RBFNN parameters. We demonstrate that these steps can be efficiently mapped in a parallel ring pipeline scheme suitable for distributed memory machines.



We overcome the problems of implementing the mini-batch gradient descent in a pipeline by using the ring pipeline. We also show that in the Ring Pipeline one can straightforwardly use the {Send-Compute-Receive} pattern that permits overlapping computation delays with communication delays. The same pattern is used for all the algorithms. Experiments with the parallel implementations reveal speedups close to linear on increasing the number of processors. The proposed scheme is also scalable on increasing the size of the datasets.

6.1 Introduction

The Radial Basis Function (RBF) neural network [1] [2] [3] [4] is one of the most popular types of artificial neural networks. RBF neural networks are widely used for many years in various fields of machine learning, artificial intelligence, statistical analysis and many others. The popularity of RBF neural networks is due to their fast training and global approximation capabilities with locally tuned RBF responses, like biological neurons. There is a continuously increasing variety of tasks performed by RBF neural networks including function approximation [5], interpolation [6], system identification [7], dynamical modeling and control [7], classification [5], regression [1] [2] [4], data mining [3] for rule extraction, hierarchical learning [8] and others. Although most of the RBF training algorithms are fast they are still time consuming, especially for large training datasets. New techniques, which can further reduce the necessary computations, will enhance the applicability and the effectiveness of the RBF neural networks.

There are several different algorithms for optimized training in RBFNNs and popular examples are [9] strategies selecting the RBFs randomly from the training data,

strategies employing supervised procedures (i.e. gradient descent) for RBF center and weights estimation, or unsupervised procedures (i.e. clustering) for RBF center selection. Usually the first stage in RBF NN training is the initialization of the RBF centers which traditionally is implemented using clustering [2], and many works like [10] [11] [12] [13] [14] [15] have use clustering algorithms to create the initial model of RBFs. Many approaches also include gradient descent RBFNN parameter learning [1] [16] [17] [15] which are well established over the years.

The common problem of RBF Neural Network is that the number of hidden neurons usually depends on the number of training examples. On increasing the training data more hidden neurons are required to capture their complexity. Larger datasets require larger neural network models. For large scale datasets the RBFNN training and operation can become rather slow and quite demanding in memory and CPU resources.

Although conventional RBF neural network models might work poorly on large data collections, a direct way to satisfy their computational demands is to use parallel processing methods [18] [19] [20] [21] which are well known for cop with common scalability issues. The principal idea of parallel processing is to divide a large-scale problem into a number of smaller problems that can be solved concurrently, on either distributed memory or shared memory machines.

In many real systems data are largely distributed in the processors and learning all data from a single site would be difficult. Hence, if data must remain distributed locally then the RBF Neural Network model is the one that must be found in a global sense, by broadcasting its learning parameters to all processor nodes per training cycle. Usually, either the model or the data must be centralized. In the case of Master/Worker parallel architecture [19] [20] each Worker node holds a portion of the training data only, while the Master node holds the entire RBFNN model which is globally updated/communicated to all workers in each training cycle. In the case of pipeline parallel architecture each processor node holds a portion of the RBFNN model only, and the first node holds all the training data which are propagated through the pipeline from the first node to the last. In the case of the Ring Pipeline parallel architecture we study in this work, each node holds a portion of the RBFNN model together with a portion of the training data. This strategy permits easily the training of large scale models on large scale datasets.

To this end we present scalable RBFNN algorithms suitable for a Ring Pipeline parallel architecture in distributed memory machines. No user-defined parameters are required during the proposed training and the RBFNN model is created automatically. The training scheme uses the recently proposed leave-one-out Kernel Averaged Gradient descent Subtractive Clustering (KG-SC) [22] for selecting appropriate RBF centers and then proceeds by mini-batch gradient descent for learning all the RBF network parameters. The approach presented here is most suitable for both distributed clusters of workstations as well as shared memory parallel machines. Through the efficient mapping into the ring pipelined architecture the communication between parallel nodes is drastically reduced to point-to-point only and common execution bottlenecks as well as overheads due to frequent synchronizations are avoided.

In the Ring Pipeline we can easily use a {Send-Compute-Receive} pattern that permits for overlapping computation delays with communication delays. There is an additional advantage for data storage since data as well as neurons are distributed in all the processors which contribute equally. While not many algorithms can be mapped

explicitly in a ring pipeline topology we will show that the proposed RBFNN training scheme can. The proposed solution automatically produces the RBF neural network model and in the same time it is scalable to large data collections and parallelizable to large numbers of processors.

The system model we use is typically composed of several independent processors where each one has explicit access to its own memory and its own data partition [19]. We explicitly employ the ring pipeline parallel architecture. In the ring pipeline each processor can maintain a different data partition as well as a different neuron partition.

A similar ring pipeline of processing elements with a broadcast mechanism (messages send to all processors) is a general architecture. Some paradigms are clusters of workstations [23], digital neuro-computers in the *systolic ring architecture* [24], a ring of neural networks [25] and a virtual ring for ensembles [26].

6.2 Related work and preliminaries

6.2.1 Parallelising Neural Networks

For mapping of neural network into a parallel environment some steps are needed to define the optimal mapping scheme. These steps contain the parallelization of the sequential training algorithms and the identification of optimal decomposition of the network topology, by using either neuron parallelism, or training data parallelism. General guidelines exist (see taxonomy in [27] [28]) on how to break a Neural Network structure from coarse grained to fine grained levels in order to run them in parallel Studies in [29] [30] demonstrate systolic implementations of neural networks. [31] reports on backpropagation neural network on a cluster computer. The work in [32] implemented a parallel feed-forward neural network by dividing up training sets among the processors. [33] examines neural network parallelization in certain hardware platforms. A single gradient descent training algorithm with RBF neuron parallelism has been studied in [34]. [35] describes a parallel backpropagation in networks of workstations.

Differently from existing works we study how to parallelize both the structure determination and the parameter learning of RBF neurons on a ring pipeline topology which allows data parallelism together with neuron parallelism.

6.3 Proposed RBFNN training

In the mathematical formulas that follow throughout the paper an uppercase bold letter will symbolize a matrix, while a lowercase bold letter will symbolize a vector, and an italic letter will denote a scalar variable.

6.3.1 The basics of RBFNN

A typical RBF neural network has three layers which consist of input neurons, hidden neurons and output neurons respectively. The hidden units form a layer of K receptive fields, which have localized response functions $\varphi_k(\mathbf{x})$. These are the Radial Basis Function units. The hidden layer performs a nonlinear transformation and maps the input space onto a new space. An RBFNN topology is illustrated in fig. 6.1.

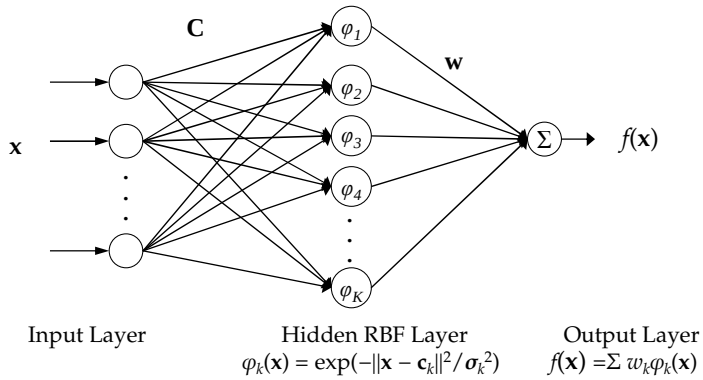


Figure 6.1. RBFNN topology with an input layer, a hidden layer and a single output neuron. Generalization to several outpouts is stratghtforward.

An RBFNN output for a given unknown $\mathbf{x}$ is the weighted linear combination of K in number radial basis functions $\varphi_k(\mathbf{x})$ given by:

$$f(\mathbf{x}) = \sum_k^K w_k \varphi_k(\mathbf{x}) \tag{6.1}$$

Each RBF unit $\varphi_k(\mathbf{x})$ is a strictly positive radially symmetric function with a unique maximum value at its center $\mathbf{c}_k$, away from which the values drop rapidly close to zero according to its width σ_k (or spread) of the peak around the center $\mathbf{c}_k$. The most common type of RBF activation function is the Gaussian function given by

$$\varphi_k(\mathbf{x}) = \exp(-\|\mathbf{c}_k - \mathbf{x}\|^2/(\sigma_k)^2) \tag{6.2}$$

With a given training set $\{\mathbf{x}_i, y_i\}_{i=1}^N$, the training algorithm of the RBFNN network search the parameter space in order to learn the best RBF centers $\mathbf{c}_k$, their number, along with their widths σ_k and output weights w_k.

Typically the learning scheme for the RBFNN training [4] [3] [36] can be performed in one-stage, two-stages or three-stages.

In ***one-stage learning*** all the network parameters are updated using a global gradient descent procedure [1] [10] [37] which is also suitable for online learning.

In ***two-stage learning*** [10] [2] the locations and widths of the RBF centers are determined in the first stage, by using clustering algorithms, and the output weights are computed in the second stage. The centers can be found by using unsupervised clustering methods [10] like K-means, fuzzy C-means, LVQ, Subtractive Clustering and others, or by using some form of gradient methods [1] [10]. In the second stage the output weights are computed by solving a least squares problem with methods like the pseudo-inverse, recursive least squares or gradient descent [1] [16] [15]. In [15] the RBF centers are initialized with subtractive clustering and the learning of all the network parameters proceeded with gradient descent. In [38] the output weights of the RBF units are obtained by linear least squares, while the RBF centers and variances are updated using a gradient descent procedure in a refining process that is performed repeatedly after the initial two-stage learning.

In the ***three-stage learning*** [36] the training of RBF networks can be split into an unsupervised part that finds centers and a linear supervised part that finds RBF widths and output weights. After finding the centers, the widths of RBF functions can be defined by using k-nearest neighbours or cluster variances or the minimum distance between all neighboring cluster centers [36] and find the output weights using least

squares (pseudo-inverse method). Many works use three-step learning. For example, in [13] the hidden RBF centers were found by Subtractive Clustering, their Gaussian widths were calculated using the P-nearest neighbor heuristic and the output weights were computed based on least squares.

A large portion of the aforementioned training methods is based on a combination of the cluster initialization for RBF center selection and the gradient descent for RBFNN parameter learning. In the next section the basics of gradient descent training will be given and in section 3.4 the proposed Kernel Gradient Subtractive Clustering (KG-SC) will be described in detail.

6.3.2 Gradient descent RBFNN parameter learning basics

For a set of RBFNN parameters $\{w_k, \mathbf{c}_k, \sigma_k\}$ the gradient descent learning [1] [16] [17] [15] phase tries to minimize the sum of squared errors at the network outputs, and updates the parameters values according to the signs of the partial derivatives. To do so, one has to compute the gradients of the error E^i with respect to the network parameters, namely weights, centers and their widths. Upon receiving an input pattern $\{\mathbf{x}, y\}$, the predicted RBFNN output is $f(\mathbf{x})$ given by eq. 6.1. All the RBF parameters are iteratively updated by taking the gradient of the squared error $E = (1/2)\,(y - f(\mathbf{x}))^2$ with respect to the predicted output $f(\mathbf{x})$, which can be readily calculated as $\partial E / \partial f(\mathbf{x}) = -(y - f(\mathbf{x}))$.

The gradient descent updates for the output weights w_k (k=1,2,…,K) are [1]:

$$\Delta w_k = -\xi \frac{\partial E}{\partial w_k} = -\xi \left(\frac{\partial E}{\partial f(\mathbf{x})} \right) \left(\frac{\partial f(\mathbf{x})}{\partial w_k} \right) = \xi \left(y - f(\mathbf{x}) \right) \varphi_k(\mathbf{x}) \tag{6.3}$$

where ξ is the learning rate and $\varphi_k(\mathbf{x})$ is the k^{th} RBF unit given by eq. 6.2.

The RBF center updates are defined as [1] $\Delta \mathbf{c}_k(d) = -\xi\, \partial E / \partial \mathbf{c}_k(d)$ for k=1 to K centers and d=1 to D dimensions. Here $\Delta \mathbf{c}_k(d)$ is the update for the d^{th} dimension of the k^{th} RBF center at the time the training pattern arrives. These updates can be derived by the chain rule as follows [1]:

$$\begin{aligned} \Delta \mathbf{c}_k(d) &= -\xi \frac{\partial E}{\partial \mathbf{c}_k(d)} = -\xi \left(\frac{\partial E}{\partial f(\mathbf{x})} \right) \left(\frac{\partial f(\mathbf{x})}{\partial \varphi_k(\mathbf{x})} \right) \left(\frac{\partial \varphi_k(\mathbf{x})}{\partial \mathbf{c}_k(d)} \right) \\ &= 2\xi \left(y - f(\mathbf{x}) \right) w_k \frac{\varphi_k(\mathbf{x})(\mathbf{x}(d) - \mathbf{c}_k(d))}{\sigma_k^2} \end{aligned} \tag{6.4}$$

where $\mathbf{x}(d)$ is the d^{th} dimension of the input pattern and σ_k is the width of the k^{th} RBF unit at the time this input pattern arrives.

The updates of the RBF widths are similarly derived by the chain rule as [1]:

$$\begin{aligned} \Delta \sigma_k &= -\xi \frac{\partial E}{\partial \sigma_k} = -\xi \left(\frac{\partial E}{\partial f(\mathbf{x})} \right) \left(\frac{\partial f(\mathbf{x})}{\partial \varphi_k(\mathbf{x})} \right) \left(\frac{\partial \varphi_k(\mathbf{x})}{\partial \sigma_k} \right) \\ &= 2\xi \left(y - f(\mathbf{x}) \right) w_k \frac{\varphi_k(\mathbf{x}) \left\| \mathbf{x} - \mathbf{c}_k \right\|^2}{\sigma_k^3} \end{aligned} \tag{6.5}$$

Many types of gradient learning exist like batch, online and mini-batch (or block). In batch learning, the whole training set is presented during a training cycle in order to estimate the gradient of the cost function and the updates are performed once per epoch. Since an epoch ends after all examples are introduced, in batch learning one

epoch equals one cycle. In online learning, the gradient of the cost function is estimated for one randomly selected example per cycle and hence the updates are executed once per example (one epoch equals N cycles). In mini-batch learning, the gradient of the cost function is computed per cycle by considering a mini-batch $\{\mathbf{x}_b, y_b\}_{b=1}^{L}$ of examples and thus the updates are performed per block (one epoch equals B cycles).

For one epoch step the sequential mini-batch *gradient descent* is:

for $t = 1$ **to** B data blocks

form a mini-batch $\{\mathbf{x}_b, y_b\}_{b=1}^{L}$ (here for clarity we use one example from each node)

update the RBF parameters by using:

$$\partial E / \partial w_k = -\frac{1}{L}\sum_i^L e_i \cdot \varphi_k(\mathbf{x}_i)$$

$$\partial E / \partial \mathbf{c}_k(d) = -w_k / (\sigma_k)^2 \cdot \frac{1}{L}\sum_i^L e_i \cdot \varphi_k(\mathbf{x}_i) \cdot (\mathbf{x}_i(d) - \mathbf{c}_k(d))$$

$$\partial E / \partial \sigma_k = -w_k / (\sigma_k)^3 \cdot \frac{1}{L}\sum_i^L e_i \cdot \varphi_k(\mathbf{x}_i) \cdot ||\mathbf{x}_i - \mathbf{c}_k||^2$$

end for

where $e_i = (y_i - f(\mathbf{x}_i))$

The mini-batch gradient descent learning of the RBFNN parameters is suitable for the ring pipeline parallel architecture.

The gradient descent learning of the RBFNN parameters requires that the number of RBF centers must be given by the user, together with the initial locations of the centers.

To this end we employ the recently proposed Kernel Gradient Subtractive Clustering (KG-SC) algorithm [22] that automatically selects the most representative exemplar centers from the training data.

6.3.3 Proposed Kernel Gradient Subtractive Clustering (KG-SC)

This section describes the recently proposed Kernel Gradient Subtractive Clustering (KG-SC) algorithm [22] that selects the most representative exemplar centers automatically, without any user-defined parameter. Before the descriptions of the parallel steps an introduction for the sequential algorithms is presented next.

6.3.3.1 Subtractive clustering basics

The Subtractive Clustering algorithm [39] [40] [13] [15] selects a set of most representative points (that serve as cluster centers) without any priori information about their number. Initially the algorithm considers every point $\mathbf{x}_i$ as possible center and computes its potential $P(i)$ defined by a sum of Gaussian kernels over all N data::

$$P(i) = \sum_{n=1}^{N} exp(-a||\mathbf{x}_i - \mathbf{x}_n||^2) \tag{6.6}$$

where $a = (2/\sigma_a)^2$ and σ_a is a bandwidth representing a neighbourhood radius. The potential $P(i)$ of $\mathbf{x}_i$ will be high if it has many neighbouring data points.

After finding all potentials $P(i)$ the algorithm repeatedly executes the following cycle:

1) Find data point $\mathbf{x}^*$ (center) with the highest potential value P^*

2) Revise the potential of all other points using $P(i) = P(i) - P^* \exp(-b||\mathbf{x}_i - \mathbf{x}^*||^2)$

where $b=(2/\sigma_b)^2$ and usually $\sigma_b = 1.5\,\sigma_a$ in order to avoid the selection of closely located centers. In step 2 the potential P^* of the selected point $\mathbf{x}^*$ will substantially affect all the potentials of the points near by. Thus, the data points near the selected point $\mathbf{x}^*$ will have significantly reduced density. The algorithm terminates if the max potential P_j^* at j-th iteration drops below some fraction of the potential of the first center, and the stopping criterion is $P_j^* <= \varepsilon P_1^*$ [40] [13] [15]. The choice for σ_a is difficult. Using a very small bandwidth σ_a will practically neglect the contributions of neighbouring data points in each potential $P(i)$ and all points will be selected. Using a small σ_a many points will be selected. Using a large σ_a will increase the contributions of all the data points in each potential $P(i)$, not only the neighbour ones, thus cancelling the effect of dense clusters and few points will be selected. Hence the main problem is how to find a suitable value for the bandwidth σ_a. In our scheme the recently proposed leave-one-out kernel averaged gradient descent [22] provides such a value. The details are given next.

6.3.3.2 The leave-one-out Kernel Averaged Gradient descent

Here we describe the main derivation steps of the recently proposed gradient descent learning of the leave-one-out kernel averaged regression function [22] that automatically estimates a bandwidth parameter for subtractive clustering. Given a dataset $\{\mathbf{x}_i, y_i\}_{i=1}^N$ where $\mathbf{x}_i$ represents points and y_j the desired labels (which will be defined later in eq. 6.9), the conventional kernel averaged (or weighted average) regression function $f(\mathbf{x})$ for an input $\mathbf{x}$ is:

$$f(\mathbf{x}) = \sum_k^N g_k(\mathbf{x})\, y_k \qquad \text{with } g_k(\mathbf{x}) = \varphi_k(\mathbf{x}) \Big/ \Big(\sum_j^N \varphi_j(\mathbf{x}) \Big) \tag{6.7}$$

where $\varphi_k(\mathbf{x}) = \exp(-\delta_k(\mathbf{x})/\sigma^2)$ is the common Gaussian kernel, with bandwidth σ, and $\delta_k(\mathbf{x}) = \|\mathbf{x}_k - \mathbf{x}\|^2$ is the squared Euclidean distance between point $\mathbf{x}_k$ and $\mathbf{x}$.

When the input $\mathbf{x}$ is one of the $\mathbf{x}_i$ points in the dataset $\{\mathbf{x}_i\}_{i=1}^N$ the leave-one-out kernel averaged regression function [22] can be defined by leaving out from the sum a percentage γ of the self-contribution of $\mathbf{x}_i$ as:

$$f_{loo}(\mathbf{x}_i,\gamma) = \sum_k^N g_k(\mathbf{x}_i)\, y_k - \gamma\, g_i(\mathbf{x}_i)\, y_i \tag{6.8}$$

where γ is the small leave-one-out parameter. The proposed method needs the desired labels y_i for the points $\mathbf{x}_i$ which are defined as [22]:

$$y_i = (1/N) \sum_{j=1}^N \| \mathbf{x}_j - \mathbf{x}_i \|^2 \tag{6.9}$$

Thus, each desired label y_i is considered as the variance of the corresponding $\mathbf{x}_i$, if this $\mathbf{x}_i$ was the center of the training set.

The squared error $E^i(\sigma, \mathbf{x}_i)$ for each $\mathbf{x}_i$ in terms of the prediction function $f_{loo}(\mathbf{x}_i,\gamma)$ is:

$$E^i(\sigma, \mathbf{x}_i) = (1/2)\, (f_{loo}(\mathbf{x}_i,\gamma) - y_i)^2 \tag{6.10}$$

The gradient descent update $\Delta\sigma = \sigma^{new} - \sigma^{old}$ for the bandwidth σ can be defined in terms of the gradient of the squared error $\partial\, E^i(\sigma,\mathbf{x}_i)/\partial\sigma$, with respect to bandwidth σ, as:

$$\Delta\sigma = -\,\xi\, \partial E^i(\sigma,\mathbf{x}_i)/\partial\sigma \tag{6.11}$$

where ξ is a learning rate.

The chain rule for the gradient $\partial E^i(\sigma,\mathbf{x}_i)/\partial\sigma$ gives:

$$\partial E^i(\sigma,\mathbf{x}_i)/\partial\sigma = (\partial E^i(\sigma,\mathbf{x}_i)/\partial f_{loo}(\mathbf{x}_i,\gamma))\ (\partial f_{loo}(\mathbf{x}_i,\gamma)/\partial\sigma)$$
$$= (f_{loo}(\mathbf{x}_i,\gamma) - y_i)\ (\partial f_{loo}(\mathbf{x}_i,\gamma)\ /\partial\sigma) \tag{6.12}$$

where the derivate $(\partial f_{loo}(\mathbf{x}_i,\gamma)\ /\partial\sigma)$ is:

$$\frac{\partial}{\partial\sigma} f_{loo}(\mathbf{x}_i,\gamma) = \sum_k^N \left(\frac{\partial}{\partial\sigma} g_k(\mathbf{x}_i)\ y_k\right) - \gamma \frac{\partial}{\partial\sigma} g_i(\mathbf{x}_i)\ y_i \tag{6.13}$$

where we only need to find the derivate $\partial g_k(\mathbf{x}_i)/\partial\sigma$ given by:

$$\frac{\partial}{\partial\sigma} g_k(\mathbf{x}_i) = \frac{\partial}{\partial\sigma}\ \left[\varphi_k(\mathbf{x}_i)\cdot\left(\sum_j^N \varphi_j(\mathbf{x}_i)\right)^{-1}\right] =$$

$$= \left(\frac{\partial}{\partial\sigma}\varphi_k(\mathbf{x}_i)\right)\cdot\left(\sum_j^N \varphi_j(\mathbf{x}_i)\right)^{-1} - \varphi_k(\mathbf{x}_i)\cdot\left(\sum_j^N \varphi_j(\mathbf{x}_i)\right)^{-2}\left(\sum_j^N \frac{\partial}{\partial\sigma}\varphi_j(\mathbf{x}_i)\right) \tag{6.14}$$

This equation by using $\frac{\partial}{\partial\sigma}\varphi_k(\mathbf{x}_i) = \varphi_k(\mathbf{x}_i)\ \delta_k(\mathbf{x}_i)\ /\sigma^3$ becomes:

$$\frac{\partial}{\partial\sigma} g_k(\mathbf{x}_i) = \left(\varphi_k(\mathbf{x}_i)\delta_k(\mathbf{x}_i)/\sigma^3\right)\cdot\left(\sum_j^N \varphi_j(\mathbf{x}_i)\right)^{-1} - \varphi_k(\mathbf{x}_i)\cdot\left(\sum_j^N \varphi_j(\mathbf{x}_i)\right)^{-2}\left(\sum_j^N \left(\varphi_j(\mathbf{x}_i)\ \delta_j(\mathbf{x}_i)/\sigma^3\right)\right)$$

$$= (1/\sigma^3)\ \varphi_k(\mathbf{x}_i)\left(\sum_j^N \varphi_j(\mathbf{x}_i)\right)^{-1}\left[\delta_k(\mathbf{x}_i) - \left(\sum_j^N \varphi_j(\mathbf{x}_i)\right)^{-1}\left(\sum_j^N \left(\varphi_j(\mathbf{x}_i)\ \delta_j(\mathbf{x}_i)\right)\right)\right] \tag{6.15}$$

Eq. 6.15 by replacing each term $\varphi_k(\mathbf{x}_i)\left(\sum_j^N \varphi_j(\mathbf{x}_i)\right)^{-1}$ with $g_k(\mathbf{x}_i)$ gives the general derivate for any function $g_k(\mathbf{x}_i)$ as:

$$\frac{\partial}{\partial\sigma} g_k(\mathbf{x}_i) = (1/\sigma^3)\ g_k(\mathbf{x}_i)\ \left[\delta_k(\mathbf{x}_i) - \left(\sum_j^N \left(g_j(\mathbf{x}_i)\ \delta_j(\mathbf{x}_i)\right)\right)\right] \tag{6.16}$$

The derivate $\partial g_i(\mathbf{x}_i)/\partial\sigma$ (of the contribution of $\mathbf{x}_i$ to itself) has a shorter expression produced by eq. 6.15 which after simplifications (by setting $\delta_i(\mathbf{x}_i) = 0$ and $\varphi_i(\mathbf{x}_i) = 1$) is:

$$\frac{\partial}{\partial\sigma} g_i(\mathbf{x}_i) = -(1/\sigma^3)\left(\sum_j^N \left(\varphi_j(\mathbf{x}_i)\ \delta_j(\mathbf{x}_i)\right)\right)\left(\sum_j^N \varphi_j(\mathbf{x}_i)\right)^{-2} \tag{6.17}$$

Finally by substituting eq. 6.16 and eq. 6.17 into eq. 6.13 we can compute the derivate $(\partial f_{loo}(\mathbf{x}_i,\gamma)/\partial\sigma$. In a more shorthanded notation it gives:

$$\frac{\partial}{\partial\sigma} f_{loo}(\mathbf{x}_i,\gamma) = \frac{1}{\sigma^3}\left\{\sum_k^N\left(\left(\delta_k - \frac{\Sigma(\varphi_j\delta_j)}{\Sigma\varphi_j}\right)\cdot\frac{\varphi_k}{\Sigma\varphi_j}\ y_k\right) + \gamma\frac{\Sigma(\varphi_j\delta_j)}{\left(\Sigma\varphi_j\right)^2}\ y_i\right\} \tag{6.18}$$

The leave-one-out parameter $\gamma \in [0, 1]$ prevents the gradient from converging into tiny values of the bandwidth σ. There exists a trade-off between γ=1 which gives large bandwidths and γ=0 which gives tiny bandwidths.

A naïve implementation of eq. 6.18 will compute in the first round the partial sum variables $\Sigma\varphi_j$, $\Sigma\varphi_j\delta_j$ and in the second round computes the outer sum $\sum_k^N$ and finally

adds the leave-one-out term $\gamma \cdot y_i \cdot \Sigma\varphi_j\delta_j / (\Sigma\varphi_j)^2$. This strategy needs two passes over the N examples and raises the cost to O(2N). However there is a more efficient implementation. If we decompose eq. 6.18 then we get:

$$\sum_k^N \frac{\delta_k \cdot \varphi_k \cdot y_k}{\Sigma\varphi_j} - \sum_k^N \left(\frac{\Sigma(\varphi_j\delta_j)}{\Sigma\varphi_j} \cdot \frac{\varphi_k \cdot y_k}{\Sigma\varphi_j} \right) + \gamma \frac{\Sigma(\varphi_j\delta_j)}{(\Sigma\varphi_j)^2} y_i = \\ = \frac{1}{\Sigma\varphi_j} \sum_k^N (\varphi_k \delta_k y_k) - \frac{\Sigma(\varphi_j\delta_j)}{(\Sigma\varphi_j)^2} \sum_k^N (\varphi_k y_k) + \gamma \frac{\Sigma(\varphi_j\delta_j)}{(\Sigma\varphi_j)^2} y_i \tag{6.19}$$

Eq. 6.19 needs only one pass over the N examples where for each $\mathbf{x}_i$ it computes:

$$\Sigma\varphi_j = \sum_j^N \varphi_j(\mathbf{x}_i) \tag{6.20a}$$

$$\Sigma\varphi_j\delta_j = \sum_j^N (\varphi_j(\mathbf{x}_i)\delta_j(\mathbf{x}_i)) \tag{6.20b}$$

$$\Sigma\varphi_k y_k = \sum_k^N (\varphi_k(\mathbf{x}_i) y_k) \tag{6.20c}$$

$$\Sigma\varphi_k\delta_k y_k = \sum_k^N (\varphi_k(\mathbf{x}_i)\delta_k(\mathbf{x}_i)\, y_k) \tag{6.20d}$$

Eq. 6.19, eq. 6.20 and eq. 6.7 reveal the partial sum variables $\Sigma\varphi_j$, $\Sigma\varphi_j\delta_j$, $\Sigma\varphi_k y_k$ (or $\Sigma\varphi_j y_j$) and $\Sigma\varphi_k\delta_k y_k$ we will need to circulate in the parallel ring pipeline. By avoiding the second round using the decomposition in eq. 6.19 the parallel computation cost is reduced from O(2N) to O(N).

All initializations and settings for the Kernel Gradient Subtractive Clustering (KG-SC) algorithm are the same as in [22]. Given N points $\mathbf{x}_n$ each one in d dimension, with their mean vector $\boldsymbol{\mu} = (1/N)\sum_n^N \mathbf{x}_n$ we set the initial bandwidth σ in the beginning of the gradient descent (first epoch) as $(1/d)\sum_i^d \sqrt{r_{ii}}$, where r_{ii} are the diagonal elements of the covariance matrix $\mathbf{R} = (1/N)\sum_n^N (\boldsymbol{\mu} - \mathbf{x}_n)\,(\boldsymbol{\mu} - \mathbf{x}_n)^T$. The data features are scaled into the range [0, 1]. By scaling the data features first, we can then use a fixed value for ξ for all datasets and thus avoiding the search for suitable learning rates each time we use a different dataset. For the gradient descent we set a fixed learning rate $\xi = 0.2$ and maximum epochs = 10. Usually it converges after the first epoch if the dataset size is larger than 10000. Therefore, for the larger datasets we set maximum epochs = 2. For the leave-one-out kernel averaged regression function we set the leave-one-out parameter $\gamma = 0.1$ which is always sufficient enough to prevent bandwidth from converging into tiny values, so as to provide a stable solution. For the settings in Subtractive Clustering (SC) we apply two minor modifications (see also [22]). We use a variable σ_b that equals to $\sigma_b = \sigma_a + 0.5\,(1.0 - k_{\text{sofar}}/N)\,\sigma_a$, which starts from $\sigma_b = 1.5\sigma_a$ and decays. As k_{sofar} (the number of selected exemplars so far) increases one-by-one during the potential $P(i)$ updating cycle of SC, the parameter σ_b gradually decreases and in the theoretical limit k=N the value σ_b becomes equal to σ_a. This strategy works around the problem in higher dimensions where the fixed 1.5 percentage influences more strongly the nearby points. We also set $e = 1/P_1^*$. That is, Subtractive Clustering terminates at j-th iteration when $P_j^* < 1$. Thus, every point starts with potential $P(i) >= 1$ and finally ends up with potential $P(i) < 1$.

With these simple default settings, which are the same for all datasets, no adjustable user-defined parameters are required and the whole process is automatic.

6.3.4 Visual snapshots for exemplar selection with the proposed KG-SC

Figure 6.2 illustrates visual snapshots for the exemplar selection phase via KG-SC.

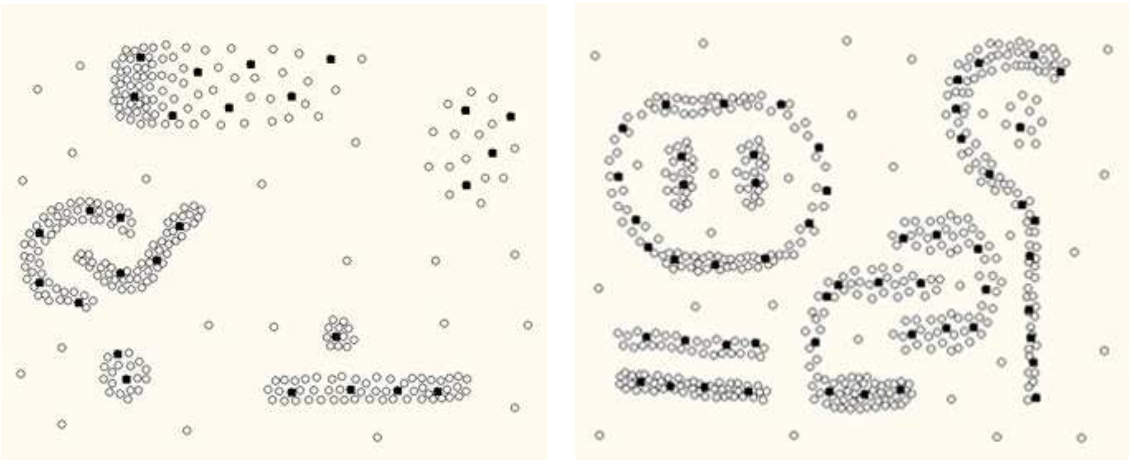


Figure 6.2. Visual snapshots of the proposed KG-SC results. Exemplars are black squares, other points are white circles. In (a) the dataset has 322 points and KG-SC selects 27 exemplars automatically. In (b) the dataset has 523 points and KG-SC selects 55 exemplars automatically.

6.3.5 The final RBFNN training scheme

The final training scheme is:

1) use the leave-one-out Kernel Averaged gradient descent Subtractive Clustering to find the location of the centers $\mathbf{c}_k$ and their number automatically

2) use the previously extracted center set for final RBF training via the mini-batch gradient descent learning.

We demonstrate that all these algorithms can be performed in a parallel ring pipeline that permits each processor node to hold a portion of the RBFNN model together with a portion of the training data.

6.4 Data or Neuron parallel training basics

A computational problem can be decomposed into sub-problems that can be executed in parallel. RBF neural network is intrinsically parallel. In the RBFNN parallel training there are two main decomposition patterns: data decomposition that splits the data into the processors and neuron (task) decomposition that splits the neurons into the processors.

6.4.1 Data Parallel training in Master/Worker

For complex multidimensional problems, the number of training data is usually large, while the number of model parameters, like RBF weights, remains comparatively moderate. To tackle the problem, the Data Parallelism scheme with Master/Worker topology [20] can be adopted. Each Worker processor holds a different partition of data while the Master holds the entire Neural Network architecture. During the training steps the Master processor sends the same RBFNN model parameters to all Workers. Each Worker, by receiving the entire RBFNN copy, partially update the model

parameters using its own data partition, and sends these updates back to the Master which merges the results from all the workers.

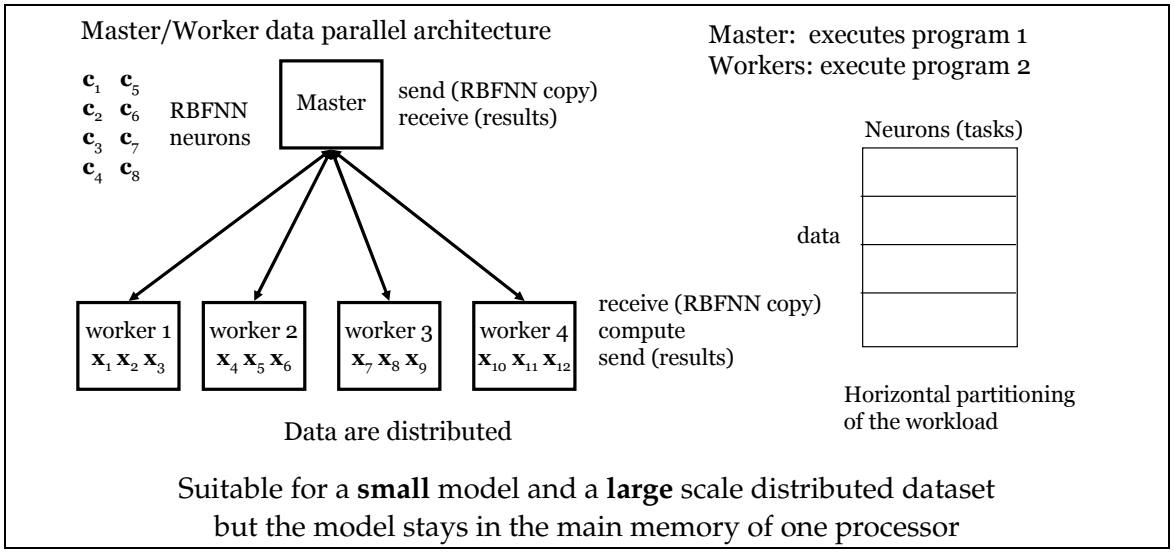


Figure 6.3. In Master/Worker data parallel architecture, the master processor holds the neurons and the workers processors hold the data partitions. This is a classical parallel architecture.

6.4.2 Neuron parallel training in a pipeline

In the Neuron Parallelism approach, neurons are initially distributed among the processor nodes. In the case of RBFNN these are the hidden neuron parameters {centers $\mathbf{c}_k$, widths σ_k, weights w_k}. With large numbers of hidden neurons this method can speed up the computations. A neural network belongs to the synchronous class of problems. The same operation is carried out for each pattern in the training set. Neuron parallelism attacks the training time problem by improving the time to process a single pattern.

In a pipeline topology works the RBF neuron parallel training like the instruction pipeline in parallel systems. Pipeline is a programming pattern mainly for task (neuron) parallelism where the program tasks are partitioned in a sequence of stages, each one implemented at a different processor. Data flows through the pipeline, from the first stage to the last stage. Each stage performs a transform on the data. Parallelism is accomplished by running stages concurrently on subsequent data vectors. In a Pipeline topology [20] the communication time between processors is minimised to point-to-point, allowing each node to receive-send messages with only his 2 neighbours (previous - next). This reduces as much as possible the idle time of processing elements. The pipeline has three kinds of nodes, Source (first node), Seil (internal nodes) and Sink (last node). The dataset remains in the Source which iteratively sends every instance **x** (or block of data) to the next node. Each Seil node concurrently receives **x** and the list of partial results from previous node, then adds its contribution to the partial sums of **x** and finally sends **x** and its list of partial sums to next node. The Sink node receive the list of final sums for each instance **x**.

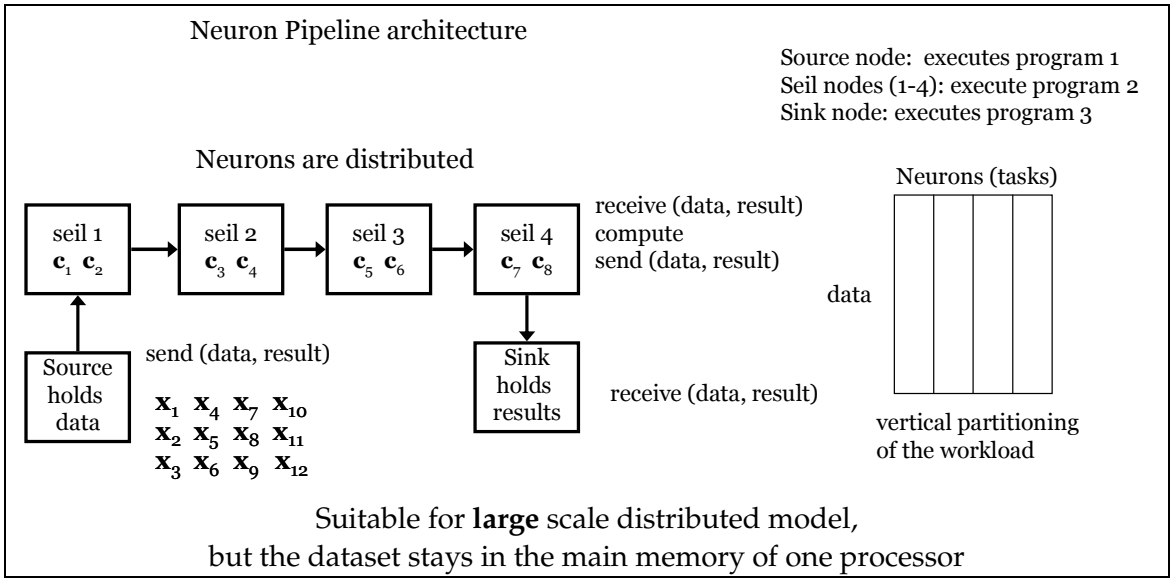


Figure 6.4. In Neuron parallel Pipeline the source holds the data and the other nodes hold the pattern neurons.

In large-scale parallel applications one can try to mix data and neuron parallelism. In order for a hybrid Neural Network parallel scheme to work efficiently, the algorithmic framework must simultaneously allow for efficient data-parallelism as well as for efficient neuron-parallelism.

6.5 Proposed Data and neuron parallel RBFNN in a Ring pipeline

The ring pipeline in fig. 6.5 facilitates both neuron partitioning as well as data partitioning. In contrast to master/worker or pipeline where either data or neurons are partitioned, in the ring pipeline both neurons and data can be held distributed across the processors which by their part are organized in a virtual ring pipeline. The simple virtual ring design that connects the processor nodes is well known for many years [23] [25].

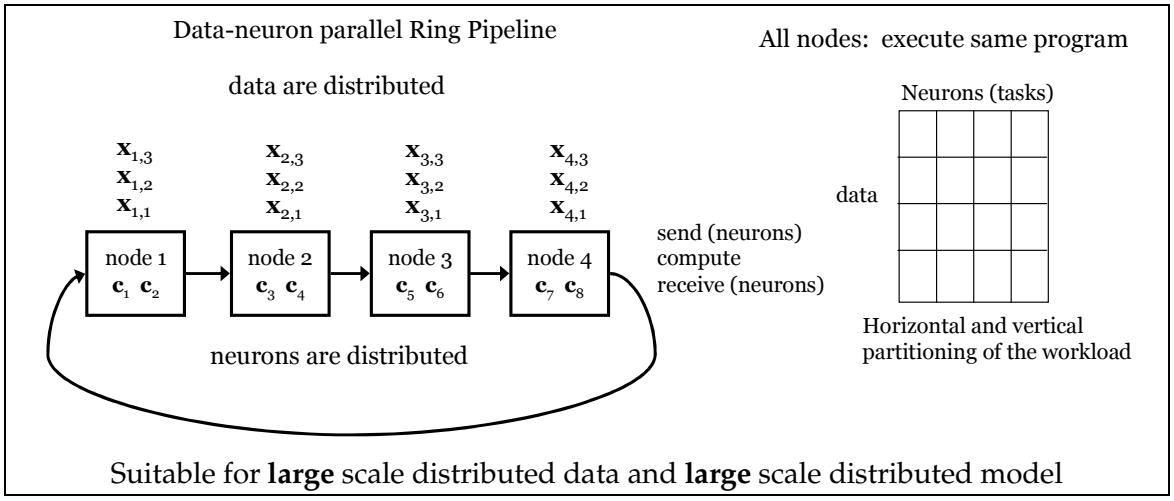


Figure 6.5. In data-neuron ring pipeline (similarly to data-instruction ring pipeline problems [23]) all nodes hold data partitions together with neuron partitions. The {Send-Compute-Receive} pattern that overlaps computation delays with communication delays is also illustrated.

Here we also adopt a {Send-Compute-Receive} pattern, which allows overlapping communications delays with computations delays. This pattern easily makes a data-block available in the queue before the corresponding processor node requests it. However few training algorithms for Neural Networks can be entirely operate in a ring pipeline. The current study demonstrates two such algorithms. The first is a ring pipelined mini-batch parallel version of the recently proposed Kernel Gradient Subtractive Clustering (KG-SC) in order to extract the RBF centers and their number. The second is a ring pipelined parallel mini-batch gradient descent for updating the RBF parameters {centers, widths, output weights}.

6.5.1 Overlapping delays

Ring Pipeline has the advantage that we can use overlapping the communications delays with the computations delays (see chapter 9.2 in [23]) so as to minimize the idle time. In Fig. 6.5 the Send() and Receive() commands have a communication delay and the Compute() command has a computation delay. If the Compute command was the first and the Send() command was second, then the two delays would have been cumulated in the ring. Instead, now they are overlapped. Fig. 6.5 shows the ring pipeline where the {Send-Compute-Receive} pattern is employed as:

- *Send*(*block*)*:* Each node sends a block (with either data or neurons) to its next node.
- *Compute* (*block*): during the data transfer (while network send/receive communications complete in the background) the node continues partial computations with the available data in this block.
- *Receive*(*block*): receives a block from its previous node.

All the blocks are gradually propagated around the ring.

6.5.2 Main program structure

A paradigm for the main program consists of loops of {Send-Compute-Receive} commands that circulate the blocks (data or neurons) through the Ring pipeline. In the {Send-Compute-Receive} sequence the Send and Compute commands use the same block. There are two loops one inside the other. The same program is executed from all nodes as illustrated in fig. 6.6.

```
Every node:      //2 loops
  for cycle = 1 to B                    //for my data blocks
       Send (myBlock_cycle)
       Compute Partial (myBlock_cycle)
       Receive (prevBlock)
       for node = 2 to L                //for other data blocks
           Send (prevBlock)
           Compute Partial (prevBlock)
           Receive (prevBlock)
       end for
       Compute Final (prevBlock)        //finalize computations
  end for                               //my block list of size B becomes empty
```

Figure 6.6. A Ring pipeline program paradigm that circulates the data-blocks through the ring using two loops of many {Send-Compute-Receive} sequences. The first loop (outer) is for my data blocks, the second loop (inner) is for the blocks of the other nodes.

The paradigm in fig. 6.6 is used for all the ring pipeline algorithms.

A simple load balancing is required in order for the data-neuron ring pipelining to work efficiently. All L nodes must hold the same number of data points $N_p = N/L$ points. If the modulo $N\%L$ isn't zero then the remainder points will be distributed one-by-one into the nodes. Hence, some nodes will have one data point less than the others.

The program termination is also simple. As usual all the nodes will group their examples into a predefined number B of blocks, same to all processors. Some nodes will have one more example only in their last block and the block lists in all nodes will have the same size B. The program in fig. 6.6 (the outer loop for cycle…) terminates when this block list becomes empty. Finally all nodes stop simultaneously.

6.5.3 Ring pipelined leave-one-out kernel averaged gradient descent

Online gradient descent uses a single example at a time to compute the gradient. Batch mode computes the gradient by averaging the contribution of all examples. Mini-batch gradient descent uses a mini-batch $\{\mathbf{x}_b, y_b\}_{b=1}^{L}$ and computes the averaged gradient over the $\mathbf{x}_b$ examples in the mini-batch. An epoch ends after all examples are introduced. For one epoch step the *mini-batch leave-one-out Kernel averaged Gradient descent* is:

for $t = 1$ **to** B data blocks

form randomly a mini-batch $\{\mathbf{x}_b, y_b\}_{b=1}^{L}$ of $\mathbf{x}_b$ examples (one from each node)

update the parameter σ by using $\sigma^{(t+1)} = \sigma^{(t)} - \xi^{(t)} \ \frac{1}{L}\sum_{b}^{L} \ \partial E(\sigma^{(t)}, \mathbf{x}_b)/\partial\sigma$

end for

Only mini-batch gradient descent is suitable for ring pipeline parallel mapping.

The Ring pipeline parallel mini-batch KG (Kernel averaged Gradient descent) is illustrated in algorithm 1. The {Send-Compute-Receive} pattern is employed with two loops (outer and inner). The outer loop iterates through the number B of data blocks each node holds. Every node sends a small block of examples $\mathbf{x}_i$ and their parameters into its next node. The inner loop iterates through the number of processor nodes. All processors have the same number B of blocks and terminate simultaneously. In algorithm 1 for clarity reasons the blocks of data contain one example $\mathbf{x}_i$. The same holds for the *send/receive* messages. The generalization to blocks of many examples is straightforward. As illustrated in algorithm 1 there is a mini batch of examples $\mathbf{x}_b$ with their partial sums $\Sigma\varphi_j$, $\Sigma\varphi_j\delta_j$, $\Sigma\varphi_j y_j$ and $\Sigma\varphi_k\delta_k y_k$ that circulates in the ring pipeline. The mini-batch size is a multiple of the number L of processors, which equally supply their examples. During passing throughout the nodes each $\mathbf{x}_b$ continues to sum up the local kernels contributions by using $\Sigma\varphi_j = \Sigma\varphi_j + \varphi_j(\mathbf{x}_b)$, $\Sigma\varphi_j\delta_j = \Sigma\varphi_j\delta_j + \varphi_j(\mathbf{x}_b) \cdot \delta_j(\mathbf{x}_b)$, $\Sigma\varphi_j y_j = \Sigma\varphi_j y_j + \varphi_j(\mathbf{x}_b) \cdot y_j$ and $\Sigma\varphi_k\delta_k y_k = \Sigma\varphi_k\delta_k y_k + \varphi_k(\mathbf{x}_b) \cdot \delta_k(\mathbf{x}_b) \cdot y_k$ until it computes the total sums, and finally arrives at its origin processor from which it had began circulating. Then this processor that holds $\mathbf{x}_b$ will also have the finalized sums $\Sigma\varphi_j$, $\Sigma\varphi_j\delta_j$, $\Sigma\varphi_j y_j$, $\Sigma\varphi_k\delta_k y_k$ from which it will compute the leave-one-out kernel averaged regression function [22] as $f_{loo}(\mathbf{x}_b,\gamma) = \Sigma\varphi_j y_j/\Sigma\varphi_j - \gamma \cdot y_b/\Sigma\varphi_j$, It will also compute the residual error $e_b = (f_{loo}(\mathbf{x}_b,\gamma) - y_b)$, the derivate $f_{loo}(\mathbf{x}_b,\gamma)/\partial\sigma = \Sigma\varphi_j\delta_j y_j / \ \Sigma\varphi_j - \Sigma\varphi_j\delta_j \cdot \Sigma\varphi_j y_j / \ (\Sigma\varphi_j)^2 + \gamma \cdot y_b \cdot (\Sigma\varphi_j\delta_j)/(\Sigma\varphi_j \cdot \Sigma\varphi_j)$ and the gradient of the squared error $\partial E_{\text{local}} = \partial E(\sigma^{(t)}, \mathbf{x}_b)/\partial\sigma$ for this $\mathbf{x}_b$ in the mini-batch. After that all processors will merge those $\partial E_{\text{local}}$ to form the sum ∂E_{global} for the mini-batch.

Algorithm 1: Ring pipeline parallel mini-batch gradient descent of the leave-one-out kernel averaged (one epoch). The number of processor nodes is L. Each node has N_{local} data. *my_rank* is the number of the node that runs the algorithm in the ring pipeline.

Every Node: //computes $\sigma^{(t+1)} = \sigma^{(t)} - \xi^{(t)} \ (1/L)\sum_b^L \ \partial E(\sigma^{(t)}, \mathbf{x}_b)/\partial\sigma$

next = Next(*my_rank*), $t = 1$

Random Shuffle the order of my local data list of $\{\mathbf{x}_i, y_i\}_{i=1}^{N_{local}}$

for *cycle* = 1 **to** B //for each one of my data blocks

$\Sigma\varphi^{next} = 0$, $\Sigma\varphi\delta^{next} = 0$, $\Sigma\varphi y^{next} = 0$, $\Sigma\varphi\delta y^{next} = 0$, $i = cycle$

Send ($\mathbf{x}_i$, *my_rank*, $\Sigma\varphi^{next}$, $\Sigma\varphi\delta^{next}$, $\Sigma\varphi y^{next}$, $\Sigma\varphi\delta y^{next}$) //send my data point $\mathbf{x}_i$

Compute $s\varphi_{\text{local}}$, $s\varphi\delta_{\text{local}}$, $s\varphi y_{\text{local}}$, $s\varphi\delta y_{\text{local}}$ using ($\mathbf{x}_i$, $\sigma^{(t)}$))

Receive ($\mathbf{x}$, *rank*, $\Sigma\varphi$, $\Sigma\varphi\delta$, $\Sigma\varphi y$, $\Sigma\varphi\delta y$)

for *node* = 2 **to** L //for each one of the other nodes

$\Sigma\varphi=\Sigma\varphi+s\varphi_{\text{local}}$, $\Sigma\varphi\delta=\Sigma\varphi\delta+s\varphi\delta_{\text{local}}$, $\Sigma\varphi y=\Sigma\varphi y+s\varphi y_{\text{local}}$, $\Sigma\varphi\delta y=\Sigma\varphi\delta y+s\varphi\delta y_{\text{local}}$

Send ($\mathbf{x}$, *rank*, $\Sigma\varphi$, $\Sigma\varphi\delta$, $\Sigma\varphi y$, $\Sigma\varphi\delta y$)

Compute $s\varphi_{\text{local}}$, $s\varphi\delta_{\text{local}}$, $s\varphi y_{\text{local}}$, $s\varphi\delta y_{\text{local}}$ using ($\mathbf{x}$, $\sigma^{(t)}$))

Receive ($\mathbf{x}$, *rank*, $\Sigma\varphi$, $\Sigma\varphi\delta$, $\Sigma\varphi y$, $\Sigma\varphi\delta y$)

end for

$\Sigma\varphi=\Sigma\varphi+s\varphi_{\text{local}}$, $\Sigma\varphi\delta=\Sigma\varphi\delta+s\varphi\delta_{\text{local}}$, $\Sigma\varphi y=\Sigma\varphi y+s\varphi y_{\text{local}}$, $\Sigma\varphi\delta y=\Sigma\varphi\delta y+s\varphi\delta y_{\text{local}}$

//Close the loop

Send ($\Sigma\varphi$, $\Sigma\varphi\delta$, $\Sigma\varphi y$, $\Sigma\varphi\delta y$)

Receive ($\Sigma\varphi$, $\Sigma\varphi\delta$, $\Sigma\varphi y$, $\Sigma\varphi\delta y$)

//Find the leave-one-out kernel averaged regression function (see eq. 6.8)

$f_{loo}(\mathbf{x}_i,\gamma) = \Sigma\varphi y/\Sigma\varphi - \gamma \cdot y_i/\Sigma\varphi$

//Find the derivate of $f_{loo}(\mathbf{x}_i,\gamma)$ with respect to σ (see eq. 6.18 and eq. 6.19)

$f_{loo}(\mathbf{x}_i,\gamma)/\partial\sigma = \left(\Sigma\varphi\delta y/\Sigma\varphi - \Sigma\varphi\delta \cdot \Sigma\varphi y/(\Sigma\varphi)^2 + \gamma \cdot y_i \cdot (\Sigma\varphi\delta)/\ (\Sigma\varphi)^2 \right) / \ (\sigma^{(t)})^3$

// Find the gradient of the squared error (see eq. 6.12)

$\partial E_{\text{local}} = \partial E(\sigma^{(t)}, \mathbf{x}_i)/\partial\sigma = (f_{loo}(\mathbf{x}_i,\gamma) - y_i)\ (\partial f_{loo}(\mathbf{x}_i,\gamma)\ /\partial\sigma)$

//here $\partial E_{\text{local}} = \frac{1}{\sigma^3}\left\{\sum_k^N\left(\left(\delta_k - \frac{\Sigma(\varphi_j\delta_j)}{\Sigma\varphi_j}\right)\cdot\frac{\varphi_k}{\Sigma\varphi_j}y_k\right) + \gamma\frac{\Sigma(\varphi_j\delta_j)}{(\Sigma\varphi_j)^2}y_i\right\}$ for my local point $\mathbf{x}_i$

Reduce $\partial E_{\text{local}}$ into $\partial E_{\text{global}}$ // $\partial E_{\text{global}}$ is the sum of all local $\partial E_{\text{local}}$

Broadcast $\partial E_{\text{global}}$

$\sigma^{(t+1)} = \sigma^{(t)} - \xi^{(t)}\ \partial E_{\text{global}}\ /\ L$ //find the new σ

$t = t + 1$

end for // my block list of size B becomes empty

Function Next(*node*) : **if** (*node*==L) **return** 1 **else return** *node*+1 **end if**.

Function Compute $s\varphi_{\text{local}}$, $s\varphi\delta_{\text{local}}$, $s\varphi y_{\text{local}}$, $s\varphi\delta y_{\text{local}}$ using ($\mathbf{x}$, $\sigma^{(t)}$) :

first compute the vector $\boldsymbol{\delta}(\mathbf{x}) = [\delta_1(\mathbf{x}) \ldots. \delta_n(\mathbf{x}) \ldots. \delta_{N_{local}}(\mathbf{x})]$

$s\varphi_{\text{local}} = \sum_{n=1}^{N_{local}} \varphi_n(\mathbf{x})$ where $\delta_n(\mathbf{x}) = \|\mathbf{x}_n - \mathbf{x}\|^2$

$s\varphi\delta_{\text{local}} = \sum_{n=1}^{N_{local}} \varphi_n(\mathbf{x})\ \delta_n(\mathbf{x})$ $\varphi_n(\mathbf{x}) = \exp(-\delta_n(\mathbf{x})\ /\ (\sigma^{(t)})^2)$

$s\varphi y_{\text{local}} = \sum_{n=1}^{N_{local}} \varphi_n(\mathbf{x})\ y_n$

$s\varphi\delta y_{\text{local}} = \sum_{n=1}^{N_{local}} \varphi_n(\mathbf{x})\ \delta_n(\mathbf{x})\ y_n$

The circulation is illustrated in fig. 6.7 for one ring pipeline cycle.

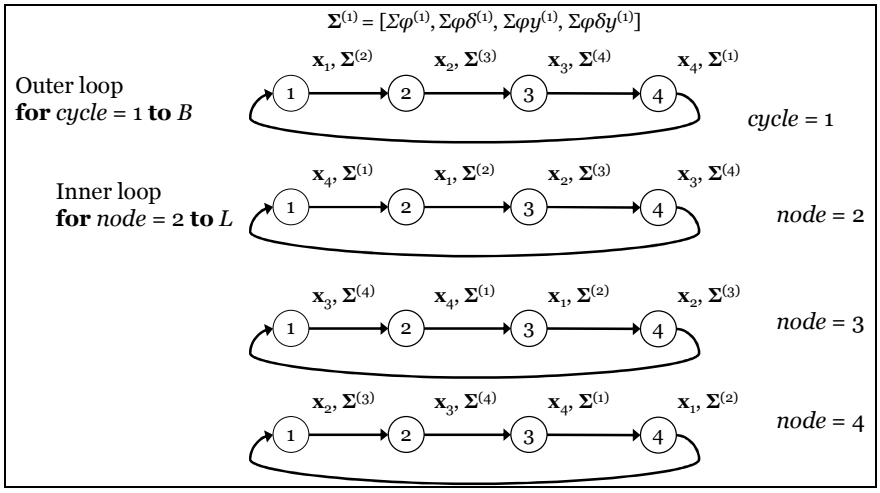


Figure 6.7. Circulating a mini-batch of points $\mathbf{x}_i$ and sums $\Sigma\varphi$, $\Sigma\varphi\delta$, $\Sigma\varphi y$, $\Sigma\varphi\delta y$. Each such sum accumulates the results of the point that precedes it in a round robin fashion. All nodes send the center to the next node. The pipeline is always full.

6.5.4 Ring pipeline parallel subtractive clustering

The ring pipeline Subtractive Clustering uses the previously found bandwidth σ_a from algorithm 1 for computing the potentials $P(i)$, is illustrated in algorithm 2. First the dataset is equally partitioned into the processors which must find and store all the potentials $P(i)$ for their local data points $\mathbf{x}_i$. Following the {send-compute-receive} pattern every processor node Sends() a small block of data into its next node, then Computes PartialPotentials() with the data in this block, and then Receive() from the queue buffer the block that has been arrived from the previous node. For clarity reasons in the computations the send/receive messages in algorithm 2 use one data point $\mathbf{x}$. The generalization to messages that have blocks of several points is straightforward.

Algorithm 2: Ring pipeline in parallel Subtractive clustering for computing the potentials $P(i)$:

Every Node:
Input: number B of data blocks, parameter $a = (2/\sigma_a)^2$, number L of processors
Output: potentials $P(i)$ of my local data
for *cycle* = 1 **to** B //for each one of my data blocks
Send ($\mathbf{x}_{cycle}$) // here each cycle sends one point in the message
Compute PartialPotentials ($\mathbf{x}_{cycle}$)
Receive ($\mathbf{x}_{prev}$)
for *node* = 2 **to** L //for each one of the other nodes
Send ($\mathbf{x}_{prev}$)
Compute PartialPotentials ($\mathbf{x}_{prev}$)
Receive ($\mathbf{x}_{prev}$) // the last received message contains my data block
end for
end for //here my block list of size B becomes empty

Function Compute PartialPotentials ($\mathbf{x}$) :
for each $\mathbf{x}_i$ in my local Data Partition **do**
$P(i) = P(i) + exp(-a\,||\mathbf{x}_i - \mathbf{x}||^2)$
end for

At the end of algorithm 2 every node in the ring pipeline will hold the potentials $P(i)$ for the $\mathbf{x}_i$ points of its own data partition.

Algorithm 3 presents the revision of the potentials and the selection of centers. Algorithm 3 repeatedly executes the following cycle: (1) every node finds its best local point $\mathbf{x}^*_{local}$ with the highest local potential value P^*_{local}, (2) These pairs $\{\mathbf{x}^*_{local}, P^*_{local}\}$ are communicated through the ring pipeline. (3) Every node finds among these locally best pairs the pair $\{\mathbf{x}^*, P^*\}$ with the highest potential $\{\mathbf{x}^*, P^*\}$. (4) Every node performs local computations in order to revise the potentials of its own local points by using $P(i) = P(i) - P^*\exp(-b\|\mathbf{x}^* - \mathbf{x}_i\|^2)$. Here the costly computations are those of the 4th step. These are concurrently executed in parallel.

Algorithm 3: Ring pipeline in parallel Subtractive clustering for revising the potentials

Every Node:
Input: potentials $P(i)$, my local data partition, parameter b
Output: local centers

Find my local $\mathbf{x}^*_{local}$ with the highest local potential P^*_{local}
repeat
Send $(\mathbf{x}^*_{local}, P^*_{local})$
Set $P^* = P^*_{local}$, $\mathbf{x}^* = \mathbf{x}^*_{local}$
Receive $(\mathbf{x}^*_{prev}, P^*_{prev})$
for *node* = 2 **to** L //for each one of the other nodes
Send $(\mathbf{x}^*_{prev}, P^*_{prev})$
if $P^*_{prev} > P^*$ **then** set $P^* = P^*_{prev}$, $\mathbf{x}^* = \mathbf{x}^*_{prev}$ //update current max P^*
endif
Receive $(\mathbf{x}^*_{prev}, P^*_{prev})$
end for
Revise potentials $P(i)$ using $(\mathbf{x}^*, P^*)$ //finalize computations
Find my next local $\mathbf{x}^*_{local}$ with the highest local potential P^*_{local}
until $(P^* <= 1)$

Function Revise potentials $P(i)$ using $(\mathbf{x}^*, P^*)$:
for each $\mathbf{x}_i$ in my data partition **do**
$P(i) = P(i) - P^*\exp(-b\|\mathbf{x}_i - \mathbf{x}^*\|^2)$
end for

After performing the algorithms 1, 2 and 3 in the Ring pipeline for finding the centers and their number, then the RBF gradient descent training algorithm refines all the parameters.

6.5.5 *Ring pipeline parallel Mini-batch gradient descent learning of RBFNN parameters*

Algorithm 4 shows one epoch of the Ring pipeline parallel mini-batch gradient descent updating of the RBF parameters. It again adapts the {Send-Compute-Receive} pattern. There are two loops (outer and inner). The outer loop iterates through the number B of data blocks each node holds and sends through the pipeline. The inner loop iterates through the number L of nodes. An epoch terminates when this data block list is empty.

Every node sends a small block of examples $\mathbf{x}_i$ into its next node. For the sake of clarity the data blocks in algorithm 4 contain one example $\mathbf{x}_i$. The same holds for the *send/receive* messages. The circulating mini-batch is composed from all the examples in the messages that circulate in the pipeline. The generalization to blocks of many examples is straightforward.

The circulation is illustrated in fig. 6.8

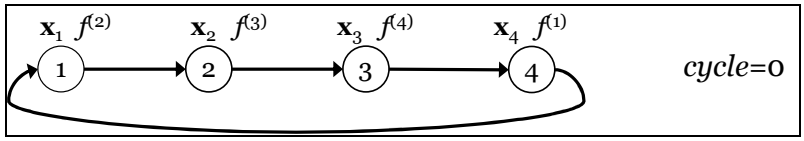


Figure 6.8. Circulating a mini-batch of points $\mathbf{x}_i$ and sums f. Each such sum accumulates the results of the point that precedes it in a round robin fashion.

At cycle t the mini batch of points $\mathbf{x}_i$ with their partial sums f_i circulate in the ring pipeline. Here the mini batch size is L which equals the number of processors. The partial sum for one point $\mathbf{x}_i$ in such a batch is $f_i = \sum_k^{NeuronsSoFar} w_k \cdot \varphi_k(\mathbf{x}_i)$. During passing throughout all nodes it continues to sum up the contributions from the local RBF units until it computes the total output sum $f(\mathbf{x}_i)$. Then the node that holds $\mathbf{x}_i$ will compute the residual $e_i = (y_i - f(\mathbf{x}_i))$. In the second round each processor node sends the residual e_i so as the other nodes which hold a part of the RBF parameters find the local gradients via the summations and update their local parameters.

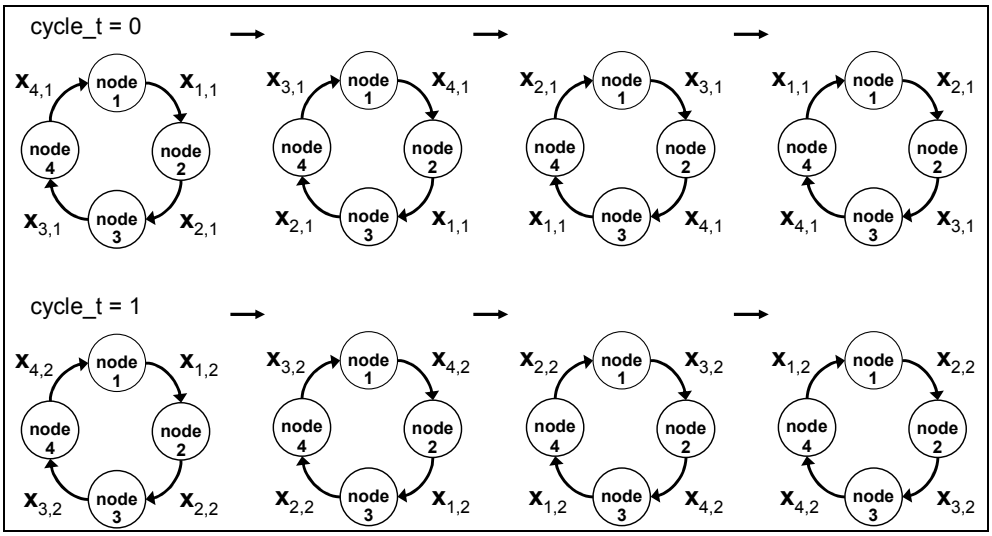


Figure 6.9. In a new cycle t all nodes send their new point $\mathbf{x}_t$ to the next node. The next cycle starts instantly after the first cycle ends. All nodes will terminate simultaneously. The pipeline is always full.

Algorithm 4: Ring pipeline parallel mini-batch RBF gradient descent (one epoch). The number of processor nodes is L. Each node has N_{local} data and K_{local} RBFs. *my_rank* is the number of the processor node in the ring pipeline.

Every Node:

Input: number B of my training data blocks, my local data partition $\{\mathbf{x}_i, y_i\}_{i=1}^{N_{local}}$

my local RBF parameters {centers $\mathbf{c}_k$, widths σ_k, weights w_k} $_{k=1}^{Klocal}$

Output: mini-batch RBF gradient descent updates of my local RBF parameters

next = Next(*my_rank*), *cycle* = 1

Random Shuffle the order of my local data list of $\{\mathbf{x}_i, y_i\}_{i=1}^{N_{local}}$

for *cycle* = 1 **to** B

//1st round

$f^{next} = 0$, $i = cycle$, $t = 1$

Send ($\mathbf{x}_i$, *my_rank*, f^{next}) // send my data point $\mathbf{x}_i$ and the f of next point

Compute f_{local} and cache distances using ($\mathbf{x}_i$, *my_rank*)

Receive ($\mathbf{x}$, *rank*, f) // this is the f of my $\mathbf{x}_i$

for *node* = 2 **to** L // for each one of the other nodes

$f = f + f_{\text{local}}$

Send ($\mathbf{x}$, *rank*, f)

Compute f_{local} and cache distances using ($\mathbf{x}$, *rank*)

Receive ($\mathbf{x}$, *rank*, f)

end for

$f = f + f_{\text{local}}$

Send (f)

Receive (f) //Close the loop by receiving the f of my $\mathbf{x}_i$

$f(\mathbf{x}_i) = f$

$e_i = y_i - f(\mathbf{x}_i)$ // find residual

//2nd round

// for summing the mini-batch contributions to the gradients

$p = my\_rank$, $e_p = e_i$

Send (*my_rank*, e_p)

Compute Additions to *LocalRBFgradients* using (*my_rank*, e_p)

Receive (*rank*, e)

for *node* = 2 **to** L //for each one of the other nodes

Send (*rank*, e)

Compute Additions to *LocalRBFgradients* using (*rank*, e)

Receive (*rank*, e)

end for

Update my local RBF parameters ($\xi^{(t)}$, L)

$t = t + 1$

end for //my local data list becomes empty

Function Next(*node*) : **if** (node==L) **return** 1 **else return** *node*+1.

Function Compute f_{local} and cache distances using ($\mathbf{x}$, *rank*):

// these steps are exactly the same with that of the sequential algorithm

Store the cached matrix of vector differences of my centers $\mathbf{c}_k$ from $\mathbf{x}$

$\mathbf{M}_{rank}(\mathbf{x}) = [\boldsymbol{\mu}_1 \ldots \boldsymbol{\mu}_k \ldots \boldsymbol{\mu}_{K_{local}}] = [(\mathbf{x}-\mathbf{c}_1) \ldots (\mathbf{x}-\mathbf{c}_k) \ldots (\mathbf{x}-\mathbf{c}_{K_{local}})]$

Store the cached vector of squared Euclidean distances

$\boldsymbol{\delta}_{rank}(\mathbf{x}) = [\delta_1(\mathbf{x}) \ldots \delta_k(\mathbf{x}) \ldots \delta_{K_{local}}(\mathbf{x})]$ where $\delta_k(\mathbf{x}) = \|\mathbf{c}_k - \mathbf{x}\|^2$

Store the cached vector of RBF responces

$\boldsymbol{\varphi}_{rank}(\mathbf{x}) = [\varphi_1(\mathbf{x}) \ldots \varphi_k(\mathbf{x}) \ldots \varphi_{K_{local}}(\mathbf{x})]$ where $\varphi_k(\mathbf{x}) = \exp(-\delta_k(\mathbf{x}) \,/\, (\sigma_k)^2)$

$$f_{\text{local}} = \sum_{k=1}^{K_{local}} w_k \varphi_k(\mathbf{x})$$

Function Compute Additions to *LocalRBFgradients* using (*rank*, *e*) :

// these steps are exactly the same with that of the sequential algorithm

// for summing the mini-batch contributions to the gradients

recall from memory the cached vector

$\boldsymbol{\delta}(\mathbf{x}_{rank}) = [\delta_1(\mathbf{x}_{rank}) \ldots \delta_k(\mathbf{x}_{rank}) \ldots \delta_{K_{local}}(\mathbf{x}_{rank})]$

recall from memory the cached matrix

$\mathbf{M}(\mathbf{x}_{rank}) = [\boldsymbol{\mu}_1 \ldots \boldsymbol{\mu}_k \ldots \boldsymbol{\mu}_{K_{local}}] = [(\mathbf{x}_{rank}-\mathbf{c}_1) \ldots (\mathbf{x}_{rank}- \mathbf{c}_k) \ldots (\mathbf{x}_{rank}- \mathbf{c}_{K_{local}})]$

// for each of my local RBFs (k=1 to K_{local})

for $k = 1$ **to** K_{local}

$\delta_k(\mathbf{x}) = \delta_k(\mathbf{x}_{rank})$, $\boldsymbol{\mu}_k = \mathbf{x}_{rank} - \mathbf{c}_k$

$\varphi_k(\mathbf{x}) = \exp(-\delta_k(\mathbf{x}) \,/\, (\sigma_k)^2)$

$\partial E / \partial \sigma_k = \partial E / \partial \sigma_k + e \cdot \varphi_k(\mathbf{x}) \cdot \delta_k(\mathbf{x})$

$\partial E / \partial w_k = \partial E / \partial w_k + e \cdot \varphi_k(\mathbf{x})$

for $d = 1$ **to** D

$\partial E / \partial \mathbf{c}_k(d) = \partial E / \partial \mathbf{c}_k(d) + e \cdot \varphi_k(\mathbf{x}) \cdot \boldsymbol{\mu}_k(d)$

end for

end for

Function Update my local RBF parameters ($\xi^{(t)}$, L):

// these steps are exactly the same with that of the sequential algorithm

for $k = 1$ **to** K_{local}

$\Delta\sigma_k = 2\, \xi^{(t)} \cdot w_k / (\sigma_k)^3 \cdot (1/L) \cdot \partial E / \partial \sigma_k$

$\Delta w_k = \xi^{(t)} \cdot (1/L) \cdot \partial E / \partial w_k$

for $d = 1$ **to** D

$\Delta \mathbf{c}_k(d) = 2\, \xi^{(t)} \cdot w_k / (\sigma_k)^2 \cdot (1/L) \cdot \partial E / \partial \mathbf{c}_k(d)$

end for

end for

In essence by using the {send-compute-receive} sequence the footprint of parallelization on the program structure and data flow is kept at a minimum.

The first advantage of this ring pipeline is the simplicity. Minimization into point-to-point communications only is the most efficient among the other types in parallel processing. The next advantage is that in every step the communication buffer in each processor that receives the messages from the previous node has all the information data available before the processor requests them.

6.6 Experimental Simulations

This section presents experimental simulations for the accuracy and parallel efficiency of the proposed algorithms. For the implementation of parallel mappings in Cluster of workstations, with distributed memory, we have coded all parallel programs in C language using the Message Passing Interface (MPI) library [18] for the communications between the processors. MPI complements standard computer languages with information distribution commands and does not depend from the hardware that supports having thus availability on almost all platforms. Almost all modern supercomputers support MPI, since large scale applications use it either directly or indirectly. The ring pipeline algorithms we present use the MPI send and receive primitives.

6.6.1 Regression results and comparisons

In the experimental simulations we use benchmark datasets for regression in order to test the accuracy of the proposed method. The input features have been normalized into the range [-1, 1] while the outputs have been normalized into the range [0, 1]. Each dataset is randomly split into a training set (50%) and a test set (50%). The learning procedure is repeated 20 times for each dataset and the results are averaged.

Table 6.1 Benchmark datasets

dataset	instances	features
Abalone	4177	8
Bank	8192	8
California Housing	20640	8
Census (house8L)	22784	8
Computer activity	8192	12
Delta Ailerons	7129	5

We measure the regression performance using the root mean squared error (RMSE). We compare the proposed RBFNN method with Back-Propagation (BP), Support Vector Regression (SVR), Extreme Learning Machine (ELM) [41] trained with regularized least squares and the basic Radial Basis Function Neural Network [2]. The ELM algorithm is trained with regularized least squares, the basic RBFNN is trained by clustering and regularized least squares [2] using the same number of RBF units as the proposed method. The comparison results are illustrated in fig. 6.10.

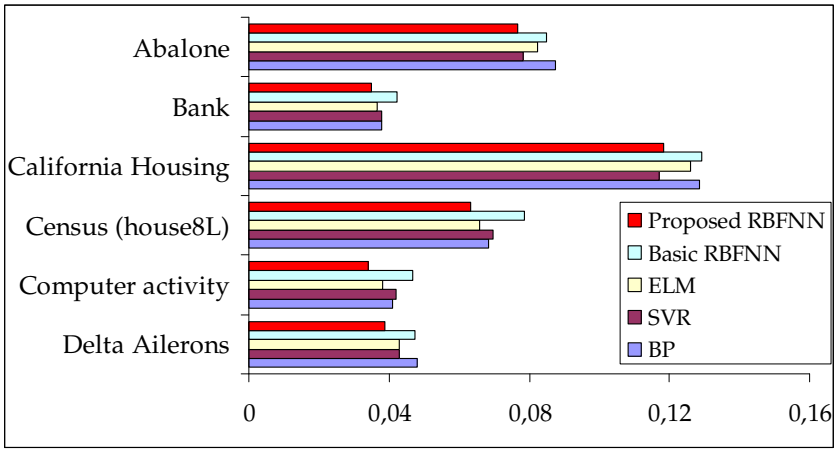


Figure 6.10. Root Mean Squared Error (RMSE) of the test set, averaged for 20 runs

6.6.2 Speedup Measurements

In this section we present the simulation results of the proposed ring pipeline parallel RBFNN training. The target platform for our study is a computer cluster that consisted of distributed memory processors each with 2GB memory, 2.5 GHz CPU and Linux operating system. All processors are interconnected with 1000 Mbps Ethernet network. We evaluate the strong scaling by fixing the number of data points. Two performance measures are used. The first is *Speedup* and the second is *Efficiency*. The *Speedup* is given by the ratio $Speedup(P) = T_{seq}/T_{parallel}$, where T_{seq} and $T_{parallel}$ are the sequential and the parallel computation times, in 1 and P processors respectively. *Speedup* evaluates the strong scaling and finds out sequential bottlenecks. Normally, *Speedup(P)* is smaller than P, and is called sub-linear, while ideally *Speedup(P)* is equal to P, and is called linear. The *Efficiency* is defined by the ratio $Speedup(P)/P$. Hence *Efficiency* measures the usage of the computational resources by computing the fraction of time between performance and the resources.

The speedups reached by the ring pipelined parallel algorithms are shown in figs 6.11, 6.12 and 6.13. Based on the speedup results, it is clear that the speedup is progressively improved on increasing the dataset size.

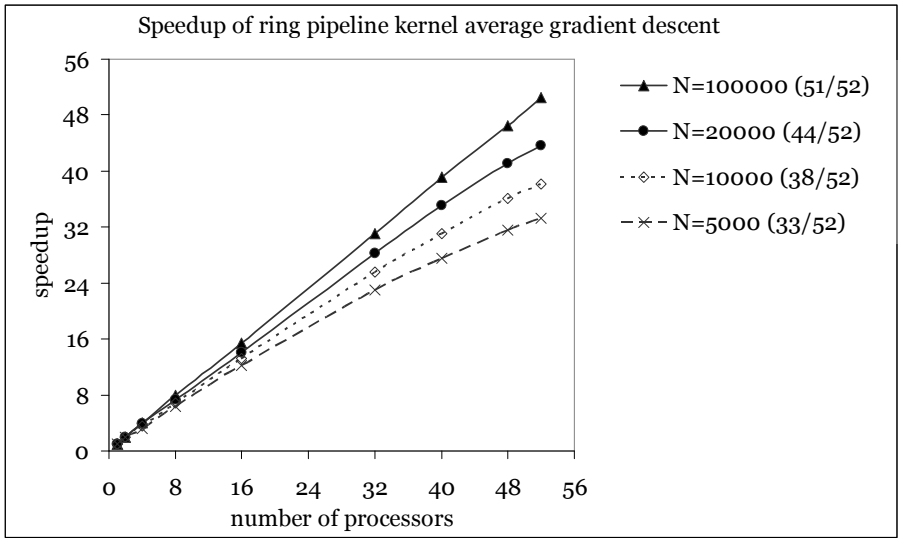


Figure 6.11. The Ring pipeline parallel leave-one-out Kernel Average Gradient Descent *Speedup* curves the number of processors for various data sizes. For each data size the efficiency value is in the parenthesis.

Fig. 6.11 illustrates the Speedup curves with their Efficiency values obtained for the ring pipeline parallel leave-one-out Kernel Average Gradient Descent. The datasets are ranked based on their efficiency. For each dataset the corresponding *Efficiency* value for P=52 processors is indicated in the parenthesis. While for small dataset sizes the curves are clearly sub-linear, when the data size increases the curves move closer to linear and the efficiencies move closer to one.

After ring pipeline parallel Kernel Average Gradient Descent the speedup curves for the subtractive clustering that follows are shown in fig. 6.12. We can notice from their ranking that the curves reveal improvements for larger sizes for which the speedup curves approach the linear and the efficiencies move closer to one.

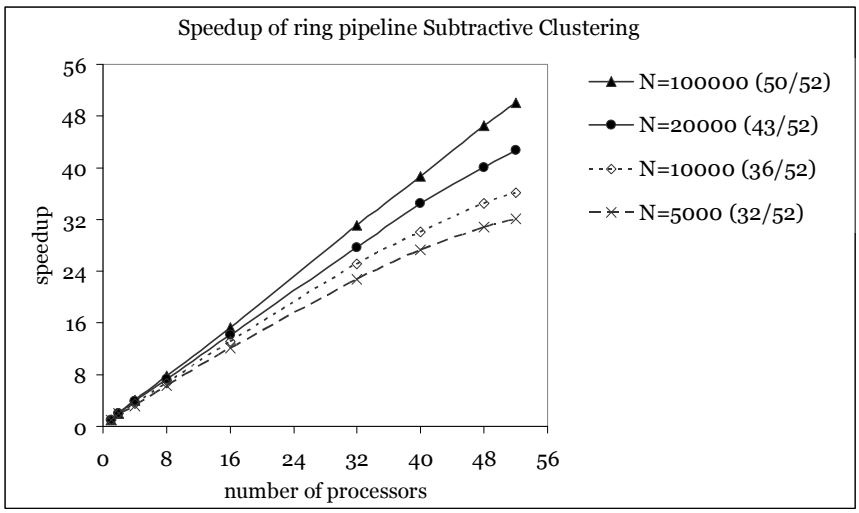


Figure 6.12. Speedup of the ring pipeline parallel Subtractive Clustering. For each dataset the corresponding *Efficiency* value for *P*=52 processors is indicated in the parenthesis

Fig. 6.13 shows the simulation results for the ring pipeline parallel mini-batch RBFNN gradient descent parameter learning. Again the datasets are ranked based on their efficiency. While for the moderate size datasets the curves are evidently sub-linear, on increasing the data size the speedup curves move closer to the ideal case, and the efficiencies move closer to one.

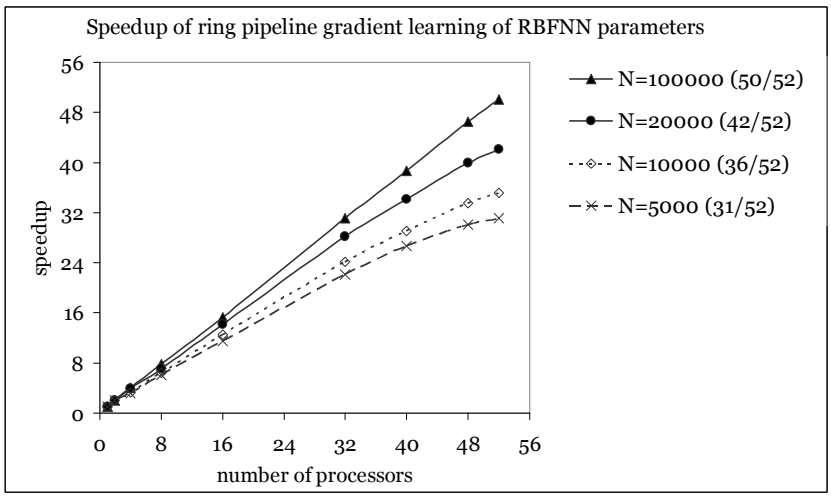


Figure 6.13. Speedup curves for the ring pipeline parallel mini-batch gradient descent learning of RBFNN parameters. For each dataset the corresponding *Efficiency* value for *P*=52 processors is indicated in the parenthesis

6.7 Summary and future work

For constructing Radial Basis Function Neural Networks we proposed a new learning scheme, and additionally show how this scheme can be implemented efficiently in a ring pipeline parallel architecture. In order to automatically find the most representative RBF centers from the training data the scheme uses the recently proposed KG-SC algorithm which does not require a-priori knowledge for the number of centers. After acquiring the network structure the scheme continues with the mini-batch gradient descent learning of the RBFNN parameters. We demonstrate how these algorithms can be efficiently mapped in a parallel ring pipeline. In contrast to master/worker or pure pipeline where either data or neurons are partitioned, in the

ring pipeline both neurons and data can be held distributed across the processors which by their part are organized in a virtual ring pipeline. Hence, there is an additional advantage for data storage since the data portions together with the neuron portions are distributed across the ring. This strategy is suitable for a large scale distributed dataset and a large scale distributed RBF model and permits to circulate on demand either data or neurons. We overcome the problems of implementing the mini-batch gradient descent in a pipeline by using the ring pipeline. We also show that in the Ring Pipeline one can straightforwardly use the {Send-Compute-Receive} pattern that permits overlapping computation delays with communication delays. The same pattern is used for all the algorithms. Experiments with the parallel implementations reveal speedups close to linear on increasing the number of processors. The proposed scheme is also scalable on increasing the size of the datasets.

KG-SC returns also the ranking of the most representative exemplars, in an ordered set from the most important to the least important. It is possible to proceed incrementally. First we can perform the modified Gram-Schmidt orthogonalisation procedure on these exemplars, using the same order, and ensure that each new column added to the design matrix of the growing subset is orthogonal to all previous columns. Thus by applying orthogonalisation future work could explore this direct combination of KG-SC with the orthogonal least squares versions.

References for chapter 6

[1] Bishop C. M. (1995) Neural Networks for Pattern Recognition, Oxford University Press.

[2] Hu Y.H., Hwang J.N. (2002) Handbook of neural network signal processing, CRC Press.

[3] Wang L., and Fu X., (2005) Data Mining with Computational Intelligence. Springer-Verlag.

[4] Haykin S. (2009) Neural networks and learning machines. New York, Prentice Hall.

[5] Wu, Y., Wang, H., Zhang, B., & Du, K.-L. (2012). Using radial basis function networks for function approximation and classification. ISRN Applied Mathematics.

[6] Gutierrez P.A., Hervas-Martinez C. Martinez- Estudillo F.J. (2011) Logistic regression by means of evolutionary radial basis function neural networks. IEEE Transactions on Neural Networks, 22(2) 246-263.

[7] Yu H., Xie T., Paszczynski S., Wilamowski B.M. (2011) Advantages of radial basis function networks for dynamic system design. IEEE Transactions on Industrial Electronics 58, 5438-5450.

[8] Kokkinos Y., Margaritis K.G. (2015) Topology and simulations of a Hierarchical Markovian Radial Basis Function Neural Network classifier. Information Sciences 294, 612–624

[9] Karayiannis N., Randolph-Gips M., (2003) The Construction and Training of Reformulated Radial Basis Function Neural Networks. IEEE Transactions on Neural Networks, 4, 835-844.

[10] Karayiannis N.B. and Mi G.W. (1997) Growing Radial Basis Neural Networks: Merging Supervised and Unsupervised Learning with Network Growth Techniques, IEEE Transactions on Neural Networks, 8(6), 1492–1506.

[11] Zhu Q., Cai, Y., and Liu, L. (1999) A global learning algorithm for a RBF network. Neural Networks, 12, 527-540.

[12] Uykan, Z., Gzelis, C., Celebei, M. E., and Koivo, H. N. (2000) Analysis of Input Output Clustering for Determining Centers of RBFN. IEEE Transactions on Neural Networks, 11(4), 851-858.

[13] Sarimveis H., Alexandridis A., and Bafas G. (2003) A fast training algorithm for RBF networks based on subtractive clustering. Neurocomputing, pp. 501–505.

[14] Zarita Zainuddin, Lim Eng Aik & CalenWoi, Subtractive Clustering for training Radial Basis Function Networks, ICOQSIA 2005 Conference Proceedings, UUM, Malaysia, 2005.

[15] Yang P., Zhu Q., and Zhong X., (2009) Subtractive Clustering Based RBF Neural Network Model for Outlier Detection. Journal of Computers, 4(8), pp.755-762.

[16] Karayiannis B. (1999) Reformulated Radial Basis Neural Networks Trained by Gradient Descent. IEEE Transaction on Neural Networks, 10(3), 657–671.

[17] Alexandridis A., Sarimveis H., Bafas G. (2003) A new algorithm for online structure and parameter adaptation of RBF networks. Neural Networks, 16(7), 1003-1017.

[18] Pacheco P. (1997) Parallel Programming with MPI. Morgan Kaufmann, San Francisco, CA.

[19] Grama A., Gupta A., Karypis G., Kumar V. (2003) Introduction to Parallel Computing, Pearson Education Limited.

[20] Wilkinson B., Allen M. (2004) Parallel Programming: Techniques and Applications Using Networked Workstations and Parallel Computers (2nd Edition) Prentice Hall.

[21] Upadhyaya S. R. (2013) Parallel approaches to machine learning—A comprehensive survey. Journal of Parallel Distributed Computing 73, 284–292.

[22] Kokkinos Y. and Margaritis K. G. (2016) Exemplar selection via leave-one-out kernel averaged gradient descent and Subtractive Clustering. In 12th International Conference on Artificial Intelligence Applications and Innovations (AIAI 2016), IFIP AICT 475, pp. 1–13.

[23] Lester B.P. (1993) The art of Parallel programming. Prentice Hall.

[24] Paolo Ienne, Quantitative comparison of architectures for digital neuro-computers. International Joint Conference on Neural Networks, Nagoya, Japan, vol II, pp. 1987-1990.

[25] Parallel algorithms for Digital Image Processing, Computer Vision and Neural Networks. (I. Pitas eds.) WILEY 1993, pages 316-317

[26] Fernández C, Valle C, Saravia F, Allende H (2012) Behavior analysis of neural network ensemble algorithm on a virtual machine cluster. Neural Computing and Applications 21: 535–542

[27] Nordström T., Svensson B. (1992) Using and Designing Massively Parallel Computers for Artificial Neural Networks, Journal of Parallel and Distributed Computing, 14, 260-285.

[28] Šerbedžija, N. (1996) Simulating Artificial Neural Networks on Parallel Architectures. IEEE Computer, Special Issue on Neural Computing, 29(3), 56–63

[29] Margaritis K.G., Evans D.J. (1992) Systolic implementation of neural networks. Parallel Computing (18), 325-334

[30] Margaritis K.G. (1995) On the systolic implementation of associative memory artificial neural networks. Parallel Computing (21), 825-840.

[31] Pethick, M., Liddle, M., Werstein, P., and Huang, Zh. (2003) Parallelization of a backpropagation neural network on a cluster computer. In 15th IASTED International Conference on Parallel and Distributed Computing and Systems, CA, USA, pp. 574-582

[32] Morchen F. (2004) Analysis of speedup as function of block size and cluster size for parallel feed-forward neural networks on a beowulf cluster. IEEE Transactions on Neural Networks 15(2), 515–527.

[33] Seiffert U. (2004) Artificial neural networks on massively parallel computer hardware, Neurocomputing 57, 135–150.

[34] Strey A. (2004) A comparison of OpenMP and MPI for neural network simulations on a SunFire 6800, Parallel Computing: Software Technology, Algorithms and Applications.

[35] Suresh S., Omkar S.N., and Mani V., (2005) Parallel Implementation of Back-Propagation Algorithm in Networks of Workstations, IEEE Transactions on Parallel and Distributed Systems, 16(1), 24-34.

[36] Schwenker F., Kestler H.A. and Palm G. (2001) Three learning phases for radial-basis-function networks. Neural networks, 14(4), 439-458,

[37] Santamaría, I., Lázaro, M., Pantaleón, C. J., García, J. A., Tazón, A., and Mediavilla, A. (1999) A nonlinear MESFET model for intermodulation analysis using a generalized radial basis function network. Neurocomputing, 25, 1–18.

[38] Er M.J., Wu S., Lu J., Toh H.L. (2002) Face recognition with Radial Basis Function (RBF) Neural Networks, IEEE Transactions on Neural Networks, 13(3), 697-710.

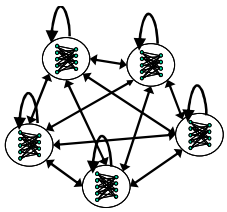
[39] Chiu, S.L. (1994). Fuzzy Model Identification Based on Cluster Estimation. Journal of Intelligent and Fuzzy Systems, 2, 267-278.

[40] Kothari, R., Pittas, D. (1999), On finding the number of clusters, Pattern Recognition Letters.

[41] Huang G.-B., Zhu Q.-Y., Siew C.-K., (2006) Extreme learning machine: theory and applications. Neurocomputing 70, 489–501.

7 Distributed privacy-preserving Regularization Network committee machines

For distributed data mining in peer-to-peer systems this chapter describes a completely asynchronous, scalable and privacy-preserving Regularization Network committee machine. Regularization neural networks are used for all the Peer classifiers as well as the combiner committee in an embedded architecture. The proposed method builds the committee machine using the large amounts of training data distributed over the peers, without moving the data, and with little centralized coordination. At the end of the training phase no Peer will know anything else besides its own local data. This privacy-preserving obligation is a challenging problem for trainable combiners but is crucial in real world applications. Only classifiers are transmitted to other peers to validate their data and send back average accuracy rates in a classical asynchronous peer-to-peer execution cycle.



Here the validation set for one classifier becomes the training set of the other and vice versa. From this entirely distributed and privacy-preserving mutual validation a coarse-grained asymmetric mutual validation matrix can be formed to map all Peer members. We demonstrate here that it is possible to exploit this matrix to efficiently train another regularization network as the combiner committee machine

7.1 Introduction

Committee machines [1][2] have long ago been recognized for their ability to combine multiple independently trained neural network modules together and to make them efficiently collaborate for the same task. The whole system is notable for its inherently parallel and distributed architecture [3]. Visually a committee machine [1][2] can be realized as a modular Neural Network which has other Neural Network modules in place of its neurons. In practice, when these Neural Network modules share the same input space and statically divide the training dataset among them then they are situated in the hidden layer neurons of the committee and the whole scheme acts as the celebrated ensemble of neural networks [1][2][3][4][5][6][7]. When these Neural Network modules dynamically divide the input space region among them, meaning that their combining weights are also dynamically dependent on the input signal as well, then they act as the well known mixture of experts [1][2]. Many other variations exist, especially in the ensemble types.

The field of distributed data mining [8] has efficiently dealt with distributed and grid computing algorithms. Data can be sensitive or private, such as consumer purchase data, medical data, census data, communication data and data gathered by government agencies. As a result, data mining was many times viewed as a threat to privacy. The related field of privacy-preserving data mining [9] has been explored independently by the cryptography, database, and statistical disclosure control communities. When Peer-to-Peer (P2P) distributed computing systems emerge, the data mining applications try to move in there to cope with the P2P restrictions on distributed data exchange, and their asynchronous and loosely coupled nature.

Hence distributed privacy-preserving Peer-to-Peer data mining [8][10][11][12] refers to the discovery of interesting models and aggregate statistics from the distributed data and without disclosing private information within the different participating Peer locations. Usually the different locations contain different sets of data records with the same attributes. Data collections can generally lay either in physically distributed database systems, in which case a small number of locations hold large volumes of data, or in Peer-to-Peer systems, where a large number of locations hold usually small volumes of data. In either case, the total data volume can become impressively large. We rather concentrate our study on Peer-to-Peer (P2P) systems, although the proposed method is more general. A part of the current work results on the proposed committee training strategy can be found in an earlier report [23].

Fully distributed P2P systems [8][10][11][12] are popular in many domains, because unlike client-server systems, they do not rely upon the servers to carry out computation and storage-intensive tasks, and do not need a central infrastructure. Some desired characteristics for algorithms that are designed for P2P data mining tasks [10][11][12] are scalability (linear to a large number of peer processor nodes), communication efficiently (if only point-to-point messages), asynchronism (without the need of frequent synchronization points), and privacy-preserving restrictions (peer processor nodes do not share local data). These factors are considered in the current work.

The basic problem in question is the following: Assume that from the whole physically distributed dataset each Peer holds a portion of the data records. First every Peer independently trains a local neural network using its own local data. Then a well defined committee machine that is composed of several such peer neural network modules can be created. The committee machine combining weights must be found. In theory for online training such a machine all Peer locations must share a portion of their data for a common validation set to be created, and all Peer locations must be synchronized to simultaneously estimate this validation data set. However, in practical applications, scalable synchronization points are difficult, and usually no location is willingly sharing its valuable data. As a consequence, the distributed locations need to cooperate asynchronously and without revealing their distributed data among them. The last factor, which is related to distributed privacy-preserving methods, is more challenging.

The essential objective in most distributed privacy-preserving data mining methods is to compute useful aggregate statistics over the whole data set without compromising the privacy of the individual participants. The goal is to compute the global data model so that nothing but the output is learned. Free flow of information is frequently prohibited by legal obligations or by personal concerns. Each participant that builds a local classifier model may wish to collaborate with all the others in order to use their

models as well, but usually it might not fully trust them to share its local data. As a rule, data are important and their gathering is costly. Hence, owning to commonly imposed restrictions on data exchange, the field of distributed privacy-preserving data mining is devoted to many real life applications and practical implementations.

To mention few privacy-preserving data mining applications [9] privacy-preservation in personalized systems (newspapers, shopping catalogs, etc.) is one, privacy-preserving medical data mining is another, genomic privacy is one more, while privacy-preserving recommendation systems and collaborative filtering also belong in this list. Many other such applications exist in security-control and surveillance, like in homeland security, intrusion detection, and bio-surveillance.

The practical implementations exploit ideas from secure multiparty computations. The central idea of any secure multiparty computation is that at the end of it no party will know anything else except its own input and the aggregated result. For instance, the secure sum protocol [13] computes the sum of a collection of numbers without revealing anything but the output sum. Some types of classifiers which need total sums, like the Bayesian ones, can be implemented in this fashion. Classifier examples that have been generalized to distributed privacy-preserving data mining field are the Naïve Bayes classifier [14][15], the SVM classifier with nonlinear kernels [16] and the k-nearest neighbour classifier [17]. In this work, we take a different path and focus on a more general approach; a committee machine. A simple alternative method for training the proposed committee machine is also demonstrated.

We study a typical committee machine that consists of Regularization Networks (RN) distributed over the interconnected Peer workstations which hold their own data. The Regularization Networks [18][19][20][21] are kernel based. They use the real training data points in their hidden neurons to form the kernel function centers. Thus, they manage to capture the data closeness approximate of the underlined problem distribution. Using real points as kernel centers is valuable when data features have discrete values, e.g., in cases of image processing, computer vision and data mining [22]. This fact elevates such type of kernel based ridge regression methods [24][25] to state-of-the-art.

In the distributed system each Peer trains a Regularization Network module to efficiently classify its own local data partition. Then all these distributed RN classifier modules can be combined by using another Regularization Network, to serve as a committee machine. This RN committee machine places the RN modules in its hidden layer neurons to build a global model. Without the privacy-preserving restriction the individual RN classifier modules can be combined [4][5][26[27] by means of performance weighting, statistical techniques, Dempster-Shafer theory and belief functions, entropy weighting, and other fusion methods [7]. These methods normally require the use of extra information from the RN classifier input-output mappings. At least two-by-two the classifiers must share either input vectors, or output vector results with respect to all instances of an independent common validation set. The usual weighting approaches are to give greater weight to the RN modules that deliver better results. These approaches, however, requires a separate validation set in order to find individual module errors, and to weight the modules according to their classification performance. If the separate validation set is available then one can also use it to train a neural network meta-learner by following the stacked generalization principles.

On the other hand, the challenging problems are to find the committee weights for the RN committee modules without moving or sharing the local data, with only point-to-point communications, and with little centralized coordination. We propose to compute the Regularization Network Committee Machine combination weights by using a simple distributed privacy-preserving mutual validation matrix. Thus the main contribution of this chapter is that the proposed committee training uses a simple strategy to tackle the previous factors that is communication efficiency, asynchronous, and distributed privacy-preserving. The individual pair-wise computations for the mutual validation matrix entries are asynchronous, fully distributed, scalable, and preserves privacy at the same time. Therefore this strategy renders the committee machine a possible candidate for distributed privacy-preserving data mining in physically distributed data repositories, like those in peer-to-peer systems.

The rest of the chapter is organized as follows. Section 2 briefly presents background knowledge on the basic concepts of committees and neural network ensembles. Section 3 describes the proposed committee machine architecture of regularization neural networks. Section 4 clarifies our implementation details of the distributed privacy-preserving strategy for training the committee machine. It is based on an asynchronous computation for a distributed asymmetric mutual validation matrix, and a simple kernel based training method to find the committee combining weights. Section 5 describes implementation details for the conventional stacked generalization (stacking) with no privacy-preserving restrictions. Stacking is used for further experimental comparisons. Section 6 provides experimental results on the effectiveness of the proposed method and presents some points for discussion. Section 7 contains summaries, conclusions and future research issues.

7.2 Committees and neural network ensembles

There are two well known types of committee machines, which are 1) neural network ensembles and 2) mixture of experts. In a neural network ensemble, all modules operate on the same input space. In a mixture of experts, each module expert dynamically specializes over a sub-region of the input space. Committees of ensemble neural networks [27][28][29] are studied here, which can have very flexible architectures. They have excellent generalization capabilities, and their performance can be better than the best stand-alone neural network used in isolation [1]. As a rule of thump, the uncorrelated errors of the individual neural network classifiers can be eliminated through averaging [6][7]. The individual neural network module classifiers are independently constructed in parallel, based on their local training data. After this construction phase, the committee machine must combine them, via a typical classifier combination scheme [30] through proper weights to form the global data model.

In classification tasks, after training the neural network modules, the combination of their decisions is typically implemented as a (weighted) voting scheme. The committee assigns the pattern to the class that gets the (max) majority of the vote:

$$class(\mathbf{x}) = \arg \max_{j} \left(\sum_{i=1}^{L} g_i \cdot f_{i,class=j}(\mathbf{x}) \right) \tag{7.1}$$

where $f_{i,class=j}(\mathbf{x})$ is the output of the neural network classifier module i for class j, and L is the population of these classifiers. Typically the outputs either correspond to the posterior class probability range [0, 1] or to a binary class decision {0, 1}. Using

posterior class probabilities, this combination is the same as in typical regularization networks, support vector machines and many others.

Committee machines can be useful in many ways. The first reason is that the committee usually exhibits a generalization performance far better than any single committee member, simply because the errors of the individual committee members cancel out to some degree. Techniques like Bagging, averaging and boosting are working along this line.

A second reason for using committee machines is modularity. It is sometimes beneficial if a mapping from input $\mathbf{x}$ to target y is not approximated by one estimator but by several ones, where each estimator can focus on a particular region in input space. Mixture of experts, and its variants, is the most important representative of this approach.

A third reason for using committee machines is a reduction in computational complexity. Instead of training one estimator using all training data, it is computationally more efficient for some types of estimators to partition the data set into several data sets, train different estimators on the individual data sets, and then combine their predictions. Typical examples of estimators for which this procedure is beneficial are Gaussian process regression, regularization neural networks, smoothing splines, and the support vector machine, since for those systems, the training time increases cubically as $O(N^3)$, and thus drastically with increasing training data set size N. By using a committee machine approach, the computational complexity can be significantly reduced.

A forth reason for using a committee is in data mining large scale physically distributed data repositories as well as peer-to-peer systems. Gathering large volumes of distributed data to a single location for centralized data mining is unfeasible. The causes that prevent this lay in technical issues like limited network bandwidth and enormous main memory demands, practical issues like huge required training times, algorithmic issues in where mining algorithms operate only on data in main memory, and especially privacy concerns that restrict the transferring of sensitive data.

Since for multi-class problems the neural networks have been proven to be the best over the years, we present a Regularization Network committee machine that consists of Regularization Network modules. Each module is implemented in a different Peer processor and the high level distributed committee combines these participants. Thus in order to combine their decisions, the proposed committee training phase finds proper weights for each one of the participating Peer modules, and in the end leaves them with no extra knowledge of the other participants' data. The use of a distributed privacy-preserving regularization network as such a committee machine, is presented next.

7.3 A Regularization Networks Committee Machine

In this section a Regularization Network (RN) committee machine composed of Regularization Network modules for classification problems is analyzed. In the mathematical formulas an upper bold letter will symbolize a matrix, while a lower bold letter will symbolize a vector, and an italic letter will denote a scalar variable. A traditional Regularization Network [18][19][20] [21] has one input layer, one hidden layer, and one output layer. All the real training data points are loaded to the hidden neurons to form the kernel function centers.

Typically given N training set pairs $\{\mathbf{x}_n, y_n\}_{n=1}^{N}$, a kernel function $k(\cdot,\cdot)$ and a Kernel matrix $\mathbf{K}$ with entries $K_{i,j}=k(\mathbf{x}_i, \mathbf{x}_j)$, the Regularization Network training phase finds an optimum nonlinear mapping function with proper weights w_i, such that the label estimation y for an unknown input vector $\mathbf{x}$ is given by $f(\mathbf{x})=\sum_{n=1}^{N} w_n \cdot k(\mathbf{x}_n, \mathbf{x})$. The kernel function is usually the Gaussian kernel $k(\mathbf{x}_n,\mathbf{x})=\exp(-||\mathbf{x}_n-||^2/\sigma^2)$ centered at a particular point.

Many such Regularization Network modules can compose an ensemble, and if another RN is used to combine their decisions with proper weights then a RN Committee Machine can be formed. Thus a particular hidden 'neuron' of this RN committee is actually another RN. The predicted class for an unknown input $\mathbf{x}$ is the one having the maximum output value for this class.

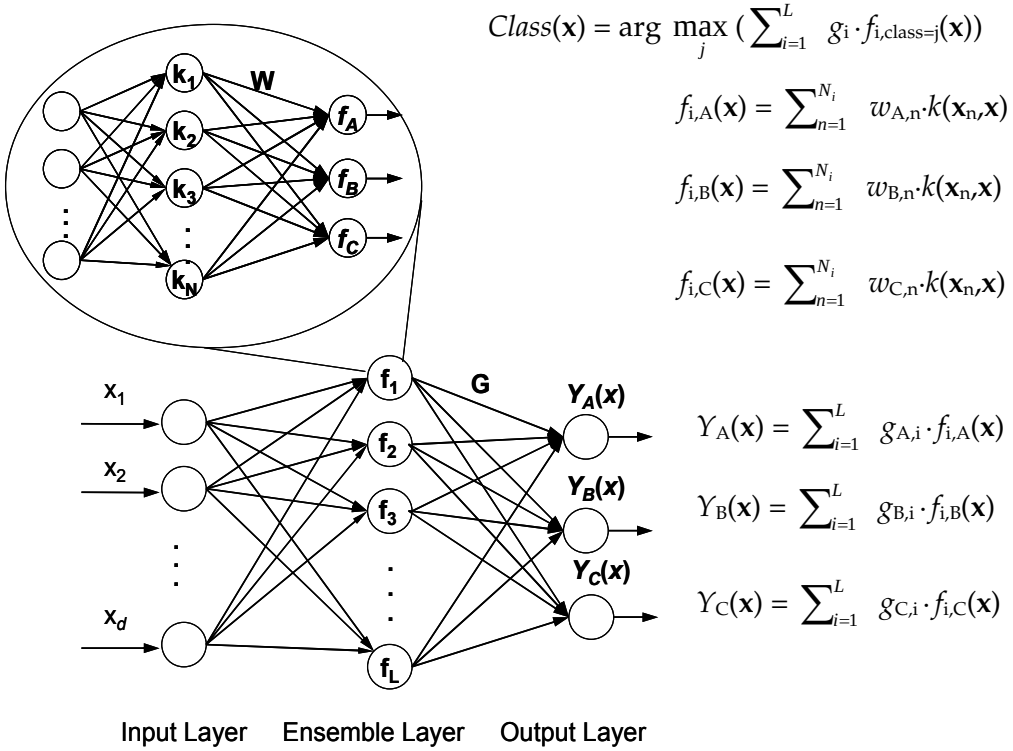


Figure 7.1: Architecture of a Regularization Network committee machine composed of many Regularization Network modules situated in each one of the Peers.

A Regularization Network committee machine for the three-class problem is illustrated in fig. 7.1. This architecture has L combined RN modules $\{f_1, f_2, \ldots f_L\}$. Each RN module i consists of a hidden layer with N_i neurons of kernel units and three linear output units. The RN committee machine has also three outputs, one for each class. Note here that, in the case a RN module holds no data points from a particular class, it still must retain three outputs, where naturally for this missing class all kernel weights w_n would be zero. This comes purely for compatibility reasons among the RN modules. The architecture in fig. 7.1 also illustrates the required weight matrix $\mathbf{W}$ of each RN module and the weight matrix $\mathbf{G}$ of the committee.

Thus for each individual RN module the weight vectors $\mathbf{w}_A$, $\mathbf{w}_B$ and $\mathbf{w}_C$, one for each class, must be found first. The conventional algorithm was generated from the Tikhonov regularization schemes. Following the regularization approach [18][19][20][21] the kernel matrix $\mathbf{K}$ is assumed to be continuous and positive semi-

definite, for a finite set of training points. Then the Reproducing Kernel Hilbert Space (RKHS) H_K associated with the kernel matrix **K** is well defined, and the learning problem is stated as a minimization of a regularized functional (eq. 7.2). This regularized functional on the H_K consists of the usual data term plus a second regularization term that plays the role of the stabilizer [18][19][20][21]:

$$\arg\min_{f \in H_K} \left\{ \frac{1}{N_i} \sum_{n=1}^{N_i} (y_n - f(x_n))^2 + \gamma \|f\|_K^2 \right\} \tag{7.2}$$

In eq. 7.2 the data term is scaled proportionally to the number N_i of the training data samples that the particular RN module *i* has, and $\gamma > 0$ is the regularization parameter that controls the trade-off between the two terms, i.e. the trade-off between the closeness to data and the solution smoothness. It follows from solving the minimization problem in eq. 7.2 that for one class output, say C, the weights vector $\mathbf{w}_C$ is the solution of the linear system $(\mathbf{K} + N_i\gamma\mathbf{I})\mathbf{w}_C = \mathbf{y}_C$, where **I** is the identity matrix, **K** is the kernel matrix with entries $\mathbf{K}_{i,j} = k(\mathbf{x}_i, \mathbf{x}_j)$, and $\mathbf{y}_C = (y_1, \ldots, y_N)$ is the vector of the desired output labels, which are 1 for class C samples and 0 for the others (hot encoding). Thus the solution set for the three classes that correspond to any Peer RN module *i* in fig. 7.1 is given by eq. 7.3.

$$\mathbf{w}_A = (\mathbf{K} + N_i \gamma \mathbf{I})^{-1} \mathbf{y}_A \tag{7.3a}$$
$$\mathbf{w}_B = (\mathbf{K} + N_i \gamma \mathbf{I})^{-1} \mathbf{y}_B \tag{7.3b}$$
$$\mathbf{w}_C = (\mathbf{K} + N_i \gamma \mathbf{I})^{-1} \mathbf{y}_C \tag{7.3c}$$

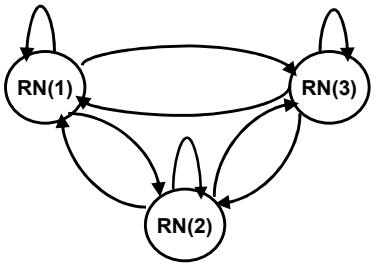
After the training for each one of these RN modules is finished, the global training phase for the high level RN committee machine must start in order to identify the still unknown weight vectors $\mathbf{g}_A$, $\mathbf{g}_B$ and $\mathbf{g}_C$. Finding these weights blindly, without revealing data vectors between modules is a challenge. We propose to train the committee using a mutual validation matrix.

7.4 Proposed training for the RN Committee Machine

7.4.1 Compute the distributed mutual validation matrix

The distributed mutual validation matrix is a simple idea we are going to use towards a fully distributed and privacy-preserving training of the P2P RN committee machine. While the presented paradigm specifically uses RNs any other type of Peer classifiers can also be used. In this section we will describe the computations needed for the distributed asymmetric mutual validation matrix entries, and in the next section we will use this matrix to find the committee weights. Assume an ensemble of three Regularization Network modules, namely RN(1), RN(2) and RN(3) that are situated in three different locations across a P2P communication network. These remote RN modules have already been trained independently from each other based on their local datasets, and are willing to participate in forming the committee machine. However, for privacy reasons the different locations cannot contribute or share even the smallest part of their datasets to other RN modules. Only RN classifiers in the form of binaries or agents can be sent to other locations. Then, one can schematically work with accuracy measures of classification rates between RN(1), RN(2) and RN(3), in which the validation set of one classifier becomes the training set of the other and vice versa. All RN modules classify each other. Only average classification rates can be returned back to fill in an asymmetric mutual validation matrix **V**. The RN(1) classifies its own

data points to produce the v_{11} entry. Likewise RN(2) classifies its own data to produce v_{22} entry and RN(3) produces v_{33} entry. In consequence RN(1) classifies the training data of RN(2) to produce v_{12} entry and RN(2) classifies RN(1) data to produce v_{21} entry. In the same way v_{13}, v_{31}, v_{23} and v_{32} entries are formed. The mutual validation matrix **V** that can be filled with those pair-wise validations is illustrated in fig. 7.2. One can observe that these matrix entries are usually asymmetric, meaning that v_{12} is not equal to v_{21}, and so on.



$$V = \begin{bmatrix} v_{11} & v_{12} & v_{13} \\ v_{21} & v_{22} & v_{23} \\ v_{31} & v_{32} & v_{33} \end{bmatrix}$$

Figure 7.2: (a) A P2P ensemble of three different Regularization Networks RN(1), RN(2) and RN(3), inter-connected with each other via accuracy measures, (b) the mutual validation matrix mapping of the ensemble member Peers.

The distributed paradigm in fig. 7.2 is an illustrative example of the simple point-to-point communications involved in the procedure. The rows of the matrix **V** are corresponding to local data and the columns of **V** to the traveling RN classifiers. Upon receiving a jth Regularization Network classifier the ith peer applies it to classify its own N_i local data. Then peer i sends back a simple average learning rate $V(i, j)$ equal to the ratio of positive local hits, of the jth classifier when classifying the samples of ith peer, per local training data size N_i. Positive hits are given by the number of correctly classified local samples of ith peer and local training data size is their N_i population. In this way privacy-preserving is achieved.

Given a classifier f_j the typical classification error for a single training sample **x** with a desired label y is given by $e_j(\mathbf{x}) = \{0$ if $y = f_j(\mathbf{x})$, and 1 otherwise$\}$. Thus in terms of the classification errors of the jth classifier against an ith dataset of N_i training sample pairs $\{\mathbf{x}_n, y_n\}$, of **x** samples and y desired labels different for each peer i, the mutual validation matrix **V** entries are defined as:

$$V(i, j) = 1 - (1/N_i) \sum_{n=1}^{N_i} e_j(\mathbf{x}_n) \tag{7.4}$$

The computation procedure of the matrix **V** entries is fully decentralized.

The diagonals of the matrix **V** are the self-validation average positive hits of each RN classifier. Distributed computations are required for all the v_{ij}, and v_{ji} asymmetric entries between different RN modules across the communication network. An asynchronous Peer-to-Peer computation cycle is continually executed. This cycle is composed of typical commands, like 'sent local module classifier', 'check for received classifier', 'compute local positive hits of received classifier', and 'sent back average learning rate'. Thus the communication model used is simple point-to-point with only sent-receive commands to or from a single Peer.

Following the proposed method, one manages to transform the committee machine training phase into a fully asynchronous embarrassingly parallel programming paradigm, known as the iterative decomposition [31]. Iterative decomposition (or equivalently task-farming) occurs when a loop parallel execution can be done in some

independent and unconnected manner. Each processor may operate independently and communicate its own results to another processor, making it an appropriate choice for various types of asynchronous cycles. Those sorts of programs can easily be made fault-tolerant, as the whole computation can sufficiently survive and recover from the loss of a processor node.

Therefore, from these fully distributed and privacy-preserving pair-wise mutual validations the mutual validation matrix **V** can be formed. The matrix **V** maps all the Peer modules. Although such a matrix was so far neglected, we here demonstrate that it is possible to fully exploit it to efficiently train another regularization network as a committee machine.

7.4.2 Finding the linear combining weights

Up to this point all the RN modules have been trained, and only the weight vectors $\mathbf{g}_A$, $\mathbf{g}_B$ and $\mathbf{g}_C$, one for each class, of their combiner regularization network committee machine are still unknown. The mutual validation matrix **V** which was previously computed can now be used to find the committee weights. Details of the proposed training method are provided here. It uses simple regularized least squares, where **V** plays the role of the regression matrix. The committee weight vectors $\mathbf{g}_A$, $\mathbf{g}_B$ and $\mathbf{g}_C$ can be found (eq. 7.5) by first considering the mutual validation matrix **V** as a regression matrix and then solving the linear equations, one for each class, in terms of the vectors $\tilde{\mathbf{y}}_A$, $\tilde{\mathbf{y}}_B$ and $\tilde{\mathbf{y}}_C$, which are used to hold the desired averaged labels, as follows:

$$\mathbf{g}_A = (\mathbf{V} + \lambda\, \mathbf{I})^{-1}\, \tilde{\mathbf{y}}_A \tag{7.5a}$$

$$\mathbf{g}_B = (\mathbf{V} + \lambda\, \mathbf{I})^{-1}\, \tilde{\mathbf{y}}_B \tag{7.5b}$$

$$\mathbf{g}_C = (\mathbf{V} + \lambda\, \mathbf{I})^{-1}\, \tilde{\mathbf{y}}_C \tag{7.5c}$$

All the desired averaged label vectors $\tilde{\mathbf{y}}_A$, $\tilde{\mathbf{y}}_B$ and $\tilde{\mathbf{y}}_C$ in eq. 7.5 that correspond to the classes A, B and C have size L equal to the number of the Peer RN classifiers. A value $\tilde{\mathbf{y}}_A(i)$ is the number of desired positive local hits of RN classifier i per total training size, that must be produced by applying the RN classifier i to the A class portion of its own local dataset, the ith dataset (if not any then set 0).

Thus, in terms of the total training data size N of all the peer samples and the classifier f_i errors $e_i(\mathbf{x})$ = {0 if $y = f_i(\mathbf{x})$, and 1 otherwise}against a sample $\mathbf{x}$ of its own A class portion of $N_{i,A}$ training sample pairs $\{\mathbf{x}_n, y_n\}$, a desired averaged label $\tilde{\mathbf{y}}_A(i)$ entry is defined as:

$$\tilde{\mathbf{y}}_A(i) = (1/N)\, (N_{i,A} - \sum_{n=1}^{N_{i,A}} e_i(\mathbf{x}_n)\,) \tag{7.6}$$

The average value $\tilde{\mathbf{y}}_A(i)$ simply measures how well the classifier i performs on its own samples of class A. For example, if the classifier i achieves 10 positive local hits on its own A class samples then $\tilde{\mathbf{y}}_A(i) = 10/N$. If it holds zero samples of class A then $N_{i,A}= 0$ and $\tilde{\mathbf{y}}_A(i) = 0$. Respectively $\tilde{\mathbf{y}}_B(i)$ is equal to desired positive local hits per total training size that are produced by the classifier i for the B class portion of its own data, and $\tilde{\mathbf{y}}_C(i)$ is similarly defined for the C class portion. Note that the notion of a desired averaged label in eq.7.4 is not restricted only to positive local hits per total training size if other cumulative accuracy measures are also known. For example, one can try some form of aggregated confidence. The main idea remains the same. A zero $\tilde{\mathbf{y}}_A(i)$ value is assigned if the classifier i has no internal knowledge of the particular class A (and consequently the local separating boundaries of A with the other classes are unknown), and a positive average value if it has. For the same reason, that is the existence of sometimes

insufficiently small sample sizes in the peers, some columns in the matrix **V** may be produced with all values equal to zero. Then a regularization parameter must compensate for this, in order to avoid singularities. In eq. 7.5 this regularization parameter is λ, and **I** is again the identity matrix. We also compare the proposed method with the non privacy-preserving training via stacked generalization that uses a separate, common to all Peers, validation set.

7.5 Comparison with Stacked generalization

Implementation details are presented in this section for stacked generalization [6][7][32], or stacking in sort, which is a common natural choice for the high level training of a committee machine. An ensemble of classifiers are first created, whose outputs are then used as inputs to a second level meta-classifier. This meta-classifier learns the combining weights by mapping between the ensemble outputs and the actual classes.

Although in trainable combiners like stacking there is no privacy-preserving restriction, we describe it here for comparison purposes. To use stacking for finding the weights **G** of the committee, all locations must share a portion of their data in order to create a global validation set **T**. According to stacking, all L individual RN classifier modules (or level-0 models) must be trained independently in the same way as before. Then stacking uses the global separate validation set **T**, gathered from all local sites, to train the meta learner (or level-1 model), which tries to learn the weights of the base RN classifiers. Thus the meta-learning algorithm first uses the samples in **T** and map each one of them to level-0 modules for creating the level-1 training set, the **A** set. Each level-1 training sample in the **A** set has L attributes, whose values are the predictions of each one of the L level-0 classifier. Therefore, a level-1 training sample is made of L attributes and the desired target class y. Once the set **A** has been built any meta-learning algorithm can be used to generate the level-1 model.

For the case of a Regularization network committee we use a similar architecture as in fig. 7.1, where now all $f_i()$ modules output one decision, their estimation. The level-1 training set **A** is used to determine the weights **G**. In operation the ensemble layer transforms any unknown new instances to the new mapping space. This way, the ensemble layer becomes the new transformed input layer and only the linear combination weights **G** are needed. Using the transformed level-1 set **A**, the weights **G** are obtained by solving the linear system $\mathbf{A}\cdot\mathbf{G}=\mathbf{Y}$ of which the regularized solution is $\mathbf{G}=(\mathbf{A}^T\mathbf{A}+\lambda\mathbf{I})^{-1}\ \mathbf{A}^T\mathbf{Y}$. Here the lambda λ parameter for regularization is optimized via cross-validation. For each dataset and each run in the experimental section we select the lambda parameter value that corresponds to the minimum cross-validation error by searching in the range [64, 0.01].

In the experimental comparisons two versions of stacking are used, namely, stacking version 1 and stacking version 2, differ only in the sampling procedure. The first version creates the validation set by stratified random sampling without replacement of a percentage of all the Peers local data, thus removing the selected samples from the peers. The second version performs stratified random sampling but with replacement, thus leaving the selected samples in the Peers, so the validation set contains samples that are all duplicates, since they also exist in the Peers.

7.6 Experimental results and discussion

The group of experiments aims at discovering the classification performance of the RN committee tested on a separate test set of points. To this end, we use several benchmark datasets taken from the UCI machine learning data repository (http://archive.ics.uci.edu/ml). A synopsis of the datasets used is illustrated in table 7.1. In order to show the efficiency of the proposed method, highly irregular data partitions must be created; otherwise accurate estimations may emerge simply from the fact that we use an ensemble. For the same reason a comparison is also made with majority voting, simple weighted average regression and two versions of stacking, namely stacking version 1 and stacking version 2. Details of these methods are described next.

The majority voting method simply accumulates votes for each class and the class with maximum votes wins. The simple weighted average regression method can calculates each weight g_i of a Peer i as $g_i = \log(p_i/(1-p_i))$ where p_i is the individual accuracy of i classifier. This is based on the commonly used democracy theorem (see theorem 4.1 in the book [26]). The democracy theorem states that the accuracy of the ensemble must be maximized by assigning these optimal weights, which do not take into account the performance of other members of the team, but only magnify the relevance of the individual classifiers.

The Stacking method is trainable and needs an independent common validation set that must be gathered in a central location from all participating Peers. Stacking version 1 uses 15% of all the Peers training data without replacement, and stacking version 2 uses 15% of all the Peers training data with replacement. Usually, gathering 15% data for a validation set is considered an adequate maximum in order to preserve the distributed character of the method; otherwise it will be regarded as a conventional training method in a central server machine.

The experimental design is as follows:

1. A dataset is randomly split into a training set (70%) and a testing set (30%) with stratification.
2. The training set is distributed unevenly, randomly and without stratification across a number of Peers.
3. Each Peer trains a Regularization Network.
4. An asynchronous computing cycle is executed to find all entries of the proposed mutual validation matrix.
5. The high level RN committee is trained using the mutual validation matrix.
6. The final RN committee machine is tested on the testing set.

This procedure is repeated 10 times for each benchmark dataset and each corresponding processor number, and the error rate results are averaged. All error rates are measured by the ratio (falsely classified samples)/(total samples) on the test set.

A single Regularization Neural Network is trained again on the same initial training set and tested on the same test set for comparison. We also compare the accuracy performance of the committee with majority voting, weighting average regression and stacking. Although stacking is not privacy-preserving, we chose to include it, in order to conduct a thorough investigation on the possibilities of the proposed committee machine training method.

In step 2 of the experimental design the uneven, as well as random and without stratification choice of a particular processor's data is important for the experiment to simulate a real situation and to show the power of the committee. To this end, we allow a quarter of processors to randomly peek a population size between 5 and 300 training points. Likewise, another quarter randomly peek a population size between 5 and 100. Similarly the remainder half of processors are allowed to have a size between 5 and 30. Then according to the total number of training points these population sizes are normalized, in favor of the smaller ones, for their sum to fit the total. This method produces a fairly uneven un-stratified distribution, with half of processors' populations being small. Many of them end up with no samples from some class. In addition, small local populations are likely to produce singularities to the mutual validation matrix inversion, in order to make the proposed training method harder and to show the benefits of the RN stabilizer. Other uneven and irregular distributions we tried have worked equally well as the former one.

Table 7.1. Benchmark datasets used in the experiments

Dataset Name	instances	features	classes
Iris	150	4	3
Wine	178	13	3
Diabetes	768	8	2
Wisconsin	683	9	2
Glass	214	9	6
Vehicle	846	18	4
Page Blocks	5473	10	5
Spam Emails	4601	57	2

The first 4 benchmark datasets in table 7.1 are considered easy and this is the reason that they all belong to the category of the most popular datasets in the UCI repository. The other datasets are regarded as more complex. For each dataset of the experimental results, a variable number of Peer modules were tested as indicated in the first column of the corresponding tables.

For the Iris dataset in table 7.2, the single RN trained on the whole training set obtains 3% error on the same test set. The first column in table 7.2, RN Peer modules, indicates the number of RNs in the ensemble layer, and is used to show the range of the applicability. Each other column designates the method that is used. In the second column all module networks in the ensemble layer perform simple majority voting. The third column shows the simple weighted average error. The fourth and fifth columns show the stacking error. The last column illustrates the proposed distributed privacy-preserving RN committee error. The RN committee outperforms the majority voting, simple weighting average regression, and shows comparable accuracy results with the two stacking versions. It also unexpectedly recovers the single RN error rate in most of the experimental cases.

Table 7.2. Iris dataset error rates and comparison results

RN Peer modules	Majority Voting	Weighted average	Stacking version 1	Stacking version 2	Privacy Preserving RN committee
10	3.6	3.9	3.5	3.8	3.0
12	5.2	4.9	3.3	3.8	3.1
14	6.6	4.9	3.6	4.0	3.8

16	5.6	4.4	3.8	3.9	3.8
18	5.4	4.8	3.3	4.0	3.9
20	5.8	5.4	3.5	4.2	3.6

The Wine dataset experimental results are illustrated in table 7.3. The Peers in this case range from 10 to 20 as indicated in the first column. The single RN training obtains 2.7% error. Again the RN committee error results are better than majority voting, weighted average regression and also are comparable with the two stacking versions. In addition the RN committee results are comparable to the single RN case.

Table 7.3. Wine dataset error rates and comparison results

RN Peer modules	Majority Voting	Weighted average	Stacking version 1	Stacking version 2	Privacy Preserving RN committee
10	4.4	2.9	2.4	2.5	2.9
12	3.4	2.4	2.2	2.6	2.5
14	3.4	2.9	2.2	2.7	2.3
16	4.2	3.4	2.5	3.3	3.3
18	4.7	2.9	1.9	2.2	2.3
20	4.7	2.1	1.8	2.2	1.8

Table 7.4 presents the Diabetes dataset results when using Peers from 50 to 100. The error rate of a single RN was found to be about 25% on the same test set. The RN committee again outperforms majority voting, weighted average and once more delivers comparable results with the two stacking versions. Also the RN committee manages to be as accurate as the single RN which was unexpected.

Table 7.4. Diabetes dataset error rates and comparison results

RN Peer modules	Majority Voting	Weighted average	Stacking version 1	Stacking version 2	Privacy Preserving RN committee
50	29.1	27.5	26.8	25.6	25.9
60	29.9	26.5	22.9	23.5	24.6
70	29.0	26.8	23.5	23.1	24.1
80	30.8	29.1	25.3	25.3	25.4
90	31.6	29.1	25.2	25.9	26.0
100	30.3	28.2	26.0	25.3	24.5

Table 7.5 shows the Wisconsin dataset results obtained by using 10 to 60 Peers. The error rate of a single RN was found to be 3.2% on the same test set. Again the RN committee performs better than majority voting, weighted average and brings comparable results with the two stacking versions.

Table 7.5. Wisconsin dataset error rates and comparison results

RN Peer modules	Majority Voting	Weighted average	Stacking version 1	Stacking version 2	Privacy Preserving RN committee
10	4.0	3.4	2.8	3.0	3.8
20	4.5	3.3	2.4	2.7	3.5
30	4.2	3.3	2.7	2.3	2.9
40	5.0	4.2	2.8	3.0	3.8

50	5.4	4.4	3.1	3.2	4.0
60	5.4	4.0	3.2	3.2	4.0

In table 7.6 the Glass dataset results are presented that span from 10 to 20 Peers. The error rate of a single RN was found to be 33.1%. Here again the RN committee performs much better than majority voting, weighted average. Furthermore this dataset, as well as the following ones, shows some potentials of the proposed RN committee to handle complex difficult datasets given that the last column entries of table 7.6 clearly overruns the two stacking versions by a substantial margin.

Table 7.6. Glass dataset error rates and comparison results

RN Peer modules	Majority Voting	Weighted average	Stacking version 1	Stacking version 2	Privacy Preserving RN committee
10	41.8	43.3	43.9	41.2	35.5
12	41.5	40.3	40.5	39.6	36.1
14	42.3	40.0	41.3	39.1	38.1
16	43.3	41.8	42.5	41.3	37.3
18	42.3	38.8	43.3	38.3	36.1
20	40.3	38.4	40.3	38.0	36.2

Table 7.7 shows the Vehicle dataset results that range from 10 to 35 Peers. This is another complex dataset that shows some possibilities of the RN committee machine. The single RN error rate was found to be about 21%. Again the RN committee performs much better than majority voting, weighted average as well as the two stacking versions from which it has a major precedence.

Table 7.7. Vehicle dataset error rates and comparison results

RN Peer modules	Majority Voting	Weighted average	Stacking version 1	Stacking version 2	Privacy Preserving RN committee
10	28.9	27.3	27.9	27.2	26.2
15	30.1	28.1	28.7	29.0	24.7
20	31.6	30.1	29.8	30.4	25.8
25	32.5	30.2	33.1	33.4	27.3
30	33.9	31.0	33.9	34.1	28.7
35	35.2	31.4	34.2	34.0	28.6

In table 7.8 the page block experimental results are presented. They span from a wide range of Peers starting from 20 to 200. The size of the dataset permits that. Here Stacking performs slightly better that the privacy-preserving committee as indicated from their error differences.

Table 7.8. Page Blocks dataset error rates and comparison results

RN Peer modules	Majority Voting	Weighted average	Stacking version 1	Stacking version 2	Privacy Preserving RN committee
20	5.7	6.3	8.2	7.9	6.9
40	6.4	6.6	8.2	8.4	6.7
60	8.2	7.9	8.0	7.9	7.5
100	8.4	7.6	7.5	7.1	7.8

150	9.8	8.3	7.5	7.6	8.1
200	10.1	8.7	7.6	7.5	8.8

Table 7.9 illustrates the Spam Emails dataset results. These experiments range from 20 to 200 Peers. This is another complex dataset that reveals some potentials of the RN committee which outperforms the stacking versions. In this dataset, the single RN trained with all samples was found to have 8.1% error on the test set. Table 7.9 clearly shows a rather large advantage of the privacy-preserving RN committee in the last column as compared with all the other four methods.

Table 7.9. Spam Emails dataset error rates and comparison results

RN Peer modules	Majority Voting	Weighted average	Stacking version 1	Stacking version 2	Privacy Preserving RN committee
20	8.8	8.9	9.5	9.1	8.8
40	10.5	10.4	9.6	9.5	8.7
60	11.5	11.1	9.4	9.1	8.4
100	12.8	12.5	9.7	9.6	8.8
150	13.4	12.9	9.3	9.7	8.4
200	14.8	14.2	10.3	10.2	9.4

Based on all these comparisons, the RN committee is found to outperform majority voting, weighted average and achieves comparable accuracy with stacking in most of the cases examined in the datasets Iris, Wine, Diabetes and Wisconsin. Although the uneven un-stratified splitting produces highly irregular data distributions, the RN committee was also found to perform slightly better than the single RN, as revealed from datasets like the Iris, Wine and Wisconsin, which was unexpected. For more complex datasets like Glass, Vehicle and Spam Emails the RN committee was found superior not only from majority voting, simple weighted average but also from the two non privacy-preserving versions of stacking which was rather surprising. We believe that the multi-class response nature of the RN committee is a key ingredient. Therefore, as revealed from the previous experimental comparison results, one may conclude that the proposed method is promising. When applied to data mining tasks in a large Peer-to-Peer system of many Peers that hold small volumes of data, this distributed privacy-preserving Regularization Networks committee machine may efficiently handle complex data distributions.

Now let us consider a general loss function denoted as $V(f(\mathbf{x}), y, \mathbf{u})$, where $f()$ is the classifier, $\mathbf{x}$ is a training pattern, y is the pattern's label and $\mathbf{u}$ is the parameter vector (weights etc.) of the classifier. The minimization of V with respect to $\mathbf{u}$ over the training patterns is the strategy that provides the optimum solution for the parameter vector $\mathbf{u}$ [20]. When two classifiers i, j are compared versus a common separate validation set, the comparison is made on their common outputs. Thus, their distance measure $d(i, j)$ is usually dependent on their pair of parameter vectors $\mathbf{u}_i$ and $\mathbf{u}_j$. This means that the outcome $d(i, j)$ is biased from their joint biases. The proposed mutual validation matrix method is independent of the joint parameter vectors $\mathbf{u}_i$ and $\mathbf{u}_j$. In our case an asymmetric measure like the mutual validation matrix entry $v(i, j)$ depends only on the

parameter vector $\mathbf{u}_i$ of classifier *i* and is thus independent from the bias and variance of the classifier *j*. Consequently it is a potential candidate to be used for uncorrelated distance measure estimation.

7.7 Summary and future work

Using a distributed committee machine situated in a large scale Peer-to-Peer system, the present study considers the challenging case to train it asynchronously and with no local data exchange among the neural network classifiers. Regularization networks are used for all the individual peer classifiers as well as the combiner committee in an embedded architecture. When the training is finished, no peer module will know anything else except its own input local data. For this distributed privacy-preserving case we propose a simple training strategy that maps all peers in an asymmetric mutual validation matrix. This matrix has their mutual classification rates as entries. First this matrix is computed asynchronously via point-to-point communications. The training set of one peer becomes the test set of the other and vice versa. Then, by using this matrix, the weights of the RN committee machine are found. The proposed method manages to compute the weights, while preserving the privacy among peers in the same time. These features render the RN committee a candidate tool for Peer-to-Peer data mining. Extensive experimental results are supportive by showing that the proposed RN committee outperforms majority voting, simple weighted average and stacked generalization in most of the cases examined. The proposed RN committee machine training method also favors the distributed asynchronous nature of the P2P system, and could also be used with other type of classifiers. While we have covered here typical classification tasks the study can be straightforwardly extended to other tasks like regression or function approximation.

At this point some future works could be mentioned. Though the Regularization neural network algorithm performs well and in many applications, it might be practically challenging when the sample size *N* of data is very large. During RN training the memory requirement is quadratic $O(N^2)$ and the computational complexity is cubic $O(N^3)$ so for $N>100000$ this algorithm is difficult to implement. It might be possible for the presented RN committee machine to be of assistance to split the work without substantial loss of accuracy. Training with fine-grained modules can be accomplished by finding clusters of data. In the future we plan to test the proposed method as a direct approach to speed up the RN training process.

We have already put the committee machine side by side with the Wolpert's stacked generalization. For an ensemble of classifiers, which operate on a data set from an unknown distribution, the stacked generalization maps the outputs of these classifiers to their true classes through the meta-learner weights. While only linear combination weights are considered here, as it is commonly done in similar studies, a stacking committee machine with one extra non-linear mapping layer is also possible. To use a kernel based regularization network as such, the high level meta-learner training is again restricted to solve a linear system of the form $(K+\lambda I)w=y$. Thus with or without the privacy-preserving limitation, the static training of such a general meta-learner may require a coarse-grained high level matrix. Future experiments could explore such a possibility.

References for chapter 7

[1] Bishop C.M. (1995) Neural Networks for Pattern Recognition. Oxford: Oxford University Press.

[2] Tresp V. (2002) Committee Machines. In: Hu YH, Hwang JN (ed) Handbook of neural network signal processing, CRC Press LLC, pp 122–141.

[3] Drucker H. (1997) Fast Committee Machines for Regression and Classification. In: KDD-97 Proceedings.

[4] Hashem S. (1997) Optimal linear combinations of Neural Networks. Neural Networks 10(4): 599–614

[5] Breiman L. (1999) Combining predictors. In: Sharkey AJC (ed) Combining artificial neural nets: ensemble and modular multinet systems. Springer, Berlin Heidelberg New York, pp 31–50.

[6] Seni G. and Elder J. (2010) Ensemble Methods in Data Mining. Morgan & Claypool publishers.

[7] Rokach L. (2010) Ensemble-based classifiers. Artificial Intelligence Review 33, 1–39.

[8] Kargupta H. and Sivakumar K. (2004) Existential Pleasures of Distributed Data Mining. Data Mining: Next Generation Challenges and Future Directions, AAAI/MIT press.

[9] Aggarwal C.C. and Yu P.S. (2008) Privacy-Preserving Data Mining: Models and Algorithms, Kluwer Academic Publishers.

[10] Datta S., Bhaduri K., Giannella C., Wolff R., Kargupta H. (2006) Distributed data mining in peer-to-peer networks. IEEE Internet Computing 10(4), 18–26.

[11] Hussain I., Irakleous M., Siddiqi M.A. and Saraee M. (2010) Privacy-preserving data mining in peer to peer networks. In: proceedings of Annual International Conference on Data Analysis, Data Quality & Metadata Management (DAMD 2010), 14-15 June 2010, Singapore.

[12] Wu X. (2011) Research on Privacy Preservation in P2P Systems. International Journal of Advancements in Computing Technology 3(8), 324–330.

[13] Clifton C., Kantarcioglu M., Vaidya J., Lin X., Zhu M. (2003) Tools for Privacy Preserving Distributed Data Mining. ACM SIGKDD Explorations 4(2), 1–7

[14] Kantarcioglu M. and Vaidya J. (2003) Privacy-Preserving Naive Bayes Classifier for Horizontally Partitioned Data. In proceedings of IEEE Workshop on Privacy- Preserving Data Mining, 2008.

[15] Yi X. and Zhang Y. (2009) Privacy-preserving naïve Bayes classification on distributed data via semi-trusted mixers. Information Systems 34(3), 371–380.

[16] Yu H., Jiang X. and Vaidya J. (2006) Privacy-Preserving SVM using nonlinear Kernels on Horizontally Partitioned Data. In: Proceedings of SAC Conference.

[17] Xiong L., Chitti S. and Liu L. (2006) k nearest neighbour classification across multiple private databases. In: Proceedings of the ACM Fifteenth Conference on Information and Knowledge Management, 5-11 November, 2006.

[18] Poggio T. and Girosi F. (1990) Regularization algorithms for learning that are equivalent to multilayer networks. Science 247, 978–982.

[19] Girosi F., Jones M., and Poggio T. (1995) Regularization theory and neural networks architectures. Neural Computation 7, 21–269.

[20] Evgeniou T., Pontil M. and Poggio T. (2000) Regularization Networks and Support Vector Machines. Advances in Computational Mathematics, 13, 1–50.

[21] Poggio T. and Smale S. (2003) The mathematics of learning: Dealing with data. Notices of the American Mathematical Society 50(5), 537–544.

[22] Wang L. and Fu X. (2005) Data Mining with Computational Intelligence. Springer-Verlag.

[23] Kokkinos Y. and Margaritis K.G. (2012) A Regularization Network committee machine of isolated Regularization Networks for distributed privacy preserving data mining. In 8th International Conference on Artificial Intelligence Applications and Innovations, (AIAI 2012), Springer/ Lecture Notes in Computer Science, IFIP AICT 381, pp. 97–106..

[24] Bottou L., Chapelle O., DeCoste D. and Weston J. (2007) Large Scale Kernel Machines. Neural Information Processing Series, MIT Press, Cambridge, MA.

[25] Kashima H., Ide T., Kato T. and Sugiyama M. (2009) Recent Advances and Trends in Large-scale Kernel Methods. IEICE Transactions on Information and systems, E92-D(7), 1338–1353.

[26] Kuncheva L.I. (2004) Combining Pattern Classifiers: Methods and Algorithms. Wiley Interscience.

[27] Hansen L.K. and Salamon P. (1990) Neural network ensembles. IEEE Transactions on Pattern Analysis and Machine Intelligence, 12, 993–1001.

[28] Krogh A. and Vedelsby J. (1995) Neural network ensembles, cross validation and active learning. In: Advances in Neural Information Processing Systems (7), MIT Press, Cambridge, MA.

[29] Perrone M.P. and Cooper L.N. (1993) When networks disagree: ensemble method for neural networks. In: Mammone R.J., (Ed) Neural Networks for Speech and Image Processing, Chapman & Hall, Boca Raton, FL.

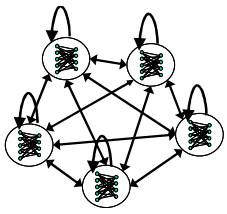
[30] Jain A.K., Duin R.P.W. and Mao J. (2000) Statistical Pattern Recognition: A Review. IEEE Transactions on Pattern Analysis and Machine Intelligence, 22(1), 4–37.

[31] Wilson G. (1995) Parallel Programming for Scientists and Engineers. MIT Press: Cambridge.

[32] Wolpert D.H. (1992) Stacked generalization. Neural Networks 5(2), 241–259.

8 Distributed neural network Ensemble Selection via Confidence ratio Affinity Propagation

This chapter considers distributed neural network ensemble selection for privacy-preserving data mining in large decentralized data locations which can build several neural networks to form an ensemble. The best neural network classifiers are selected via the proposed confidence ratio affinity propagation in an asynchronous distributed and privacy-preserving computing cycle. Existing methods usually need a shared to all classifiers dataset, in order to examine the classification accuracy of each pair of classifiers. This process is neither distributed nor privacy-preserving. On the other hand in the proposed distributed privacy-preserving solution the classifiers validate each other in a distributed way. The training set of one classifier becomes the validation set of the other and vice versa and only partial sums of confidences for the correctly and the falsely classified examples are collected.



By locally defining a confidence ratio between each pair of classifiers the well known affinity propagation algorithm finds the most representative ones. The construction is parallelizable and the cost is O($L \cdot N$) for L classifiers and N examples. A-priori knowledge for the number of best classifiers is not required since in affinity propagation algorithm this number emerges automatically. Experimental simulations on benchmark datasets and comparisons with other pair-wise diversity based measures and other existing pruning methods are promising.

8.1 Introduction

Distributed data mining tasks [1][2][3][4] refer to the discovery of potentially useful patterns from large physically distributed data banks. Collecting large data volumes to a single location for centralized data mining is usually unfeasible. The reasons for decentralization lay in the huge communication costs, computation costs, network bandwidth, central storage requirements, main memory demands and privacy preservation. Distributed data mining [1-4] is challenging due to the large number of distributed data sources, the dynamic character of data and the privacy-preserving issues that concern the participants. Distributed data mining focuses on developing efficient algorithms for mining patterns or information from distributed (usually disjoint) datasets without the need to centralizing them and sometimes without the need to reveal them to others [4].

Existing distributed data mining approaches vary. One versatile approach is to keep the disjoint datasets to their locations and perform, in parallel, local data mining to produce local models [5] [6] [7]. According to this advanced distributed data mining scenario the local models are those that can be transmitted to a central site that combines them into an ensemble, or global model. A second approach is sub-sampling a representative subset of data from each local site and accumulating these subsets to a central site in order to form a global subset. If this representative subset is close to the overall data distribution, then centralized data mining algorithms can be straightforwardly carried out on it, although in some cases the sub-sampling on huge datasets could produce very large subsets and the original scalability problem will remain [6]. A third approach is to create a meta-learner [8] from the ensemble, sometimes by combining the first and second approaches. Distributed data mining via several meta-learning methodologies [9] [10] [11] [12] split the dataset into different sites, train a classifier on each site and then post-train a non-linear combining or pruning scheme for the ensemble members [13] [14] [15], by using their prediction outputs from an independent evaluation set.

A fourth approach belongs to fully decentralized distributed data mining algorithms [1] [2]. The participating locations can communicate directly with each other in a pair-wise fashion via message passing. Some operational characteristics desired for these highly decentralized data mining algorithms are [1] distributed (data stays on each site), scalable (can handle large numbers of data), communication efficient (if only point-to-point messages), lock-free (without locking mechanism for simultaneously broadcasting), asynchronous (without the need of synchronization points), decentralized (without the need of a server) and privacy-preserving (without revealing local data). Privacy concerns are those that restrict the transferring as well as the sharing of the sensitive data. In this work we suggest a neural network ensemble selection strategy that possesses a number of the aforementioned operational characteristics.

Without the privacy-preserving factor, or the asynchronous factor, things are easy and the literature is teeming of different ensemble selection methods. We discuss many of them in section 2. Privacy-preserving means that data exchange is hindered or restricted and thus a participant location must not acquire any extra knowledge of the other participant's data. So essentially they are not able to read other's data or produce an independent evaluation set, gathered from sub-sampling all locations, and compare their output estimations on it. Asynchronous means the lack of a synchronization mechanism or a central coordination. Hence, the essence of the proposed solution is based on plain asynchronous point-to-point message passing in a mutual validation cycle. The basic operation two participants in the ensemble can do is to exchange messages. Such a classical point-to-point one-directional communication is depicted in fig. 8.1. We then exploit the possibility of mutual validation for mapping all the individual neural network classifiers based on their local accuracy, by using only simple asynchronous pair-wise message exchanging, like send a classifier and receive performance.

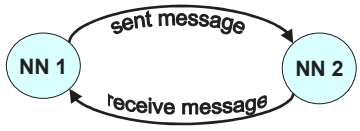


Figure 8.1. The classical asynchronous point-to-point one-directional message passing between two locations that hold a different neural network classifier.

The proposed solution for distributed privacy-preserving neural network ensemble selection consists of typical training the neural networks inside each location, message passing in a mutual validation cycle (classifiers are exchanged), creating the pair-wise similarities as locally defined confidence ratios that accumulate sums of confidences (we also compare with other pair-wise diversity based similarities), selecting the best classifiers by the affinity propagation clustering algorithm [19] which uses message passing (we also compare with other pruning methods).

Therefore, the contribution in this chapter is the proposed confidence ratio affinity propagation which can cope with the previously mentioned operational characteristics (asynchronous, lock-free communications, scalable, distributed privacy-preserving). We simply suggest using as key factors confidence ratios as pair-wise classifier similarities, a classical pair-wise computing cycle to locally compute these similarities between all classifier pairs, and the well known affinity propagation algorithm (see section 8.3). Preliminary experimental results on the last where reported in an earlier work [20]. The present work gives the detailed framework together with extensive comparisons with other pair-wise diversity based measures and other existing pruning methods. We cover the case of employing affinity propagation for distributed ensemble selection in view of the fact that this state-of-the-art algorithm had not been exploited so far. The confidence ratio is by nature a locally defined measure of similarity between two classifiers and depends only on their two local training datasets. It is produced via the plain point-to-point message passing illustrated in fig. 8.1. Hence, intrinsically the computations for the confidence ratios are: 1) distributed locally, 2) asynchronous and lock-free, 3) privacy-preserving since they leaving the local data banks intact without the need to share data from one another, 4) decentralized without collecting any data to a central location, 5) independently parallelizable.

In addition, the confidence ratio affinity propagation selects the best k neural network classifiers without the number k to be given in advance, and without the need for monitoring the pruned ensemble performance on a common to all validation set. It is parameter-free and automatically selects the best number of classifiers. Another advantage of the proposed method is the scalability that came from the independently parallel construction of the mutual validation matrix which is a result of the message passing computations. Given L classifiers and N training examples, distributed across L locations where each one holds N/L examples, the computational complexity of constructing the mutual validation matrix is reduced to $O(L^2N/L) = O(L \cdot N)$. Therefore the proposed solution is fast and scalable. The results demonstrate that the method automatically manages to select few classifiers and delivers a fast and accurate ensemble without the necessity for additional user input.

The rest of the chapter is organized as follows. Section 2 provides short literature review and background knowledge on the basic concepts of distributed privacy-preserving data mining and related work on neural network ensemble selection methods. Section 3 presents in details the proposed ensemble of regularization neural networks, elucidates our procedure of computations for the mutual validation, and describes the ensemble selection via the proposed confidence ratio affinity propagation and the combining via majority voting. Section 4 describes implementation details of many existing pair-wise diversity based measures that we also use as similarities to compare with. Section 5 presents the existing pruning methods we use in the comparisons. Section 6 provides experimental results and comparisons. Section 7 gives a discussion and section 8 summarizes our conclusions.

8.2 Background material

8.2.1 Distributed privacy-preserving data mining

A practical definition for 'distributed' is to learn without moving the local data to other locations and for 'privacy-preservation' is to learn without exposing the local data to any other. In a large scale distributed system that composed of several disjoint data banks local data exchange is usually impractical. In addition, the free flow of information is often prohibited by legal obligations or personal concerns. The participants may wish to collaborate but might not fully trust each other. Then distributed privacy-preserving data mining is the how to build valid data mining models and find meaningful patterns without disclosing any private information among the participants. Distributed privacy-preserving data mining is devoted to many real life applications and practical implementations. Few examples of applications [3] are privacy-preservation in personalized systems (newspapers, catalogs, etc.), privacy-preserving medical data mining, genomic privacy, privacy-preserving recommendation systems as well as applications in security-control, intrusion detection and surveillance.

The basic problem to be solved has a simple definition. In a large cooperative environment each participating node has a private input $\mathbf{x}_i$. All nodes wish to collaborative in order to jointly compute the output $f(\mathbf{x}_1, \mathbf{x}_2,\ldots, \mathbf{x}_n)$ of some function f while at the end of the process nothing but the output should have been exposed. Then distributed privacy-preserving data mining solves this problem by allowing nodes to safely share data or extracting useful data patterns without revealing any sensitive information, mainly employing ideas from secure multiparty computation. The secure sum is often given as a simple paradigm of the secure multiparty computations [4]. The secure sum protocol [4] works by computing the sum of a list of numbers without revealing anything but the output sum. The private values or data records are statistically manipulated by random data perturbation techniques to protect the privacy of individuals. Thus the aim of secure multiparty computation becomes the same with that of a private computation. That is a computation is secure if at the end no party knows anything except its own input and the final result. Classifiers which need total sums like Naive Bayes can be worked in this fashion. Classifier examples that have been generalized to the distributed privacy-preserving data mining problem are the Naïve Bayes classifier [21] [22], the SVM classifier with nonlinear kernels [23] and the k-nearest neighbour [24].

Large scale fully decentralized data mining may have differences from the traditional distributed data mining. For instance, there cannot be any global synchronization or a central access point. Nodes must be able to act very independently from each other on the basis of information exchange. Many paradigms like the secure sum protocol or the secure union require synchronization points and thus unlikely to scale very well in large network systems. In addition, some times privacy is not equal to security. Secure computations only deal with the process of computing a function which can be different for distributed problems. Not every distributed privacy problem can be cast as a distributed secure computation. In this work we use average classification rates in between pairs of participating locations, in order to compute matrices composed of pair-wise partial sums as elements. This leaves the participants with no knowledge for the other's data.

8.2.2 Neural Network Ensemble Selection methods

Neural Networks [12] are very good in dealing with noisy data, excellent in predictive accuracy, but rather poor in large scaling. However, if the data are physically distributed in several locations, then scalability issues and large training times can be intrinsically resolved by building, in parallel, several much smaller local neural network classifiers using the disjoint data sub-sets. Thus by utilizing ensembles [13] [14] [15] of neural networks the whole process can be significantly accelerated. In this work we employ Regularization Networks [16] [17] [18] which are known to use as hidden kernel neurons the actual points from real data. This is very useful when data features may have discrete values, e.g., in cases of data mining [12].

Ensembles of neural network classifiers have many advantages. Ensembles [13] [14] [15] work because uncorrelated errors of individual classifiers can be eliminated through averaging. Combining multiple neural network models into one ensemble is a strategy that increases accuracy since an ensemble is generally better than a single model. Such neural network ensembles are known to have excellent prediction capabilities. The construction of an ensemble composed of physically distributed neural network classifiers can use various ensemble learning techniques [13] [14] [15].

Each neural network member of the ensemble produces its own result for a newly arrived unknown example, and all combine their estimations to form the global decision for the class label. While ensembles in centralized datasets can often be created via bagging or boosting [13] [14], in the case of physically distributed data locations each location builds its own neural network and all of them comprise the ensemble. Ensemble selection is crucial. In a large scale pool of neural network members we need to select a subset of most important members in order to reduce computational cost, to provide scalability, to prune those members that have high correlated errors among each other and to maintain a high diversity among them. Therefore a neural network ensemble selection phase [13] [14] [15], also called ensemble pruning [25] [26] [27] [28] [29] or thinning, may precede the combination phase. This search for the best k neural network classifiers among L ones is actually a combinatorial problem of identifying the optimal sub-set of ensemble members and can be shown to be NP complete [29]. The k-classifiers must be highly diverse and as accurate as possible. Numerous strategies have been proposed for this selection/pruning problem.

If there are no privacy-preserving constraints then things are straightforward and the literature grows richer and richer on different ensemble selection methods emerging over the recent years. Proposed methods like diversity-based pruning [10], kappa pruning [25], reduced error (forward selection) pruning [25], test and select [26], reinforced learning [27], pruning by statistical tests [28], margin distribution based bagging pruning [30], orientation ordering [31] [32] and clustering based selection [33] [34] [35] [36] [37] [38] [39] [40] have been studied extensively.

In the case of clustering-based ensemble selection [33] [34] [35] [36], the best classifiers are chosen by clustering a larger set of them. By applying clustering to the classifier models in the ensemble one can select a single representative classifier for every cluster [7] that has been identified. If a proper distance measure between classifiers is defined then many clustering algorithms can be used like clustering by deterministic annealing [36], the k-means in [37] [38], agglomerative hierarchical clustering in [7] [33] [34] [39], affinity propagation clustering algorithm in [40], where the methods produces a large

number of neural network classifiers and cluster them according to their diversity. We choose the selection by affinity propagation clustering algorithm [19] since it is the best known unsupervised method for selecting the top k-centers (exemplars) from L total objects, which like any problem where the k is not given in advance, is NP complete also [19].

To define the diversity or any distance measure between two neural networks that cannot share any of their data, is an issue. In principle the typical hold-out evaluation set is lacking. However pair-wise diversity-based measures or distances can be defined from their partial sums in cases where a location receives the two classifiers then compares them on its local data and sends back their partial sum. Partial sums preserve the privacy. Thus many existing pair-wise diversity-based measures we compare with (see section 4) can actually be computed in the same pair-wise computing cycle we use here.

To work around many restrictions the proposed method uses solely message passing and mutual validation. The test set of one classifier becomes the local training set of the other. This way a location is not obligated to sent or share its private local data. It only shares its own locally trained classifier. The usage of message passing makes the nature of the proposed method intrinsically lock-free and asynchronous which renders it well fitted for distributed privacy-preserving data mining where employing neural network ensembles is scarcely exploited so far.

8.3 Pruning an Ensemble of Regularization Networks

In the mathematical formulas that follow an uppercase bold letter will symbolize a matrix, while a lowercase bold letter will symbolize a vector, and an italic letter will denote a scalar variable. We assume that a training set $\{\mathbf{x}_n, y_n\}_{n=1}^{N}$ is held distributed in the ensemble of L, in number, locations. Each p^{th} location has locally a disjoint data partition N_p where $N = N_1+N_2+\ldots+N_L$. The number of classes is M. Thus each p^{th} location holds $N_p=N/L$ training examples on average and can train and maintain a neural network classifier, denoted by $f_p()$. An ensemble composed of Regularization Networks [16] [17] [18] is illustrated in fig. 8.2.

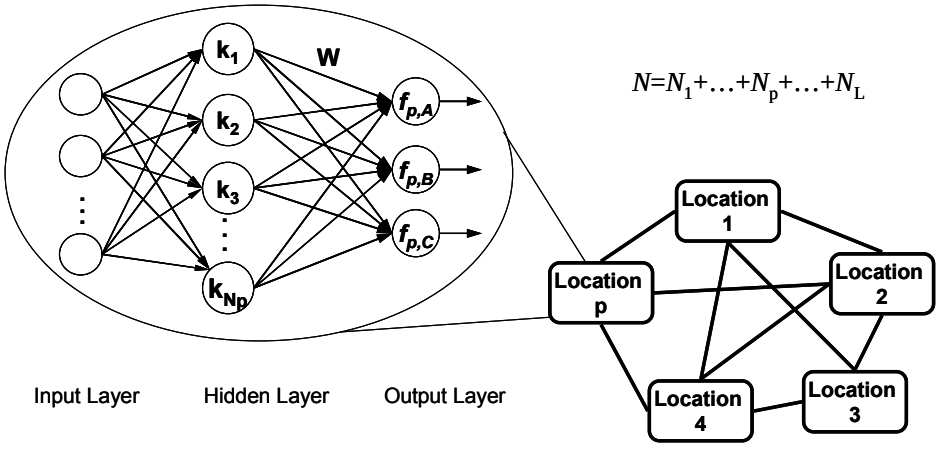


Figure 8.2. An Ensemble of Regularization Neural Network classifiers (for the three-class problem) distributed in L locations each one having a different local data partition N_p.

Each Regularization Network $f_p()$ in fig. 8.2 holds a local training subset $\{\mathbf{x}_n, y_n\}_{n=1}^{N_p}$ different from the others, and has an input layer, a hidden layer with N_p neurons of kernel units and an output layer for the class output predictions. It

performs a nonlinear mapping from the input space to the hidden layer followed by a linear mapping from the hidden space to the output space. The Regularization Networks are kernel based classifiers known to use as hidden neuron centers all the real training examples, to form the kernel functions and capture the data closeness of the underlined problem distribution. Regularization Networks are proven robust machine learning tools over the years and are used extensively for classification.

The proposed scheme for distributed privacy-preserving ensemble selection is summarized in the following steps:

1) Training the Regularization Networks: Each location constructs a local classifier model, which is trained on its local data patterns.

2) Message passing pair-wise computations for mutual validations: Each location sends to others its own classifier as a black box, in the form of a binary or an agent (classifiers are exchanged). Then each location tests the received classifiers on its local data, computes the local errors and sends back partial sums of confidences for the falsely classified and the correctly classified patterns respectively (we also use mutual validations to compute the partial sums of other diversity-based measures needed for comparison).

3) Locally defined confidence ratios for map the pair-wise similarities: For each pair of classifiers $\{f_i,f_j\}$ the ratio (sum of confidences for their falsely classified patterns) / (sum of confidences for their correctly classified patterns) defines an entry for the proposed pair-wise similarity matrix **S** (we also compare with other pair-wise diversity-based similarity matrices) for all the pairs of classifiers.

4) Ensemble selection by confidence ratio Affinity Propagation: we employ the Affinity Propagation clustering algorithm (we also compare with other pruning methods) in which we use as input the similarity matrix **S** from the previous step, in order to find the best local Regularization Network models.

5) Combine the selected Peer classifiers: a typical combination rule like majority voting is applied to form the final ensemble.

The L population can be very large. Distributed computing has generally two types, namely task-parallel where the processes are distributed across the nodes and data-parallel where the data are distributed across the nodes. This ensemble approach is clearly hybrid, both task as well as data parallel. Tasks are the classifiers which travel across the distributed system and when arriving in a location they are applied to local data.

8.3.1 Training the distributed Regularization Networks

The input neurons in each Regularization Network (see fig. 8.2) are as many as the number of data features. The hidden neurons are as many as the number of the local training instances. The output neurons correspond to the classes. Each location p has N_p training instances different from the other locations. The hidden-to-output layer weights w_n are computed for each class-output by solving a regularized risk functional. Assuming a given local training set $\{\mathbf{x}_n,y_n\}_{n=1}^{Np}$ of size N_p, a kernel activation function $k(\mathbf{x}_n, \mathbf{x})$ and a kernel matrix **K** (of size $N_p \times N_p$), a supervised learning task is to find an optimum nonlinear mapping function with proper weights **W**. For the three-class case in fig. 8.2 the matrix **W** equals $[\mathbf{w}_A\ \mathbf{w}_B\ \mathbf{w}_C]^T$. The output estimation of the p^{th} classifier for the m^{th} class of an unknown vector **x** is given by:

$$f_{p,m}(\mathbf{x}) = \sum_{n=1}^{N_p} w_{m,n} \cdot k(\mathbf{x}_n, \mathbf{x}) \tag{8.1}$$

where the kernel function is usually the Gaussian kernel $k(\mathbf{x}_n, \mathbf{x})=\exp(-||\mathbf{x}_n-\mathbf{x}||^2/\sigma^2)$ centered at a particular training point $\mathbf{x}_n$.

We additionally define $h_{p,m}(\mathbf{x})$ as the soft max confidence of the p^{th} classifier that $\mathbf{x}$ example belongs to class m.

A class label prediction is given in terms of the M class outputs $f_{p,m}(\mathbf{x})$ by:

$$\hat{y}_p(\mathbf{x}) = \arg\max_{m=1}^{M} \left(f_{p,m}(\mathbf{x}) \right) \tag{8.2}$$

The local kernel matrix $\mathbf{K}$ that has entries $\mathbf{K}_{ij} = k(\mathbf{x}_i,\mathbf{x}_j)$, created from the local data, contains all the information concerning the high density regions inside classes and the separating planes in between classes. The kernel matrix $\mathbf{K}$ is assumed to be continuous and positive semi-definite, for a finite set of training points. Then the Reproducing Kernel Hilbert Space (RKHS) H_K associated with the kernel matrix $\mathbf{K}$ is well defined, and the learning problem is stated as a minimization [16] [17] [18] of a regularized functional (eq. 8.3):

$$\arg\min_{f \in H_K} \left\{ \frac{1}{N_p} \sum_{n=1}^{Np} (y_n - f_p(\mathbf{x}_n))^2 + \gamma \left\| f_p \right\|_K^2 \right\} \tag{8.3}$$

Thus the hidden layer weights for the Regularization Network [16] [17] [18] are given by the solution of a minimization problem of the usual data term plus a second regularization term that plays the role of the stabilizer [16] [17] [18]. The data term is scaled proportionally to the number N_p of data points, and $\gamma > 0$ is a regularization parameter.

In each local Regularization Network the solution of the minimization is unique. For the three class problem in fig. 8.2 the weights $[\mathbf{w}_A \ \mathbf{w}_B \ \mathbf{w}_C]^T$ are given by solving the linear system:

$$\mathbf{w}_A = (\mathbf{K} + N_p \gamma \mathbf{I})^{-1} \mathbf{y}_A \tag{8.4a}$$

$$\mathbf{w}_B = (\mathbf{K} + N_p \gamma \mathbf{I})^{-1} \mathbf{y}_B \tag{8.4b}$$

$$\mathbf{w}_C = (\mathbf{K} + N_p \gamma \mathbf{I})^{-1} \mathbf{y}_C \tag{8.4c}$$

where $\mathbf{I}$ is the identity matrix, $\mathbf{K}$ is the kernel matrix (of size $N_p \times N_p$) and $\mathbf{y}_A$, $\mathbf{y}_B$ and $\mathbf{y}_C$ are the vectors (of size N_p) that hold the desired labels ($y_1, y_2, \ldots, y_{Np}$) for each class (in the hot encoding a desired label 1 is used for the same class examples and 0 for the others). The regularization parameter $N_p\gamma$ is usually small and can be found during training via cross validation. In the experimental runs we use a typical value $N_p\gamma = 0.1$ for all the local Regularization Networks.

For the M-class problem there are M weight vectors. Note that if in the ensemble an individual classifier does not posses any training data from a particular class then it still retains M outputs, where naturally for this missing class all weights are zero. This technicality is purely for compatibility among them.

8.3.2 Message passing computations for mutual validations

Initially L neural network classifiers are constructed from L disjoint data partitions of the dataset. The following scheme for selecting the best classifiers among them is also

applicable to other type of neural networks. Ensemble classifiers are treated as black boxes. Generally, in the distributed privacy-preserving case there exist no independent dataset, common to all classifiers, in order to use it for evaluating their performance. Although data exchange is hindered, the classifiers can be transmitted to other locations. The strategy for all-to-all exchange of classifiers between the different sites has been used many times in the literature [5][7][10][11][37]. This strategy can compensate for the lack of sharing data and the absence of a common to all classifiers validation dataset.

Typically each location has a network address and can send messages to others. The communicating messages can be hindered by arbitrary delays, and they can be lost as well in the way to their destination. These are the reasons which dictate that merely point-to-point types of computations, like message passing, are feasible within this distributed environment. In addition, the privacy concerns, which prevent the different locations to exchange or contribute any even smallest part of their data to other location sites, are extra constraints. To that end a simple fully distributed and privacy-preserving computation of two mutual validation matrices is the basis for the proposed ensemble selection approach. Provided that only Regularization Network classifiers are being sent, as binaries or agents, to other locations in order to validate their local data examples, then only average learning rates can be returned back to fill in the mutual validation matrices. In this way the training set from one classifier becomes the validation set of the other and vice versa. Those pair-wise mutual validations can finally construct a two-by-two mapping of all the pairs of Regularization Networks of the ensemble to a mutual validation matrix by making use of simple one directional point-to-point communication messages.

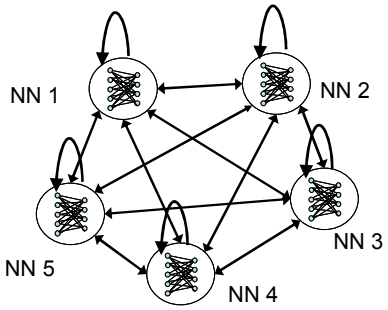


Figure 8.3. A graphical representation shows message passing point-to-point communications in the neural network ensemble. Send classifier and receive performance. This mutual validation maps the ensemble members with pair wise similarities between them.

Fig. 8.3 provides an illustrative example of the communications involve in the proposed procedure. The diagonal entries of the mutual validation matrix are the self-validation of each classifier. Distributed extensive computations are required for all non-diagonal entries of asymmetric validations between them across the network. The matrix is filled in with values produced by each location which continually executes a classical pair-wise computing cycle that composed of typical commands like:

(1) Send the local Regularization Network classifier to one location

(2) Check for a received classifier in the queue (of limited size)

(3) Apply a received classifier to the local data examples

(4) Send back only the partial sums (of errors, or confidences, or diversities)

(5) Send the local classifier to another location

A location receives classifiers or partial sums of performance. All received values of partial sums are automatically stored in an array. However due to the limited queue size not all received classifiers can be temporary maintained in such a queue. Having only send-receive commands to-from a single location the communication model in the computing cycle is a simple point-to-point. For a received Regularization Network classifier each location sends back two simple confidence sums, for the falsely classified local examples and the correctly classified ones respectively.

All the pair-wise diversity measures we will examine and compare can be computed in this pair-wise message passing cycle.

8.3.3 Proposed locally defined confidence ratios for pair-wise similarities

During the message passing cycle the partial sums of the results from the local validations are transmitted back. The pair-wise similarity $\mathbf{S}(i,j)$ between two classifiers $\{f_i,f_j\}$ must be defined from these mutual validations. For a problem with M classes, all types of classifiers can output one class label $\{1, 2,\ldots, M\}$. In many cases like the neural network types a classifier can also provide the confidence for a class label prediction.

We first make use of two mutual validation matrices **U** and **V** which are finally created in order to locally define pair-wise similarities among the different data locations. **U** and **V** matrices have size $L\times L$. Their entries $\mathbf{U}(i,p)$ and $\mathbf{V}(i,p)$ correspond to partial sums of confidence that have been sent back. A value $\mathbf{U}(i,p)$ is the sum of (Bayesian) confidences for the falsely classified examples of classifier p when these are classified by the i^{th} classifier. A value $\mathbf{V}(i,p)$ is the sum of confidences for the correctly classified examples of p when classified by the i^{th} classifier. Using only these simple partial sums all the private data vectors are stay unrevealed. In this way privacy-preserving for local data is achieved.

For M classes an i^{th} classifier $f_i()$ has M outputs $f_{i,m}()$ one for each class m, and can also produce confidence $h_{i,m}()$ for each prediction. Thus when it is applied in the N_p local data partition of the p^{th} classifier it produces the $\mathbf{U}(i,p)$ sums of confidences for the falsely classified examples and the $\mathbf{V}(i,p)$ sums of confidences for the correctly classified as follows (eq. 8.5):

$$\mathbf{U}(i,p) = \sum_{n}^{N_p} \{ \text{ if } y_n \neq \arg\max_{m=1}^{M}\left(f_{i,m}(\mathbf{x}_n) \right) \text{ then } h_{i,m}(\mathbf{x}_n) \text{ else } 0 \} \tag{8.5a}$$

$$\mathbf{V}(i,p) = \sum_{n}^{N_p} \{ \text{ if } y_n = \arg\max_{m=1}^{M}\left(f_{i,m}(\mathbf{x}_n) \right) \text{ then } h_{i,m}(\mathbf{x}_n) \text{ else } 0 \} \tag{8.5b}$$

$$h_{i,m}(\mathbf{x}_n) = \exp(f_{i.m}(\mathbf{x}_n)) / \left(\sum_{m=1}^{M} \exp(f_{i,m}(\mathbf{x}_n)) \right) \tag{8.5c}$$

where for the case of an i^{th} Neural Network and the class output $f_{i,m}(\mathbf{x}_n)$ the confidence is produced by the softmax function $h_{i,m}(\mathbf{x}_n)$.

Note that if a neural network does not posses any training data from a particular class then, since for compatibility reasons it retains M outputs, all the weights for the missing classes will be zero. In this case the exponentials $\exp(f_{i,m}(\mathbf{x}_n))$ of the missing classes outputs must not contribute to the sum in the denominator of the soft max function in eq. 8.5c. This is a minor technical detail that works. The missing class outputs always produce 0 and since exp(0)=1, where they should be 0 as well, only the

exclusion of them from the denominator's sum in eq. 8.5c can lead to the true confidence estimation.

The confidence must progressively increase for the correctly classified examples and decrease for the falsely classified. The previous partial sums of confidences in **U** and **V** can be combined in a confidence ratio which equals to (sum of confidences for falsely classified patterns) / (sum of confidences for correctly classified patterns). Such a local confidence ratio is used for each pair-wise similarity **S**(*i,j*) between two classifiers (the of-diagonal values). Note that because of the absence of a global dataset this local confidence ratio depends only on their two local training datasets N_i and N_j. Eq. 8.6a gives the locally defined confidence ratio similarities **S**(*i,j*) between two classifiers $\{f_i,f_j\}$ which are computed by combining only their local entries of **U**(*i,j*) and **V**(*i,j*). An average confidence ratio is used for the self-similarities **S**(*i,i*), the diagonal values (eq. 8.6b). Thus the confidence ratios are defined by (eq. 8.6):

$$\mathbf{S}(i,j) = -\left(1 + \mathbf{U}(i,j) + \mathbf{U}(j,i)\right) \,/\, \left(1 + \mathbf{V}(i,j) + \mathbf{V}(j,i)\right) \tag{8.6a}$$

$$\mathbf{S}(i,i) = -\left(1 + (1/L)\sum_{p=1}^{L} \mathbf{U}(i,p)\right) \,/\, \left(1 + (1/L)\sum_{p=1}^{L} \mathbf{V}(i,p)\right) \,/\, div \tag{8.6b}$$

Suppose that f_i classifies one example that belongs to f_j classifier. For this local definition there are four possible cases. Two of them are positive and two of them are negative. If f_i is correct and f_j is correct then they agree and they like each other (positive). If f_i is incorrect and f_j is incorrect then they agree but they are of no use for each other (negative). If f_i is correct and f_j is incorrect then they disagree but this is positive for f_j since the chance for this example to be classified correctly is increased (positive). If f_i is incorrect and f_j is correct then they disagree and this is negative for f_j since f_i will misguide the correct classification of f_j (negative). All the four cases reduce to eq. 8.6a. When classifier f_i is correct for the examples of the other classifier f_j then the sum of confidences for the correctly classified examples of f_j will increase. When classifier f_i is incorrect for the examples of the other classifier f_j then the sum of confidences for the falsely classified examples of f_j will increase.

The diagonal self-similarities **S**(*i,i*) express the 'preferences' that Affinity Propagation algorithm uses (see next section). Note also that if the i^{th} classifier makes no errors then its corresponding entry will still remains negative. Although in eq. 8.6 it seems that 1 is added in the numerators to keep them negative any tiny number can work. This negativeness is a requirement for the matrix **S** to be used as input for the affinity propagation algorithm that is presented next. In eq. 8.6b we employ the divisor *div* as a simple tuning parameter that can control the diagonal self-similarities, namely the preferences. Having a divisor larger that 1 and thus decreasing the preferences more classifiers can potentially be selected by the Affinity Propagation algorithm (see next section). In the experimental simulations for all the disjoint data partitions we kept fixed *div*=1.5 in all experiments.

Affinity Propagation algorithm takes as input such a similarity matrix **S**. For the non-diagonal entries of this matrix the algorithm can use typical pair-wise distances which simply reflect how close an object is to another. Locally a classifier must be as accurate as possible to another. Thus, for two classifiers their pair-wise classification error rate on their mutual local data must be small, and as a pair they must be mutually accurate to each other. This means locality for every pair of classifiers $\{f_i,f_j\}$ whose similarity **S**(*i,j*) depends only on their two local data partitions. For the diagonal entries of the similarity matrix **S**(*i,i*) Affinity Propagation needs the average similarities.

8.3.4 Ensemble selection by Affinity Propagation

Once the computing cycle via message passing mutual validation ends and the similarity matrices between classifiers are found, the ensemble selection is performed off-line. The best k classifiers must be found. Given a similarity matrix, Affinity Propagation (AP) of Frey and Dueck [19] is a state-of-the-art clustering algorithm that can select the best k-centers, called exemplars, by applying the message passing principle. This section describes particular details.

The AP algorithm tries to solve the k-centers problem. Formally the definition of the k-centers problem is: "from a set of data objects find the best representative k-centers so as to minimize the maximum distance from an object to its nearest exemplar center". AP algorithm determines the best k-centers from the real data objects. The number k is not given in advance since the AP algorithm does not require the number of exemplars to be pre-specified. Only a similarity matrix with values $\mathbf{S}(i,k)$, filled-in with negative similarities between the pairs (i,k) of objects, is needed as input. This matrix can be sparse. This matrix can be asymmetric as well, and the similarity measures can be of any kind and not confined only to metrics, or distance measures, thus making AP suitable for our case.

The first step of the AP algorithm is to choose a measure of similarity, $\mathbf{S}(i,k)$, between pairs (i,k) of data objects. Generally, as similarity, the negative of Euclidean distance is used. In our case, by using neural networks the similarities become the entries of the similarities matrix $\mathbf{S}$ we have constructed in the previous section (eq. 8.6).

The second step is the choice of the 'preferences' which are the diagonal values $\mathbf{S}(i,i)$ that denote the self-similarities. The preferences represent a measure of how much an object i is candidate to be an exemplar. Objects with larger diagonal values $\mathbf{S}(i,i)$ are more likely to be chosen as exemplars. Note that since all similarities are negative, larger values means less negative. The preferences choice is an issue, since self-distances in real data objects are zero. Typically AP proposes to set the initial value of all preferences equal to the mean of all similarities (resulting in a moderate number of exemplars) or to their minimum (resulting in a small number of exemplars). In the case of neural network classifiers these self-similarities can be based on their performance, like in eq. 8.6b.

AP algorithm [19] follows the same message passing principle of local point-to-point communications we have already see in fig. 8.1. It recursively transmits messages between pairs of objects until the best set of exemplars emerges. There are two kinds of messages being passed between object i and object k: Responsibilities $\mathbf{R}(i,k)$ and availabilities $\mathbf{A}(i,k)$ which both can be viewed as log-probability ratios. Responsibilities $\mathbf{R}(i,k)$ are sent from object i to candidate exemplar k in order to compute the max value of each row i. It is a measure that quantifies how well-suited k is to be the exemplar for object i, taking into account other potential exemplars for i. Availabilities $\mathbf{A}(i,k)$ are sent from candidate exemplar k to object i, in order to compute the sum of each column k. It is a measure that reflects the evidence for object i to choose k as its exemplar, considering that other objects may have k as an exemplar. At the beginning of the algorithm, the availabilities are initialized to zero and an additional round off in $\mathbf{S}(i,j)$ values is performed (by adding a tiny random number) so as to avoid numerical oscillations.

Throughout message updating [19] every message is set to λ times its value from the previous iteration, plus $(1 - \lambda)$ times the predefined updated values, where the

damping factor λ is between 0 and 1. Usually a default $\lambda = 0.5$ is used. An iteration of Affinity Propagation algorithm [19] consists of:

1) Updating all responsibilities $\mathbf{R}(i,k)$:

$$\mathbf{R}(i,k) = \lambda \cdot \mathbf{R}(i,k) + (1-\lambda) \cdot (\mathbf{S}(i,k) - \max_{k's.t.k' \neq k} \{\mathbf{A}(i,k') + \mathbf{S}(i,k')\}) \tag{8.7}$$

2) Updating all availabilities $\mathbf{A}(i,k)$:

$$\mathbf{A}(i,k) = \lambda \cdot \mathbf{A}(i,k) + (1-\lambda) \cdot \min\left\{ 0, \mathbf{R}(k,k) + \sum_{i's.t.i' \notin \{i,k\}} \max\{0, \mathbf{R}(i',k)\} \right\} \tag{8.8a}$$

$$\mathbf{A}(k,k) = \lambda \cdot \mathbf{A}(k,k) + (1-\lambda) \cdot \sum_{i's.t.i' \neq k} \max\{0, \mathbf{R}(i',k)\} \tag{8.8b}$$

3) Combining availabilities $\mathbf{A}(i,k)$ and responsibilities $\mathbf{R}(i,k)$ to check for exemplar decisions and the algorithm termination. These objects k with $\mathbf{A}(k,k) + \mathbf{R}(k,k) > 0$ are the identified exemplars.

If decisions made in step 3 do not change for a certain times of iteration or a fixed number of iteration reaches, the algorithm terminates.

An advantage of AP is that the number k of exemplars is not given in advance, but emerges from the message passing and depends on the preferences, the diagonals the diagonal values $\mathbf{S}(k,k)$ of the similarities matrix we previously defined. This permits the automatic model selection. In addition, the ability of AP to operate in problems where any pair-wise similarity can be defined between two objects makes the algorithm suitable for exploratory ensemble selection.

8.3.5 Combining by Majority Voting

The combining of the selected Regularization Networks classifiers follows the selection phase. The selected classifiers are called independently, in parallel, to classify an unknown example and their results are fused with a popular combination rule [13] [14] like majority voting, averaging, etc. We use majority voting since it belongs to the non-trainable case which requires no access to a common to all classifiers dataset like in the privacy-preserving case. Note that most of the existing ensemble pruning methods described in section 2 also use majority voting in their comparisons. The individual classifiers produce a single class label where in this case each classifier "votes" for a particular class, and the class with the majority vote on the ensemble wins. Frequently, regardless of number of classes and neural networks in the ensemble, a tie in voting may occur. If a tie occurs we accept the prediction among the tied classes that receives the maximum confidence sum [50]. With this confidence based strategy we experience slightly better results than the random class selection within tied classes, or the class selection using prior class probabilities. The strategy of tie-breaking using confidence simply receives pairs of {predicted class, confidence} from every neural network classifier. Thus every class accumulates a voting sum and a confidence sum. If a tie occurs then the tied class with the maximum confidence sum wins [50].

8.4 Comparison with other pair-wise diversity/similarity measures

For selecting the best classifiers we employ Affinity Propagation algorithm which although is a state-of-the-art method for the *k*-centers problem it needs appropriate similarity values for all the pairs of classifiers. To define a proper pair-wise (dis)similarity or distance between a pair of classifiers is an issue. Although we propose to use the locally defined confidence ratio, other well established (dis)similarity measures also exist. A comparison with them would be essential. Hence the (dis)similarity can be based on existing diversity measures which have been proposed in the literature in order to measure how diverse two classifiers are. Note however that the diversity measures presented in this section are of global nature since they need a global dataset to compare every pair of classifiers.

The most direct pair-wise diversity measure, given a pair of classifiers $\{f_i, f_k\}$ and a global dataset $\{\mathbf{x}_n, y_n\}_{n=1}^{N}$, is the sum of different label predictions, given by $d_{i,k}$ in eq. 8.9. This is the number of instances where f_i and f_k simply output different class label predictions $\hat{y}_i(\mathbf{x})$ and $\hat{y}_k(\mathbf{x}_n)$.

$$d_{i,k} = \sum_{n=1}^{N} dif_{i,k}(\mathbf{x}_n) \ , \ \{ dif_{i,k}(\mathbf{x}_n) = 0 \text{ if } \hat{y}_i(\mathbf{x}_n) = \hat{y}_k(\mathbf{x}_n) \text{ and } 1 \text{ otherwise} \} \tag{8.9}$$

In eq. 8.9 the sum of different label predictions diversity measure is unsupervised. It shows how much two classifiers disagree based solely on their outputs. The $d_{i,k}$ diversity measure was employed in the diversity-based pruning method of the JAM meta-learning framework [10] [11] as well as in diversity-based clustering classifiers for distributed data mining [7]. For L classifiers an $L \times L$ diversity matrix is formed with elements the $d_{i,k}$ values. Distributed systems afford to maintain such a coarse grained matrix which also preserves privacy. A location upon receiving two classifiers can simply compute a partial $d_{i,k}$ sum based on the local data it hold and store the result. The accumulation of the L such partial sums can form a global $d_{i,k}$ value.

The kappa statistic is another widely known unsupervised diversity measure. Kappa does not take into account the actual classification errors but rather whether the two classifiers agree more than expected by chance. This measure is the basis of the well known Kappa Pruning algorithm [25]. The Kappa statistic κ between two classifiers, with M classes and given N examples, is:

$$\kappa = \frac{\Theta_1 - \Theta_2}{1 - \Theta_2} \quad , \qquad \Theta_1 = \frac{\sum_{i=1}^{M} C_{ii}}{N}, \qquad \Theta_2 = \sum_{i=1}^{M} \left(\sum_{j=1}^{M} \frac{C_{ij}}{N} \cdot \sum_{j=1}^{M} \frac{C_{ji}}{N} \right) \tag{8.10}$$

where C is a contingency table of size $M \times M$ that contains the number of examples assigned to class i by the first classifier and to class j by the second classifier. Θ_1 is the probability they both agree and Θ_2 is the probability they agree by chance. The Kappa statistic values vary between 0 when the Θ_1 agreement equals that expected by chance and 1 when the two classifiers agree on every example.

The supervised pair-wise diversity measures [41] [42] need to record for each classifier and each example if the predicted class output is correct or incorrect. The first step given L classifiers and a global dataset $\{\mathbf{x}_n, y_n\}_{n=1}^{N}$ is to construct the ensemble error output matrix $\mathbf{E}$ of size $L \times N$, also called oracle output matrix [42], or signature matrix [31], that maps all classifiers to all examples. Fig. 8.4 illustrates the matrix $\mathbf{E}$. A value

$\mathbf{E}(i,j)$ = +1 if training example $\mathbf{x}_j$ is classified correctly by classifier f_i and $\mathbf{E}(i,j)$= –1 otherwise (fig. 8.4). The rows of matrix **E** correspond to i^{th} classifiers and the columns to j^{th} examples. Thus, the matrix **E** has L row vectors, $\mathbf{E} = [\mathbf{e}_1, \mathbf{e}_2, \ldots, \mathbf{e}_L]^T$, the N-dimensional signature vectors $\mathbf{e}_i$ (this term coined in [31][32] most suitably characterizes the rows).

	dataset 1			dataset 2			dataset 3			dataset 4		
	$\mathbf{x}_1$	$\mathbf{x}_2$	$\mathbf{x}_3$	$\mathbf{x}_4$	$\mathbf{x}_5$	$\mathbf{x}_6$	$\mathbf{x}_7$	$\mathbf{x}_8$	$\mathbf{x}_9$	$\mathbf{x}_{10}$	$\mathbf{x}_{11}$	$\mathbf{x}_{12}$
classifier1	+1	+1	+1	–1	+1	–1	+1	+1	–1	1	+1	–1
classifier 2	+1	–1	+1	+1	+1	+1	–1	+1	–1	+1	+1	+1
classifier 3	–1	+1	+1	–1	+1	–1	+1	–1	+1	–1	–1	+1
classifier 4	–1	+1	–1	+1	+1	+1	+1	+1	+1	+1	+1	–1
Partial sums	$n_1(\,,\,)$			$n_2(\,,\,)$			$n_3(\,,\,)$			$n_4(\,,\,)$		

Figure 8.4. Ensemble error output matrix **E** (or signature matrix) for L=4 classifiers and N=12 examples (the global dataset). Examples are distributed evenly and each location holds 3 of them. The matrix **E** maps the correct and incorrect classifications. Thus the columns of **E** are held distributed over the locations. Columns outlined in bold show the portions of rows each location holds. The diversity measures in this section validate every pair of classifiers by using the whole set of examples $\{\mathbf{x}_1, \mathbf{x}_2,..,\mathbf{x}_{12}\}$ each time.

Note here that in order for the supervised diversity measures to work the locations need an extra storage space to maintain their part of the Ensemble error output matrix **E** in fig. 8.4. In principle the similarities between two classifiers based on the supervised measures are composed of pair-wise partial sums (fig. 8.4) that can preserve the privacy. Among the supervised pair-wise diversity measures are the Yule's Q statistic, the correlation coefficient, the disagreement measure and the double fault measure. We try the last two since they are used in similar works. These measures count for each classifier pair $\{f_i,f_k\}$ the number of examples that: I) both f_i and f_k classify correctly, II) both f_i and f_k classify wrongly (make coincident errors), III) f_i classifies correctly and f_k classifies falsely, IV) f_i classifies falsely and f_k classifies correctly. Denoting these numbers by $n(1,1)$, $n(-1,-1)$, $n(1,-1)$, and $n(-1,1)$ respectively (see [42] for details) the supervised pair-wise diversity measures are given as follows.

The Disagreement measure $Dis_{i,k}$ is the portion of observations on which one classifier is correct and the other is incorrect [41] [42]:

$$Dis_{i,k} = \frac{n(-1,1) + n(1,-1)}{n(1,1) + n(1,-1) + n(-1,1) + n(-1,-1)} \tag{8.11}$$

The Double fault measure is the portion of coincident errors between two classifiers [41] [42]:

$$DF_{i,k} = \frac{n(-1,-1)}{n(1,1) + n(1,-1) + n(-1,1) + n(-1,-1)} \tag{8.12}$$

The double fault measure was used in [33][34][39]. The double fault measure has been observed to have the highest correlation with the ensemble error. The pair-wise distance between two classifiers is defined in [33][34][39] as $dist(f_i,f_k) = 1 - DF_{i,k}$.

In the example in fig. 8.4 a pair of classifiers can compute $n(\,,\,)= n_1(\,,\,)+ n_2(\,,\,)+ n_3(\,,\,)+ n_4(\,,\,)$ from all partial sums respectively for each column. Hence, the presentation so far

illustrates in detail how the well-known diversity measures can be computed in a distributed privacy preserving way by using partial sums. For classification tasks further reading and references for ensemble design, pair-wise measures and diversified classifier models can be found in a recent survey [47]. The work in [48], for dynamic ensemble selection in bagging classifiers, establishes ensemble diversity based on a random reference classifier and then combines a diversity measure with the classifier competence on a defined neighborhood.

Defining pair-wise distances or measures might be easier in ensembles for regression tasks, like those in [36] [38] [40] [43] whose neural networks have one output, since in this case any pair-wise distance can be calculated directly. In [36] [38] [40] [43] the difference or distance $d(f_i(\mathbf{x}_n), f_k(\mathbf{x}_n))$ between two neural networks is defined from their model outputs $f_i(\mathbf{x})$ and $f_k(\mathbf{x})$ against the same input example $\mathbf{x}_n$. By using a global training dataset that has N input examples, one N-dimensional vector of these pair differences can be formed for each neural network pair $\{f_i, f_k\}$. Then the sum of squares (or any other distance) on the N-dimensional vector elements can represent the final pair-wise distance, which is suitable for regression ensembles that output a real value. Note that in this case, while the $\mathbf{x}_n$ input vectors are directly used, there is no need for their y_n target values, since the individual prediction errors are not taken into account. Hence these label-free distances are unsupervised and depend only on the model parameters of each neural network. On the other side, supervised approaches for regression tasks employ the ensemble error output matrix which has continuous error values, instead of the discrete values -1 and 1 we have previously see in classification. This traditional ensemble error matrix, that actually maps L regressors $f_i()$ and N examples $\mathbf{x}_n$ by using their individual prediction errors $e_{i,n}=(f_i(\mathbf{x}_n) - y_n)$, was used in [44] for pruning a regression ensemble suitable for exchange rates forecasting. In contrast to the aforementioned non-generative ensemble methods a generative method can actually generate or modify the ensemble members themselves by intervening in their training phase. A successful neural network ensemble learning algorithm which puts the ensemble error inside each individual neural network training phase is Negative Correlation learning [45][46]. Negative Correlation learning iteratively modifies the ensemble members and explicitly manages the diversity by using an extra correlation penalty term in the minimization of the cost function of each individual network in order to produce biased learners whose errors tend to be negatively correlated. However the method requires full access to the local training data and full access to the simultaneous training phase of the local neural networks, which is not suitable for the distributed privacy-preserving case.

In the experimental section we compare the locally defined confidence ratio we propose for the classifier similarity $\mathbf{S}(i,k)$ that is used as input in the Affinity Propagation algorithm against: sum of different predictions, Kappa statistic, disagreement measure, double fault measure and distances of signature vectors.

Note that the Affinity Propagation algorithm has as input a similarity matrix $\mathbf{S}$ with entries the negative similarities between all pairs of objects. By using the sum of different predictions $d_{i,k}$ the pair-wise negative similarities in the experiments for Affinity Propagation will simply become $\mathbf{S}(i,k) = - d_{i,k}$. Employing Kappa statistic one can define the distance as $\text{dist}(f_i,f_k) = 1 - \kappa$, and thus the negative similarities will be $\mathbf{S}(i,k) = \kappa - 1$. Likewise, the disagreement measure must give negative similarities of the form $\mathbf{S}(i,k) = - Dis_{i,k}$. In the same way the double fault measure gives $\mathbf{S}(i,k) = - (1 - DF_{i,k})$. Two classifiers f_i and f_k are further apart the smaller a negative value $\mathbf{S}(i,k)$ is. The

maximum $\mathbf{S}(i,k)$ value is 0 and occurs when f_i and f_k are the same. In addition, we also compare with Euclidean distances that are based directly on the ensemble error output matrix **E**. The distance of a pair of classifiers $\{f_i,f_k\}$ can be defined from the Euclidean distance of their N-dimensional signature vectors $\mathbf{e}_i$ and $\mathbf{e}_k$. Hence, as one more input matrix of negative similarities for the comparisons in the Affinity Propagation algorithm we also use $\mathbf{S}(i,k) = -EuclidDistance(\mathbf{e}_i, \mathbf{e}_k)$. Since the previous pair-wise distances must be defined for $L \times L$ classifier pairs by using N-dimensional signature vectors $\mathbf{e}_i$, the total cost for a pair-wise similarity matrix is $O(L^2N)$.

Notice that the computational process of the previous pair-wise diversity based similarities can preserve the privacy. These measures have simple summations that are composed of partial sums divided into L locations (see fig. 8.4). A location p by using its own local dataset N_p can compute its corresponding partial sums $n_p(1,1)$, $n_p(-1,-1)$, $n_p(1,-1)$, and $n_p(-1,1)$. Fig. 8.4 shows at the bottom these sums. The same holds when a location computes a partial $d_{i,k}$ for the sum of different label predictions, and partial sum of squared differences for any pair $(\mathbf{e}_i, \mathbf{e}_k)$ of N_p-dimensional signature vectors it holds to define their Euclidean distance. The aggregation of these local sums can construct the global similarity matrix and the data remain private. That is way we compare and test all of them as pair-wise similarities for the Affinity Propagation algorithm.

Given that the N examples are divided into L disjointed data partitions, each one having N/L examples, the proposed confidence ratio measure in eq. 8.6 has computational cost $O(L^2N/L)=O(LN)$ to fill-in all the diagonal elements of the **S** matrix, plus $O(L\ N)$ for all the of-diagonal ones that hold the pair-wise similarities. Table 8.1 shows the computational complexities for the pair-wise similarities that can be computed in a privacy-preserving way.

Table 8.1. Computational complexities needed for the pair-wise similarity entries.

Pair-wise similarity measure	**Complexity**
Proposed Confidence ratio (eq.8.6)	$O(L\ N)$
Kappa statistic	$O(L^2N\ M^2)$
Sums of different predictions	$O(L^2N)$
Disagreement	$O(L^2N)$
Double fault	$O(L^2N)$
Distances of signature vectors	$O(L^2N)$

Beyond the difference in computational complexity of the proposed confidence ratio as compared with the other measures, another different characteristic is the global nature of the other pair-wise diversity measures which they need, by definition, a global dataset with which they must compare all the pairs of classifiers. Thus while the proposed confidence ratio given two classifiers $\{f_i, f_j\}$ defines locally their pair-wise dissimilarity $\mathbf{S}(i,j)$ based on only their two local training datasets, N_i and N_j, (using eq.8.6), the other aforementioned measures in this section need to globally define the pair-wise dissimilarity of the pair $\{f_i, f_j\}$ based on a global training dataset N.

8.5 Comparison with other pruning methods

This section describes other existing pruning methods we use for comparison in the experimental simulations. We include approaches from meta-learning, ordered pruning and ranking based pruning mentioned earlier. All pruning methods will be compared side-by-side. Specifically we compare with the following: kappa pruning [25], diversity-based pruning of the original distributed JAM meta-learning system [10] [11], reduce-error pruning [25] and pruning by orientation ordering [31] [32]. Note however that these pruning methods differ from the proposed confidence ratio affinity propagation since they rank all classifier pairs in a global sense by using the entire training set, and more importantly they need the pruning number to be given in advance. In the experiments we pre-set Kappa pruning and JAM's diversity-based pruning to select 1/3 of classifiers, as suggested by other authors. Reduce-error pruning is an exhaustive search and we pre-set it to select as many classifiers found by the proposed method. In orientation ordering the pruning angle is $\pi/2$ which usually selects half of classifiers. Some details of the methods are presented next.

8.5.1 Kappa Pruning

The well known Kappa pruning [25] tries to maximize the pair-wise diversity among selected classifiers. It ranks all $L \times L$ pairs of classifiers by computing their kappa statistic using the training set. Its cost is $O(L^2 N)$ for L classifiers and N examples. Then kappa pruning choose pairs of classifiers starting with the pair that has the lowest kappa and continues choosing such pairs in increasing order until a predefined number of them is finally selected. Since the number of selected classifiers is required in advance we set kappa pruning to select a fixed portion equal to 1/3 of the initial L population.

8.5.2 JAM's diversity-based pruning meta-classifiers

The JAM's diversity-based pruning meta-classifiers algorithm [10] [11] tries to maximize the sum of pairwise diversity of the selected subset of classifiers. It works iteratively by selecting one classifier each time, starting with the most accurate one. Initially it computes the pair-wise diversity matrix where each cell $d_{i,k}$ contains the number of training examples for which the two classifiers f_i and f_k disagree (sum of different label predictions measure). In each round, the algorithm adds to the list of selected classifiers the classifier that is most diverse to the classifiers chosen so far. With N training examples in this diversity-based pruning meta-classifiers method [10] [11] the computation of the diversity matrix of all the $L \times L$ pairs of classifiers has $O(L^2 N)$ cost. Using diversity $d_{i,k}$ that composed of simple partial sums it preserves the privacy.

8.5.3 Reduce-error pruning

A forward selection with greedy search is reduced-error pruning [25]. Starting with the classifier with the minimum error this method grows the list of selected classifiers one at a time. Each step adds to the list the classifier whose combination with the others in the list gives the lowest possible error. In our case this error is measured on the training set, since a separate validation set is missing for the privacy-preserving case. We run the reduce error pruning after the confidence ratio affinity propagation in order to prune as many classifiers as the proposed method and to examine if the

selected ones are actually the best. The original version of reduce-error pruning also suggests a backfitting process [25]. Backfitting at each step sequentially swaps a selected classifier by an unselected one and by monitoring the performance of the list it keeps the combination with the lowest error. However this dramatically increases the execution time and does not significantly reduce the generalization error [31] [32]. Thus we compare with the version without backfitting as other studies also do [31] [32]. Although the greedy search manage to select the best classifiers this algorithm is extremely slow.

8.5.4 Orientation Ordering

An efficient and effective ranking-based pruning method is Orientation Ordering [31] [32], which directly uses the *N*-dimensional signature vectors we have seen in section 8.4. It employs the angles of these vectors with a reference vector and the complexity is O(*L N*) for *L* classifiers and *N* examples. Recall that for each classifier an element of its signature vector has value +1 if a training example is classified correctly by the corresponding classifier and –1 otherwise. The average signature vector of all classifiers in an ensemble is called the ensemble signature vector. The diagonal of the first quadrant of the *N*-dimensional hype-plane (i.e., all the components are positive) corresponds to the ideal classification performance. Then the reference vector is a vector perpendicular to the ensemble signature vector that corresponds to the projection of the first quadrant diagonal onto the hyper-plane defined by the ensemble signature vector. Orientation ordering ranks the classifiers by increasing value of their angle between their signature vector and the reference vector. As suggested by the authors [31][32] the fraction of classifiers whose angle is less than $\pi/2$ is finally selected.

8.6 Experimental Simulations

8.6.1 Benchmark Datasets

The classification performance is measured using benchmark datasets from the UCI data repository (http://www.ics.uci.edu/~mlearn/MLRepository.htm) and the KEEL data repository (http://sci2s.ugr.es/keel). The details of the datasets are in table 8.2.

Table 8.2. Benchmark dataset details

Dataset Name	instances	features	classes
Wine	178	13	3
Sonar	208	60	2
Dermatology	358	34	6
Wisconsin (Diagnostic)	683	9	2
Diabetes Pima Indians	768	8	2
Vehicle Silhouettes	846	18	4
Yeast	1479	8	9
Spambase	4601	57	2
Phoneme	5404	5	2
Page Blocks	5473	10	5
Optical Digits	5620	64	10
Satellite Image	6435	36	6
Ring	7400	20	2
Magic telescope	19020	10	2
Letter	20000	16	26
Shuttle	58000	9	7

8.6.2 Experimental Design

The groups of experiments show the selective power of the proposed method as well as the error rates. The different locations are assumed to have $N_p=N/L$ examples without the ability to move them or share them. The experimental design is as follows:

1. A dataset is randomly split into a training set (80%) and a test set (20%) with stratification.
2. The training set is divided equally but without stratification into L disjoint partitions ($N_p=N_1=N_2=\ldots=N_L$) which are distributed to the different locations (in the real situation those data are distributed permanently so this step is not a part of the training phase).
3. Every location p holds N_p data and builds a Regularization Neural Network classifier based on its own local examples.
4. A pair-wise computing cycle is executed to find all entries of the proposed affinity matrix, specifically the pair-wise confidence ratio for all classifiers.
5. The affinity propagation algorithm selects the best exemplar classifiers.
6. The final pruned ensemble is tested on the test set.

This procedure is repeated 30 times for each benchmark dataset and each corresponding classifier population. As usual, we measure the classification error rate. The proposed method is local in nature (both the confidence ratio similarity as well as the Affinity propagation message passing algorithm) while all the five diversity measures described in section 8.4 are global in nature, by definition, and they need a global dataset which should be common to all classifier pairs. Also for the same reason, that is they are based on globally defined diversity measures, the four well known pruning algorithms used in the comparisons are global in nature. Thus we compare the proposed locally defined method with nine globally defined methods. We find that the confidence ratio performs better than the other diversity measures in most of the cases illustrated in table 8.3.

8.6.3 Comparison of different similarity measures

Table 8.3 illustrates the classification errors of the ensemble after the ensemble selection by using Affinity propagation and different pair-wise similarity measures. In these performance comparisons we test the five pair-wise similarity measures as potential inputs in the Affinity Propagation algorithm for selecting the best neural network classifiers. Table 8.3 shows average classification errors. The average pruning sizes are shown in table 8.4. The dataset names are in the first column. The second column shows the initial ensemble size. The third column is the majority voting classification error by using all the Neural Networks members in the ensemble. In the fourth column Affinity Propagation uses the proposed aggregated confidence ratio based $\mathbf{S}(i,k)$ similarity matrix (see eq. 8.6). The other five similarities that are used in Affinity propagation for comparison are based on the diversity measures: kappa statistic with $\mathbf{S}(i,k) = \kappa - 1$, sum of different predictions with $\mathbf{S}(i,k) = - d_{i,k}$, disagreement measure with $\mathbf{S}(i,k) = - Dis_{i,k}$, double fault measure with $\mathbf{S}(i,k) = - (1 - DF_{i,k})$, Euclidean distance of signature vectors $\mathbf{S}(i,k) =- EuclidDistance(\mathbf{e}_i, \mathbf{e}_k)$ as indicated in the last five columns.

Table 8.3. Classification error rates of the ensemble after classifier selection by using Affinity propagation and different similarity measures. Standard deviations (±) are also illustrated for these errors as averaged for 30 runs. Best results appear in bold.

Dataset	Initial Ens *size*	Initial Ens. error	Confidence ratio (eq.8.6) Affinity Propagate	Kappa statistic Affinity Propagate	Different predictions Affinity Propagate	Disagree-ment Affinity Propagate	Double fault Affinity Propagate	Signature distance Affinity Propagate
Wine	*5*	**2.9**	3.0 ±2.9	3.3 ±2.8	2.9 ±2.4	3.0 ±2.3	3.0 ±2.6	**2.7** ±2.5
	10	**4.0**	4.1 ±2.2	5.5 ±3.5	7.1 ±5.2	6.5 ±4.7	4.5 ±3.1	5.4 ±3.5
	15	3.7	**3.7** ±2.9	9.0 ±5.5	8.7 ±5.8	7.7 ±5.6	6.5 ±4.6	5.7 ±3.3
Sonar	*5*	23.6	**21.0** ±6.1	22.1 ±5.5	**21.0** ±6.2	**21.0** ±6.2	21.2 ±6.4	23.6 ±5.4
	10	25.8	**25.1** ±5.5	27.5 ±7.0	26.9 ±6.1	27.4 ±6.0	27.2 ±6.5	27.0 ±6.2
	15	**26.5**	26.7 ±5.8	28.7 ±5.3	29.1 ±6.9	29.1 ±6.9	28.4 ±7.8	27.1 ±6.9
Dermato-logy	*5*	**3.2**	3.5 ±1.5	4.7 ±2.4	4.3 ±2.5	4.4 ±2.6	3.5 ±2.2	3.3 ±1.7
	10	3.5	**3.4** ±2.0	7.1 ±3.5	6.6 ±3.5	6.4 ±3.5	4.6 ±3.4	4.6 ±2.7
	15	**3.8**	4.0 ±2.3	7.2 ±3.7	6.7 ±3.2	6.7 ±3.6	5.6 ±2.5	6.5 ±3.9
Wisconsin	*10*	2.9	2.9 ±1.3	2.9 ±1.3	2.9 ±1.4	2.9 ±1.4	**2.8** ±1.3	3.0 ±1.3
	20	3.2	**3.0** ±1.4	3.3 ±1.4	3.5 ±1.5	3.6 ±1.5	**3.0** ±1.4	3.2 ±1.5
	30	3.5	**3.1** ±1.6	4.0 ±1.6	4.0 ±1.5	3.8 ±1.5	**3.1** ±1.6	3.8 ±1.6
Diabetes	*10*	24.0	24.2 ±2.8	24.5 ±3.1	25.0 ±3.2	25.1 ±3.2	24.2 ±3.2	**23.7** ±2.8
	20	**24.5**	25.4 ±2.9	26.5 ±2.6	26.0 ±2.6	26.0 ±2.6	26.6 ±2.8	26.2 ±2.5
	30	**26.1**	**26.1** ±3.1	27.1 ±3.0	26.7 ±3.0	26.6 ±3.0	27.2 ±3.2	26.9 ±3.0
Vehicle	*10*	28.5	29.1 ±3.2	28.0 ±3.1	28.5 ±3.2	29.1 ±3.3	29.2 ±3.0	**27.9** ±2.7
	15	30.5	**30.1** ±2.6	32.9 ±2.7	32.8 ±3.6	32.9 ±3.2	34.6 ±3.2	32.1 ±3.4
	20	**32.3**	32.4 ±2.8	34.2 ±3.1	34.5 ±2.6	34.7 ±3.1	36.6 ±3.6	33.8 ±3.1
Yeast	*20*	**41.7**	41.8 ±2.1	43.0 ±2.8	42.8 ±2.4	42.7 ±2.5	43.8 ±2.6	42.9 ±2.8
	30	**41.8**	42.0 ±2.4	44.2 ±2.8	44.3 ±2.6	44.6 ±2.4	46.3 ±3.5	43.8 ±2.7
	40	**42.8**	43.1 ±2.9	44.6 ±3.0	45.2 ±2.9	44.7 ±2.5	47.2 ±3.3	44.6 ±2.8
Spambase	*20*	**8.2**	8.5 ±0.8	8.8 ±0.9	8.8 ±0.9	8.7 ±0.9	8.7 ±0.9	8.7 ±0.9
	30	**8.9**	**8.9** ±0.7	9.5 ±0.9	9.3 ±0.9	9.4 ±0.9	9.7 ±0.9	9.4 ±0.8
	40	9.4	**9.2** ±1.0	10.2 ±1.1	10.1 ±1.0	10.1 ±1.1	10.3 ±1.1	10.0 ±1.1
Phoneme	*10*	**16.4**	16.6 ±0.8	16.6 ±0.7	16.6 ±0.8	16.6 ±0.8	**16.4** ±0.9	**16.4** ±0.8
	20	17.3	**17.2** ±1.1	17.9 ±1.0	17.8 ±1.2	17.8 ±1.2	18.0 ±1.3	17.7 ±1.0
	30	**18.1**	**18.1** ±1.0	18.5 ±1.1	18.5 ±1.0	18.5 ±1.0	19.0 ±1.1	18.5 ±1.0
Page blocks	*20*	**5.6**	**5.6** ±0.6	5.9 ±0.6	5.8 ±0.6	6.0 ±0.6	5.9 ±0.7	5.9 ±0.6
	40	6.3	**6.2** ±0.6	6.7 ±0.7	6.7 ±0.6	6.6 ±0.8	7.1 ±0.8	6.6 ±0.7
	60	7.2	**6.6** ±0.7	8.2 ±0.9	7.5 ±0.8	7.6 ±0.6	7.6 ±0.7	7.5 ±0.8
Optical Digits	*20*	2.1	**2.1** ±0.4	2.7 ±0.6	2.7 ±0.6	2.7 ±0.6	2.5 ±0.7	2.4 ±0.5
	30	2.6	**2.5** ±0.5	3.3 ±0.8	3.3 ±0.8	3.2 ±0.7	3.4 ±0.6	3.0 ±0.7
	40	**2.8**	**2.8** ±0.4	3.8 ±0.6	3.7 ±0.6	3.7 ±0.6	3.9 ±0.7	3.6 ±0.6
Satellite Image	*20*	11.1	**10.9** ±0.7	11.4 ±0.6	11.3 ±0.5	11.3 ±0.6	11.4 ±0.7	11.2 ±0.7
	30	11.8	**11.6** ±0.9	12.1 ±0.8	12.1 ±0.8	12.0 ±0.8	12.9 ±1.1	11.9 ±0.9
	40	**12.5**	**12.5** ±0.7	12.8 ±0.7	12.9 ±0.8	12.7 ±0.7	13.2 ±0.7	12.8 ±0.7

Ring	*20*	**4.0**	**4.0** ±0.5	4.9 ±0.5	4.8 ±0.6	4.8 ±0.6	4.4 ±0.6	4.6 ±0.5
	40	7.6	**6.8** ±0.6	9.1 ±0.9	8.8 ±0.8	8.8 ±0.9	8.5 ±1.1	8.9 ±0.8
	60	11.2	**10.5** ±0.7	14.0 ±1.4	13.5 ±1.3	13.5 ±1.3	14.4 ±2.2	13.6 ±1.2
Magic telescope	*20*	**13.8**	**13.8** ±0.5	14.1 ±0.5	14.2 ±0.5	14.1 ±0.5	14.0 ±0.5	14.0 ±0.5
	40	**14.2**	**14.2** ±0.4	14.6 ±0.5	14.5 ±0.5	14.6 ±0.5	14.7 ±0.5	14.5 ±0.4
	60	14.7	**14.6** ±0.5	15.0 ±0.5	15.1 ±0.5	15.1 ±0.5	15.5 ±0.5	15.1 ±0.5
Letter	*20*	**9.9**	**9.9** ±0.5	12.4 ±1.0	12.4 ±1.0	12.9 ±1.2	13.6 ±1.4	11.1 ±0.7
	40	**10.8**	**10.8** ±0.4	15.0 ±1.4	15.1 ±1.4	15.6 ±1.4	15.8 ±1.4	13.4 ±0.9
	60	**13.3**	**13.3** ±0.7	19.2 ±1.4	19.0 ±1.4	19.1 ±1.1	20.1 ±1.5	18.0 ±1.0
Shuttle	*20*	**3.1**	**3.1** ±0.2	**3.1** ±0.2	**3.1** ±0.2	**3.1** ±0.2	**3.1** ±0.2	**3.1** ±0.2
	40	4.0	**3.7** ±0.3	4.0 ±0.3	4.0 ±0.3	4.0 ±0.3	**3.7** ±0.3	4.0 ±0.3
	60	4.6	**3.9** ±0.3	4.5 ±0.2	4.5 ±0.2	4.5 ±0.3	3.9 ±0.2	4.5 ±0.2

The 48 comparison cases in table 8.3 reveal that the proposed confidence ratio Affinity Propagation ranks 1st 42 times, outperforming the other similarity measures. The kappa statistic in the fourth column ranks 1st 1 time. The sum of different predictions in the fifth column as well as the disagreement measure in the sixth column ranks 1st 2 times. In the seventh column the double fault measure ranks 1st 7 times while in the eighth column the negative distance of signature vectors ranks 1st 6 times. The proposed confidence ratio measure produces a substantially smaller error as compared with the other similarity measures.Hence the proposed confidence ratio (eq. 8.6) provides on average the lower error rates among the other similarity (diversity) measures and thus can become the first choice for the similarity matrix in the Affinity Propagation algorithm for ensemble selection.

Table 8.4. Pruning sizes of the ensemble after classifier selection by using Affinity propagation and different similarity measures.

Dataset	Initial *size*	Confidence ratio (eq.8.6) Affinity Propagate	Kappa statistic Affinity Propagate	Different predictions Affinity Propagate	Disagree-ment Affinity Propagate	Double fault Affinity Propagate	Signature distance Affinity Propagate
Wine	*5*	*2.1*	*3.0*	*3.0*	*3.0*	*3.0*	*4.0*
	10	*3.3*	*3.5*	*3.2*	*3.4*	*4.1*	*4.2*
	15	*4.1*	*4.2*	*4.0*	*4.0*	*4.5*	*5.0*
Sonar	*5*	*3.8*	*4.0*	*3.8*	*3.8*	*4.1*	*4.9*
	10	*5.2*	*4.5*	*3.9*	*3.9*	*3.7*	*5.4*
	15	*6.0*	*5.1*	*4.3*	*4.3*	*4.1*	*5.0*
Dermato-logy	*5*	*3.0*	*3.2*	*3.2*	*3.3*	*3.7*	*4.3*
	10	*4.4*	*3.4*	*3.3*	*3.4*	*4.5*	*4.4*
	15	*5.6*	*4.2*	*4.0*	*4.0*	*4.5*	*4.8*
Wisconsin	*10*	*3.8*	*3.8*	*3.7*	*3.8*	*4.2*	*4.4*
	20	*5.0*	*5.4*	*5.3*	*5.3*	*5.9*	*6.2*
	30	*6.3*	*6.8*	*6.8*	*9.0*	*8.4*	*8.0*
Diabetes	*10*	*4.1*	*4.6*	*4.4*	*4.3*	*4.8*	*6.8*
	20	*6.5*	*5.5*	*5.1*	*5.1*	*5.0*	*6.1*
	30	*9.5*	*7.6*	*7.3*	*7.3*	*7.2*	*9.0*
Vehicle	*10*	*5.5*	*5.6*	*5.3*	*5.0*	*5.0*	*8.1*
	15	*7.7*	*5.3*	*5.1*	*5.0*	*4.4*	*6.8*
	20	*9.3*	*5.8*	*5.6*	*5.2*	*4.7*	*6.5*

Yeast	*20*	*6.7*	*5.3*	*5.1*	*5.2*	*4.8*	*6.3*
	30	*11.4*	*6.3*	*6.0*	*6.0*	*5.5*	*7.4*
	40	*14.1*	*7.3*	*6.7*	*7.5*	*6.4*	*8.2*
Spambase	*20*	*6.3*	*4.8*	*4.8*	*4.8*	*5.6*	*6.9*
	30	*11.2*	*6.0*	*5.9*	*5.9*	*6.3*	*7.3*
	40	*13.2*	*7.2*	*7.0*	*7.0*	*6.9*	*7.8*
Phoneme	*10*	*1.7*	*4.5*	*4.5*	*4.5*	*4.8*	*6.8*
	20	*9.8*	*5.3*	*4.8*	*4.8*	*4.9*	*6.0*
	30	*15.4*	*6.4*	*6.2*	*6.2*	*6.3*	*7.0*
Page blocks	*20*	*4.7*	*4.8*	*4.6*	*4.5*	*4.2*	*5.2*
	40	*12.5*	*7.8*	*6.8*	*7.1*	*5.8*	*8.0*
	60	*18.9*	*10.8*	*8.5*	*8.9*	*7.6*	*10.2*
Optical Digits	*20*	*12.8*	*5.8*	*5.9*	*5.5*	*6.2*	*8.7*
	30	*16.7*	*5.6*	*5.6*	*5.6*	*5.7*	*7.9*
	40	*18.1*	*6.4*	*6.4*	*6.9*	*4.7*	*8.2*
Satellite Image	*20*	*15.3*	*6.1*	*6.1*	*6.0*	*5.5*	*8.2*
	30	*21.5*	*6.5*	*6.5*	*6.4*	*5.8*	*8.9*
	40	*24.5*	*7.3*	*7.1*	*7.2*	*6.3*	*8.5*
Ring	*20*	*6.1*	*5.4*	*5.5*	*5.5*	*6.0*	*9.3*
	40	*18.6*	*6.8*	*7.0*	*7.0*	*5.0*	*8.8*
	60	*29.6*	*8.8*	*8.7*	*8.7*	*4.5*	*10.0*
Magic telescope	*20*	*14.5*	*5.5*	*5.5*	*5.6*	*6.5*	*8.7*
	40	*35.1*	*6.6*	*6.6*	*6.8*	*7.4*	*8.0*
	60	*46.5*	*8.4*	*8.1*	*8.1*	*9.1*	*10.2*
Letter	*20*	*17.0*	*7.0*	*7.0*	*7.1*	*7.0*	*10.0*
	40	*38.0*	*7.5*	*7.4*	*7.3*	*7.2*	*10.2*
	60	*58.0*	*7.6*	*7.8*	*8.2*	*7.4*	*10.5*
Shuttle	*20*	*1.3*	*4.4*	*4.3*	*4.3*	*5.5*	*5.2*
	40	*2.0*	*6.3*	*6.2*	*6.3*	*6.7*	*7.9*
	60	*4.4*	*8.3*	*7.9*	*7.9*	*13.2*	*10.3*

8.6.4 Comparison with different pruning methods

In the second set of experiments the classification performance of the proposed method is compared against: Kappa Pruning, JAM's diversity-based pruning, Reduce-error pruning and Orientation Ordering. The error rates of the ensemble after classifier selection are illustrated in table 8.5 which shows that the proposed distributed ensemble selection using affinity propagation and confidence ratio (eq. 8.6) selects a small portion of classifiers and delivers comparable accuracy results with the other pruning algorithms.

Table 8.5. Error rates of the ensemble after classifier selection by using confidence ratio Affinity propagation versus the other pruning methods as indicated. Standard deviations (±) are also illustrated for these errors as averaged after 30 runs.

Dataset	Initial ensemble *size*	Confidence ratio (eq.8.6) Affinity Propagate	Kappa Pruning	JAM's diversity-based pruning	Reduce-error pruning	Orientation Ordering
Wine	*5*	**3.0** ±2.9	3.1 ±2.7	4.2 ±2.9	**3.0** ±2.5	**3.0** ±3.2
	10	**4.1** ±2.2	5.0 ±2.6	5.7 ±3.7	**4.1** ±2.2	**4.1** ±2.6
	15	3.7 ±2.9	5.4 ±3.7	5.4 ±3.3	3.7 ±2.8	**3.5** ±3.2

Sonar	*5*	**21.0** ±6.1	21.2 ±5.5	28.2 ±5.6	**21.0** ±5.6	22.8 ±5.3
	10	25.1 ±5.5	24.8 ±7.2	29.5 ±.6.4	**24.0** ±5.6	25.1 ±6.4
	15	26.7 ±5.8	25.0 ±5.7	29.2 ±7.5	**24.1** ±6.2	26.5 ±6.5
Dermato-logy	*5*	3.5 ±1.5	5.6 ±2.6	4.5 ±2.1	**3.4** ±1.7	**3.4** ±1.7
	10	**3.4** ±2.0	6.2 ±2.9	5.9 ±2.9	**3.4** ±2.0	3.9 ±2.1
	15	4.0 ±2.3	6.4 ±2.8	5.9 ±2.5	**3.8** ±1.7	3.9 ±1.6
Wisconsin	*10*	**2.9** ±1.3	3.1 ±1.4	3.3 ±1.6	**2.9** ±1.3	**2.9** ±1.1
	20	3.0 ±1.4	3.4 ±1.7	3.2 ±1.7	**2.9** ±1.3	**2.9** ±1.3
	30	**3.1** ±1.6	3.6 ±1.6	3.4 ±1.5	3.2 ±1.5	3.2 ±1.7
Diabetes	*10*	**24.2** ±2.8	25.5 ±2.5	26.4 ±3.7	24.5 ±2.7	24.6 ±2.8
	20	**25.4** ±2.9	27.5 ±2.2	25.6 ±3.3	25.7 ±3.3	25.5 ±2.7
	30	**26.1** ±3.1	28.0 ±3.2	26.4 ±3.9	26.2 ±3.9	**26.1** ±3.5
Vehicle	*10*	29.1 ±3.2	29.6 ±2.7	31.1 ±3.3	**28.4** ±2.4	29.0 ±3.4
	15	**30.1** ±2.6	32.6 ±3.4	33.0 ±3.4	31.4 ±3.6	30.7 ±3.0
	20	32.4 ±2.8	36.2 ±3.8	33.8 ±3.1	32.5 ±2.7	**32.1** ±2.4
Yeast	*20*	41.8 ±2.1	43.5 ±2.1	42.5 ±2.3	42.1 ±2.2	**41.5** ±2.0
	30	42.0 ±2.4	44.3 ±2.8	43.4 ±2.4	**41.7** ±2.3	41.8 ±2.4
	40	43.1 ±2.9	45.5 ±3.0	44.5 ±2.5	**42.8** ±3.0	43.2 ±2.8
Spambase	*20*	8.5 ±0.8	8.5 ±0.9	8.4 ±0.8	**8.0** ±0.8	**8.0** ±0.8
	30	8.9 ±0.7	9.1 ±0.8	8.9 ±0.7	**8.4** ±0.6	8.5 ±0.7
	40	9.2 ±1.0	9.6 ±0.9	9.6 ±1.1	**8.5** ±1.0	8.6 ±1.0
Phoneme	*10*	16.6 ±0.8	16.5 ±0.8	16.6 ±1.0	16.6 ±0.8	**16.2** ±0.8
	20	17.2 ±1.1	17.8 ±1.1	17.3 ±1.0	**17.0** ±1.0	**17.0** ±1.1
	30	18.1 ±1.0	18.5 ±1.2	18.0 ±0.9	**17.6** ±1.0	**17.6** ±1.1
Page blocks	*20*	5.6 ±0.6	6.2 ±0.6	5.6 ±0.5	**5.3** ±0.5	5.5 ±0.5
	40	6.2 ±0.6	7.0 ±0.7	6.4 ±0.6	**5.7** ±0.5	6.1 ±0.5
	60	6.6 ±0.7	8.5 ±0.8	6.8 ±0.7	**5.9** ±0.7	6.6 ±0.8
Optical Digits	*20*	**2.1** ±0.4	2.5 ±0.5	2.5 ±0.5	**2.1** ±0.4	**2.1** ±0.4
	30	**2.5** ±0.5	3.0 ±0.6	2.9 ±0.6	**2.5** ±0.4	**2.5** ±0.6
	40	2.8 ±0.4	3.3 ±0.5	3.2 ±0.5	**2.7** ±0.4	**2.7** ±0.4
Satellite Image	*20*	**10.9** ±0.7	11.2 ±0.7	11.1 ±0.7	**10.9** ±0.7	**10.9** ±0.7
	30	**11.6** ±0.9	11.9 ±0.7	11.8 ±0.7	**11.6** ±0.7	**11.6** ±0.7
	40	12.5 ±0.7	12.3 ±0.9	12.3 ±0.8	12.2 ±0.7	**12.1** ±0.7
Ring	*20*	4.0 ±0.5	5.0 ±0.5	4.7 ±0.5	**3.9** ±0.5	**3.9** ±0.5
	40	6.8 ±0.6	8.9 ±0.7	8.8 ±0.7	**6.6** ±0.7	**6.6** ±0.7
	60	10.5 ±0.7	14.0 ±1.0	13.1 ±0.9	10.3 ±0.7	**10.2** ±0.8
Magic telescope	*20*	**13.8** ±0.5	14.0 ±0.5	14.0 ±0.5	**13.8** ±0.5	13.9 ±0.5
	40	14.2 ±0.4	14.3 ±0.5	14.4 ±0.4	14.2 ±0.4	**14.1** ±0.5
	60	14.6 ±0.5	14.8 ±0.5	14.7 ±0.4	14.5 ±0.4	**14.4** ±0.5
Letter	*20*	**9.9** ±0.5	11.7 ±0.5	11.6 ±0.5	**9.9** ±0.5	10.7 ±0.5
	40	**10.8** ±0.4	12.8 ±0.5	12.8 ±0.4	**10.8** ±0.4	11.8 ±0.5
	60	**13.3** ±0.7	15.0 ±0.7	14.8 ±0.7	**13.3** ±0.7	14.2 ±0.7
Shuttle	*20*	**3.1** ±0.2	3.3 ±0.2	3.2 ±0.2	**3.1** ±0.2	**3.1** ±0.2
	40	**3.6** ±0.3	3.9 ±0.3	4.0 ±0.2	**3.6** ±0.4	**3.6** ±0.2
	60	3.9 ±0.3	4.3 ±0.3	4.4 ±0.3	**3.7** ±0.3	3.9 ±0.2

As regards to the rankings of each method the 48 comparison cases in table 8.5 reveal that the proposed confidence ratio Affinity Propagation in the third column ranks 1st 20 times. Kappa pruning in the fourth column as well as JAM's pruning in the fifth

column ranks 1st 0 times. Reduce Error pruning in the sixth column ranks 1st 35 times and Orientation ordering in the seventh column ranks 1st 26 times.

Table 8.6 shows the pruning sizes after the ensemble selection phase by using confidence ratio Affinity propagation versus the other pruning methods. The pruning numbers may vary a lot. While the proposed method can automatically find the pruning number the other methods cannot. As we have already mention Kappa pruning and JAM's diversity-based pruning always select $L/3$ of classifiers, which is usually the case, while orientation ordering always selects half of them, and we pre-set Reduce-error pruning to select as many classifiers found by the proposed method. For datasets like Wine, Sonar, Dermatology, Diabetes, Vehicle, Yeast, Spambase and Page blocks the pruning methods select a similar number of classifiers which is around $L/3$. For other datasets like Phoneme, Optical Digits, Satellite Image and Ring the proposed method selects approximately $L/2$ of classifiers. Some datasets like Magic Telescope and Letter may require almost all the classifiers. For some datasets like Wisconsin or Shuttle the proposed method tends to select a much smaller number of classifiers.

Table 8.6. Pruning sizes after the ensemble selection phase by using confidence ratio Affinity propagation versus the other pruning methods as indicated.

Dataset	Initial ensemble *size*	Confidence ratio (eq.8.6) Affinity Propagate	Kappa Pruning	JAM's diversity-based pruning	Reduce-error pruning	Orientation Ordering
Wine	*5*	*2.1*	*2.0*	*2.0*	*2.0*	*2.5*
	10	*3.3*	*4.0*	*3.0*	*3.3*	*5.0*
	15	*4.1*	*6.0*	*5.0*	*4.1*	*8.0*
Sonar	*5*	*3.8*	*2.0*	*2.0*	*4.0*	*2.5*
	10	*5.2*	*4.0*	*4.0*	*5.2*	*5.1*
	15	*6.0*	*6.0*	*5.0*	*6.0*	*7.4*
Dermato-logy	*5*	*3.0*	*2.9*	*2.0*	*3.0*	*2.7*
	10	*4.4*	*4.0*	*3.0*	*4.4*	*5.3*
	15	*5.6*	*6.0*	*5.0*	*5.6*	*7.9*
Wisconsin	*10*	*3.8*	*4.0*	*3.0*	*3.8*	*4.8*
	20	*5.0*	*6.0*	*6.0*	*5.0*	*9.0*
	30	*6.3*	*10.0*	*10.0*	*6.3*	*14.2*
Diabetes	*10*	*4.1*	*4.0*	*4.0*	*4.1*	*5.0*
	20	*6.5*	*6.0*	*6.0*	*6.5*	*9.5*
	30	*9.5*	*9.1*	*9.1*	*9.5*	*12.5*
Vehicle	*10*	*5.5*	*4.0*	*3.0*	*5.5*	*5.0*
	15	*7.7*	*6.0*	*6.0*	*7.7*	*7.7*
	20	*9.3*	*6.0*	*6.0*	*9.3*	*10.4*
Yeast	*20*	*6.7*	*6.0*	*6.0*	*6.7*	*10.8*
	30	*11.4*	*10.0*	*10.0*	*11.3*	*16.3*
	40	*14.1*	*14.0*	*13.0*	*14.1*	*21.5*
Spambase	*20*	*6.3*	*6.0*	*6.0*	*6.3*	*10.1*
	30	*11.2*	*10.0*	*10.0*	*11.2*	*15.1*
	40	*13.2*	*14.0*	*14.0*	*13.2*	*19.2*
Phoneme	*10*	*1.7*	*4.0*	*3.0*	*1.7*	*5.0*
	20	*9.8*	*6.0*	*6.0*	*9.8*	*9.7*
	30	*15.4*	*10.0*	*10.0*	*15.5*	*15.0*
Page	*20*	*4.7*	*6.0*	*6.0*	*4.7*	*10.2*

blocks	*40*	*12.5*	*14.0*	*13.0*	*12.5*	*20.0*
	60	*18.9*	*20.0*	*20.0*	*18.9*	*29.0*
Optical Digits	*20*	*12.8*	*6.0*	*6.0*	*12.8*	*10.2*
	30	*16.7*	*10.0*	*10.0*	*16.7*	*15.0*
	40	*18.1*	*14.0*	*14.0*	*18.1*	*20.1*
Satellite Image	*20*	*15.3*	*6.0*	*6.0*	*15.3*	*10.0*
	30	*21.5*	*10.0*	*10.0*	*21.5*	*15.0*
	40	*24.5*	*14.0*	*14.0*	*24.5*	*20.1*
Ring	*20*	*6.1*	*6.0*	*6.0*	*6.1*	*10.3*
	40	*18.6*	*14.0*	*13.0*	*18.6*	*20.4*
	60	*29.6*	*20.0*	*20.0*	*29.6*	*30.4*
Magic telescope	*20*	*14.5*	*6.0*	*6.0*	*14.5*	*9.9*
	40	*35.1*	*14.0*	*14.0*	*35.1*	*20.1*
	60	*46.5*	*20.0*	*20.0*	*46.5*	*30.6*
Letter	*20*	*17.0*	*7.0*	*7.0*	*17.0*	*10.1*
	40	*38.0*	*14.0*	*14.0*	*38.0*	*19.6*
	60	*58.0*	*20.0*	*20.0*	*58.0*	*31.0*
Shuttle	*20*	*1.3*	*7.0*	*7.0*	*1.3*	*9.5*
	40	*2.0*	*14.0*	*13.0*	*2.0*	*19.5*
	60	*4.4*	*20.0*	*20.0*	*4.4*	*25.5*

Generally the comparisons with the other well known pruning methods reveal that the proposed method is among the first and outperforms kappa pruning and JAM's pruning at a high level of confidence. In addition, it is comparable with Reduce Error pruning and Orientation Ordering where there is no statistical significance between their ranking results and they behave similarly. However the proposed confidence ratio is local in nature since it defines the pair-wise diversity of two classifiers based on their local data only. Furthermore when combined with Affinity Propagation it does not need the pruning number to be given in advance, since the AP algorithm outputs this number. These are two advantages. In contrast all the other pruning methods are global in nature, since they define the diversity for every pair of classifiers based on the global dataset each time, and they also need to select a fixed predefined number of classifiers which they require as input. Since they don't know how many classifiers to prune they need this number as an extra parameter. Hence considering that the proposed method is fast and preserves privacy while it produces similar results with the best reduce-error pruning greedy search, renders the method a promising candidate for distributed privacy-preserving ensemble selection.

8.7 Discussion

Heterogeneous disjoint data partitions are those that don't share the same classes. In that case any pair of classifiers may have many classes different. Some classes are missing in every location and all classifiers are unique. Then two classifiers are unlikely to be similar enough so as to prune one of them. For example the Letter benchmark dataset has 26 classes of letters in which the examples are equally distributed. When this dataset is partitioned in several disjoint groups then a highly diverse population emerges. In this case every classifier is more accurate for its own data than any other. A pruning algorithm should recognize when all classifiers are necessary. The proposed method automatically selects almost all of them. Usually in pruning methods such as those in the comparisons, that need in advance the pruning

number as input, there must be another meta-learning step to determine the best pruning number by monitoring the error of the pruned ensemble using an extra validation set. This meta-learning step is an iterative process that repeatedly tests several different pruning numbers and keeps the one with minimum error on the validation set. This step is not required in confidence ratio affinity propagation.

One disadvantage of the proposed method is that it doesn't take into account any per class information. For instance Kappa pruning uses the number of classifications per class. Orientation ordering uses the complete signature vectors, which hold the individual classifications in detail, and compares it with the average signature vector that contains the class proportions. The confidence ratio we employ doesn't use any class-related information. A class-conscious version of the proposed method that finds the best classifiers for each class could possibly be investigated in the future.

During analysing the error rates in the comparisons one must bear in mind that the ensemble is created from disjoint data partitions and not bags like bagging. Thus, the errors should be slightly higher than those usually produced from a bagging ensemble which on the other hand is not privacy-preserving. However the proposed confidence ratio affinity propagation is an embarrassingly parallel, asynchronous and distributed process which can operate, without restrictions, as a candidate for pruning bagging ensembles as well. Preliminary results (not included here) for bagging ensembles were most promising.

8.8 Summary

We consider constructing an ensemble of neural network classifiers and selecting the best of them for the special problem of distributed privacy-preserving data mining from large decentralized data locations. We cover the case of using Affinity Propagation (AP) as an ensemble selection method, since this state-of-the-art algorithm had not been exploited so far in the literature for pruning classifier ensembles. A new similarity measure, the confidence ratio, is proposed for the locally defined pair-wise similarities the AP algorithm needs. The proposed method performs confidence ratio Affinity Propagation among the classifiers and prunes them without predefining the pruning number or monitoring the pruned ensemble performance on a separate validation set.

All the necessary pair-wise confidence ratio similarities between the classifiers are produced in an asynchronous distributed and privacy-preserving computing cycle. At the end of the ensemble selection phase no participant location will know anything else except its own private data and the result. We proposed the confidence ratio Affinity Propagation that is local in nature since the confidence ratio is a locally defined pair-wise measure between two different classifiers (eq.8.6) and Affinity Propagation is based on local pair-wise message passing. Given two classifiers $\{f_i,f_j\}$ the confidence ratio similarity $\mathbf{S}(i,j)$ between them depends only on their two local training datasets, N_i and N_j. We compare the proposed method against five Affinity Propagation variants that use respectively five well known globally defined pair-wise diversity measures which need a global, common to all, dataset to find the pair-wise distances among the classifiers. Experimental comparisons with the five globally defined pair-wise diversity measures show that the confidence ratio outperforms them. Considering that the computational cost of the confidence ratio similarity is $O(L{\cdot}N)$ adds one extra credit in the proposed distributed privacy-preserving strategy. The experimental comparisons with four well-known globally defined pruning methods shows that the proposed

confidence ratio affinity propagation outperforms two of them and delivers comparable error rates with the other two. As far as the pruning number is concerned, the comparisons with the other pruning methods show how the proposed confidence ratio affinity propagation selects a variable number of classifiers that is different for each dataset case. Thus another advantage is that while the other methods need to pre-define a fixed pruning number, in affinity propagation the pruning number emerges automatically during the message passing process. Based on the point-to-point local nature of the proposed method we plan to study it as a candidate for conventional ensemble pruning in bagging ensembles and possibly extend it as an on-line algorithm for peer-to-peer systems.

References of chapter 8

[1] Kargupta H. and Sivakumar K. (2004) Existential Pleasures of Distributed Data Mining. Data Mining: Next Generation Challenges and Future Directions. AAAI/MIT press, Cambridge.

[2] Talia D. (2008) Distributed data mining tasks and patterns as services. In Proceedings of the DPA Workshop 2008, Las Palmas.

[3] Aggarwal C.C. and Yu P.S. (2008) Privacy-Preserving Data Mining: Models and Algorithms. Kluwer Academic Publishers, Boston.

[4] Clifton C., Kantarcioglu M., Vaidya J., Lin X., Zhu M. (2003) Tools for Privacy Preserving Distributed Data Mining. ACM SIGKDD Explorations 4(2): 1–7.

[5] Breiman L., (1999) Pasting Small Votes for Classification in Large Databases and On-Line. Machine Learning 36: 85–103.

[6] Chawla N.V., Hall L.O., Bowyer K.W. and Kegelmeyer W.P. (2004) Learning ensembles from bites: A scalable and accurate approach. Journal of Machine Learning Research 5, 421-451.

[7] Tsoumakas G., Angelis L. and Vlahavas I. (2004) Clustering Classifiers for Knowledge Discovery from Physically Distributed Databases. Data and Knowledge Engineering, 49(3), 223-242.

[8] Brazdil P., Giraud-Carrier C., Soares C. and Vilalta R. (2009) Metalearning: Applications to Data Mining. Springer, Berlin.

[9] Chan P. and Stolfo S. (1993) Meta-learning for multistrategy and parallel learning. In Proceedings of the Second International Workshop on Multistrategy Learning.

[10] Prodromidis A. and Stolfo S. (1998) Pruning meta-classifiers in a distributed data mining system. In Proceedings of the First National Conference on New Information Technologies, pages 151–160, Athens, Greece, 1998.

[11] Prodromidis A., Chan P. and Stolfo S. (2000) Meta-learning in distributed data mining systems: Issues and approaches. In (Kargupta H., Chan P., eds.), Advances of Distributed Data Mining, AAAI/MIT Press, Cambridge, MA, 2000.

[12] Wang L. and Fu X. (2005) Data Mining with Computational Intelligence. Springer-Verlag. Berlin.

[13] Seni G. and Elder J. (2010) Ensemble Methods in Data Mining. Morgan & Claypool publishers.

[14] Re M. and Valentini G. (2012) Ensemble methods: a review. In Advances in Machine Learning and Data Mining for Astronomy, Chapman & Hall/CRC Data Mining and Knowledge Discovery Series, CRC press, Boca Raton, 2012, pp. 563–594.

[15] Zhou Z.-H., Wu J., Tang W. (2002) Ensembling neural networks: Many could be better than all. Artificial Intelligence 137, 239-263.

[16] Poggio T. and Girosi F. (1990) Regularization algorithms for learning that are equivalent to multilayer networks. Science 247, 978–982.

[17] Girosi F., Jones M. and Poggio T. (1995) Regularization theory and neural networks architectures. Neural Computation 7, 219–269.

[18] Evgeniou T., Pontil M., Poggio T. (2000) Regularization Networks and Support Vector Machines. Advances in Computational Mathematics 13(1), 1–50.

[19] Frey B.J. and Dueck D. (2007) Clustering by passing messages between data points. Science 315, 972–976.

[20] Kokkinos Y. and Margaritis K.G. (2012) A distributed asynchronous and privacy preserving neural network ensemble selection approach for peer-to-peer data mining. In ACM Proceedings of the 5th Balkan Conference in Informatics, BCI 2012, pp. 46-51.

[21] Kantarcioglu M. and Vaidya J. (2003) Privacy-Preserving Naive Bayes Classifier for Horizontally Partitioned Data. In proceedings of IEEE Workshop on Privacy-Preserving Data Mining.

[22] Yi X. and Zhang Y. (2009) Privacy-preserving naïve Bayes classification on distributed data via semi-trusted mixers. Information Systems 34, 371–380.

[23] Yu H., Jiang X. and Vaidya J. (2006) Privacy-Preserving SVM using nonlinear Kernels on Horizontally Partitioned Data. In Proceedings of the SAC Conference, 2006.

[24] Xiong L., Chitti S. and Liu L. (2006) k nearest neighbour classification across multiple private databases. In Proceedings of the ACM Fifteenth Conference on Information and Knowledge Management, 5-11 November 2006.

[25] Margineantu D. and Dietterich T. (1997) Pruning adaptive boosting. In proceedings of the 14th International Conference on Machine Learning, pp. 211-218.

[26] Sharkey A., Sharkey N., Gerecke U., Chandroth G. (2000) The test and select approach to ensemble combination. Multiple Classifier Systems, 30–44.

[27] Partalas I., Tsoumakas G. and Vlahavas I. (2009) Pruning an Ensemble of Classifiers via Reinforcement Learning. Neurocomputing 72, 1900–1909.

[28] Tsoumakas G., Partalas I., Vlahavas I. (2009) An Ensemble Pruning Primer. In O. Okun, G. Valentini (Eds.), Applications of Supervised and Unsupervised Ensemble Methods, 245, Springer, Berlin, 2009, pp. 1–13.

[29] Tamon C., Xiang J. (2000) On the boosting pruning problem. In Proceedings of 11th European Conference on Machine Learning, Springer, Berlin, pp. 404-412.

[30] Xie Z., Xu Y., Hu Q., and Zhu P. (2012) Margin distribution based bagging pruning. Neurocomputing 85, 11–19.

[31] Martínez-Muñoz G. and Suárez A. (2006) Pruning in ordered bagging ensembles. In Proceedings of the 23rd International Conference on Machine Learning, (ICML-2006), pp. 609–616.

[32] Martínez-Muñoz G., Hernández-Lobato D. and Suárez A. (2009) An analysis of ensemble pruning techniques based on ordered aggregation. IEEE Transactions on Pattern Analysis and Machine Intelligence 31(2), 245-259.

[33] Giacinto G., Roli F. and Fumera G. (2000) Design of effective multiple classifier systems by clustering of classifiers. In Proceedings of 15th International Conference on Pattern Recognition, 2000, pp. 160-163,

[34] Giacinto G. and Roli F. (2001) An approach to the automatic design of multiple classifier systems. Pattern Recognition Letters 22, 25-33.

[35] Liu R. and Yuan B. (2001) Multiple classifier combination by clustering and selection. Information Fusion 2, 163-168.

[36] Bakker B. and Heskes T. (2003) Clustering ensembles of neural network models. Neural Networks 16, 261-269.

[37] Lazarevic A. and Obradovic Z. (2001) Effective pruning of neural network classifiers. In Proceedings of the IEEE/INNS International Conference on Neural Networks, (2001) pp. 796–801.

[38] Fu Q., Hu S.-X. and Zhao S.-Y. (2005) Clustering based selective neural network ensemble. Journal of Zhejiang University 6A (5), 387–392.

[39] Chen H., Yuan S., Jiang K. (2006) Selective Neural Network Ensemble Based on Clustering. In Wang J. et al. (eds.): Advances in Neural Networks, 3971, LNCS, Springer, Berlin, 2006, pp. 545– 550.

[40] Yu H-L., Chen G.-F., Liu D.-Y., Wan B.-C. and Jin D. (2009) A Novel Neural Network Ensemble Method Based on Affinity Propagation Clustering and Lagrange Multiplier. In Proceedings of the International Conference on Computational Intelligence and Software Engineering, 2009.

[41] Kuncheva L.I. and Whitaker C.J. (2003) Measures of Diversity in Classifier Ensembles and Their Relationship with the Ensemble Accuracy. Machine Learning 51, 181–207.

[42] Tang E.K., Suganthan P.N., Yao X. (2006) An analysis of diversity measures. Machine Learning 65, 247–271.

[43] Kai Li, Hou-Kuan Huang, Xiu-Chen Ye, Li-Juan Cut (2004) A selective approach to neural network ensemble based on clustering technology. In Proceedings of the Third International Conference on Machine Learning and Cybernetics, Shanghai, 26-29.

[44] Yu L., Lai K.K., Wang S. (2008) Multistage RBF neural network ensemble learning for exchange rates forecasting. Neurocomputing 71, 3295– 3302.

[45] Liu Y. and Yao X. (1999) Simultaneous training of negatively correlated neural networks in an ensemble. IEEE Transactions on Systems, Man, and Cybernetics, Part B: Cybernetics 29(6), 716–725.

[46] Chen H. and Yao X. (2009) Regularized negative correlation learning for neural network ensembles. IEEE Transactions on Neural Networks 20, 1962–1979.

[47] Wozniak M., Graña M. and Corchado E. (2014) A survey of multiple classifier systems as hybrid systems, Information Fusion 16, 3–17.

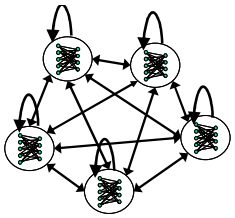
[48] Lysiak R., Kurzynski M. and Woloszynski T. (2014) Optimal selection of ensemble classifiers using measures of competence and diversity of base classifiers. Neurocomputing 126, 29–35.

[49] García S., Fernández A., Luengo J., Herrera F. (2010) Advanced nonparametric tests for multiple comparisons in the design of experiments in computational intelligence and data mining: Experimental analysis of power. Information Science 180, 2044–2064.

[50] Kokkinos Y. and Margaritis K.G. (2014). Breaking ties of plurality voting in ensembles of distributed neural network classifiers using soft max accumulations. In 10th International Conference on Artificial Intelligence Applications and Innovations, (AIAI 2014), Springer/ Lecture Notes in Computer Science, IFIP AICT 436, pp. 20–28.

9 Distributed privacy-preserving P2P data mining via Probabilistic Neural Network Committee Machines

This chapter describes a probabilistic neural network (PNN) committee machine for Peer-to-Peer data mining. The pattern neurons of the PNN committee are composed of locally trained class-specialized regularization network Peer classifiers. The training of the PNN committee takes into account the asynchronous distributed and privacy-preserving requirements that can be met in P2P systems.



The distributed classifiers are first trained in parallel based on their local data. While no local data exchange is possible among them, the peers can exchange their classifiers in the form of binaries, or agents. Then an asynchronous distributed computing P2P cycle is executed to construct a mutual validation matrix. The training set of one Peer becomes the validation set of the other and only average rates are returned back. From this matrix we demonstrate that based on regularized non-negative weighting, it is possible to perform weight based ensemble selection of best peer members for every class and in this way to find class-specialized Peer modules for the committee machine.

9.1 Introduction

A large scale Peer-to-Peer (P2P) system is composed of distributed loosely coupled Peer processors from different participating locations. Typical data mining tasks [1] gather data from several locations into a central site and run algorithms that extract globally interesting models and useful aggregate statistics. Distributed privacy-preserving data mining [2][3][4] is the study of how to make this without moving data and without disclosing private information within the different participants. Data exchanges as well as free information flow are usually being hindered by technical issues, legal responsibilities or personal concerns.

Peer-to-Peer systems usually have a large number of locations to hold small volumes of data. Thus several such isolated small private data collections are ideal to form a committee machine [5] of Neural Networks [6]. The whole system is well-known for its inherently parallel and distributed architecture [7]. By combining many neural network models together the committee has excellent generalization capabilities, avoids over-fitting, and is much more reliable that a stand-alone neural network.

Thus a P2P system is an ideal candidate for a committee machine composed of loosely coupled individual peer local classifiers. In theory for training such a committee machine of loosely coupled individual peer local classifiers all Peers must be synchronized, at least two by two, in order to find the combining weights and all peers must share a portion of their data to form a common validation set. However, in practice, large scale synchronization points are unfeasible and no peer can willingly share its personal data. This is the main problem we address here.

We assume that each individual Peer can train a single Regularization Network (RN) [8][9][10] to classify its own local data. Once all these Peer classifiers are assembled they can be called independently, in parallel, to classify an unknown instance and their results must be fused with a combination rule [11] like majority voting, averaging, weighted voting, a meta-classifier, or a committee. However a trainable combiner like the last three requires a separate validation set in order to find proper weights for the neural network modules which without data sharing is challenging.

In P2P distributed systems desirable factors for data mining algorithms are scalability, efficiency in communication, asynchronous, distributed and privacy-preserving. The privacy-preserving factor is tricky and is generally based on secure multi-part computations. The main idea of the secure multi-party computation is that a distributed computation is secure if at the end no party knows anything else except its own input data and the final results. Related work on classifier examples that have been extended to the distributed privacy-preserving data mining problem are the Naive Bayes classifier [12], the SVM classifier with nonlinear kernels [13] and the k-nearest neighbour classifier [14]. Requirements that have to do with scalable, asynchronous and communication efficient computations can be met if one transforms the learning algorithm to only lock-free point-to-point communications and computations like those for a simple mutual validation matrix proposed in a previous work [15] and the PNN committee training in this work.

We demonstrate that the Peers can be combined via a Probabilistic Neural Network [16][17] committee machine where its pattern neurons have been replaced by class specialized Peer RN local classifiers. The choice of PNN [16][17] as a committee was made in light of its ability to work with non-parametric unknown data distributions. In an asynchronous distributed and privacy-preserving way the proposed method selects the best neural network parts specialized for each class. Thus while a previous work [15] deals with unspecialized Peers this work addresses for the first time the problem of class conscious Peer specialization and selection. The resulting scheme is proven very effective.

9.2 Probabilistic Neural Network Committee Machine of Class Specialized Peers

Like the Probabilistic Neural Network (PNN) [16][17], the PNN committee machine in fig. 9.1 has four layers, namely input, pattern, summation and output. In place of the PNN pattern layer neurons the class-specialized Regularization Network Peers are now situated. The choice of Probabilistic Neural Network (PNN) [16][17] as a Committee Machine is made in light of its ability to work with non-parametric unknown data distributions, much like the distributions in Peers. The PNN Committee Machine is composed of class specialized Regularization Network Peers. A boosting or bagging ensemble of independently trained PNNs is also possible for classification tasks [18] [19] with non privacy preserving restrictions.

From Bayes theorem which minimizes the classification error, the posterior probability a given unknown pattern $\mathbf{x}$ belongs to class A is computed as $P(A \mid \mathbf{x}) = P(\mathbf{x} \mid A)P(A)/P(\mathbf{x})$. The normalization factor $P(\mathbf{x})$ in the denominator is the sum of the numerators from all classes (same for all posteriors). The conventional PNN [16][17][18][19] is a combination of the Bayes theorem and the non parametric estimation of the probability density functions via Parzen window kernels located in the pattern layer. In our case the Peer neural network outputs play the role of these pattern layer outputs.

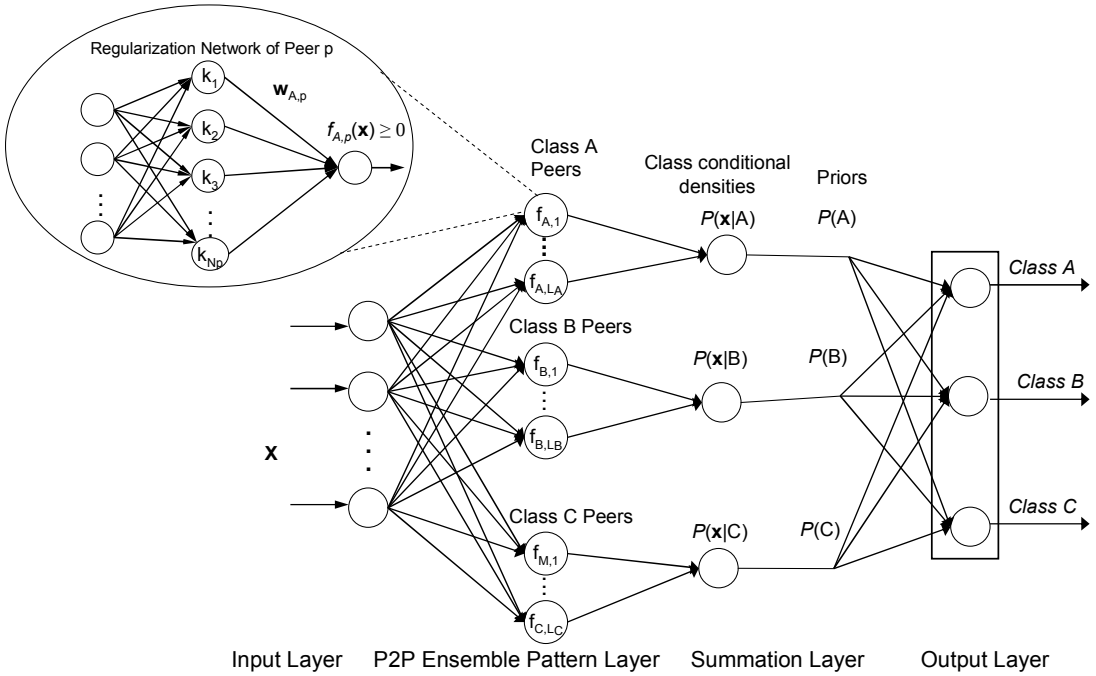


Figure 9.1. A three-class Probabilistic Neural Network Committee Machine architecture with Bayessian probability terms, and class specialized Regularization Networks in place of the usual pattern neurons

In the three-class example of fig. 9.1 the P2P pattern layer is where the Peer Regularization Networks are loaded and divided in groups, one for each class. Class A has L_A Peers as pattern neurons and a summation neuron $P(\mathbf{x} \mid A)$ in the summation layer. Similarly class B has L_B Peers and class C has L_C Peers. For an unknown sample $\mathbf{x}$ the conditional probability for, say, class A computes a sum of functions $f_{A,p}(\mathbf{x})$ of the form:

$$P(\mathbf{x} \mid \mathrm{A}) = \frac{1}{L_A} \sum_{p=1}^{L_A} f_{A,p}(\mathbf{x}) \tag{9.1}$$

where $f_{A,p}(\,)$ is the pattern 'neuron', namely the RN output of Peer p for class A. The summation layer neurons compute all the class conditional densities $P(\mathbf{x} \mid m)$. Finally the output layer classifies the unknown $\mathbf{x}$ in the class m that have maximum $P(\mathbf{x} \mid m) \cdot P(m)$. The conditional probability for class m, provides also the confidence levels of this class. Before the committee training all peers participate in all classes and $L_A=L_B=L_C=L$. Thus, the proposed committee machine training intents to find the best peers for each class. Hence which peers to use for each class is the main problem we solve via the supervised weight-based selection through the mutual validation matrix we propose next. The method solves the non-negative weighting problem of each peer through the mutual validation matrix, and keeps only the regularization network peers with non-zero weights in every class.

9.3 PNN Committee Machine Training Steps

The first step is to train every peer regularization network (RN) using its local data. Then the P2P PNN committee machine via distributed privacy-preserving training must find the best RN parts for a particular class. Such a Peer selection becomes feasible by solving a linear system of equations with non-negative weight constraints, using a P2P mutual validation matrix. The class-specialized regularization network parts of every peer p that end up with combining weights above zero ($g_{m,p}>0$) are finally selected for each class m. This is called weight-based selection. In this fashion specialized Regularization Network Peers can be formed. A Peer can participate in more than one class groups. Then the P2P PNN committee combines the selected Peers by loading them into the hidden pattern layer where the Peers are act as typical neurons.

The proposed scheme for asynchronous distributed and privacy-preserving training of the P2P PNN committee machine is summarized in the following steps:

1. Each distributed Peer constructs a local classifier model, a Regularization Neural Network (RN), which is trained on its local data patterns.
2. The P2P mutual validation matrix is computed in an asynchronous distributed and privacy preserving fashion. Each Peer sends the classifier as a black box, in the form of binary or agent, to all other Peers. Each Peer tests all the received classifiers on its local data. The Peers compute the average positive hits for the correctly classified samples, and send back the resulting sums. Only these sums are centralized to construct the asymmetric mutual validation matrix, for the pair wise entries of all the distributed RNs.
3. The supervised Peer selection via non negative weights algorithm finds the best local RN Peer parts for each class of the PNN committee pattern layer.

The following paragraphs present analytical details.

9.3.1 Training the Regularization Network Peers

A Regularization Network (RN) [8] [9][10] classifier for the two class problem (class A versus the rest) is illustrated in fig. 9.1. The input neurons in each RN are as many as the number of data features. The hidden neurons are as many as the number of the local training instances. This hidden layer is where the real data samples are loaded to form the N_p kernel centers. For a given local training set $\{\mathbf{x}_n, y_n\}_{n=1}^{Np}$, a kernel function $k(\cdot,\cdot)$ and a kernel matrix $\mathbf{K}$, the task is to find an optimum nonlinear mapping function with linear weights w_i, such that the RN output for an unknown $\mathbf{x}$ is given by $f_p(\mathbf{x}) = \sum_{n=1}^{N_p} w_n \cdot k(\mathbf{x}_n, \mathbf{x})$. The kernel function can be any symmetric, positive semi-definite function, and the most commonly used is the Gaussian kernel given by $k(\mathbf{x}_n,\mathbf{x})=\exp(-\|\mathbf{x}_n-\mathbf{x}\|^2/\sigma^2)$.

The learning problem in RN [8][9][10] is stated as the minimization on Reproducing Kernel Hilbert space (RKHS) of a regularized risk functional (eq. 9.2) that has the usual data error term plus a second regularization term as stabilizer:

$$\arg\min_{f\in H_K}\left\{ \frac{1}{N_p}\sum_{n=1}^{Np} (y_n - f_p(\mathbf{x}_n))^2 + \gamma \left\| f_p \right\|_K^2 \right\} \tag{9.2}$$

The data term in eq. 9.2 is proportional to the number N_p of data points, and $\gamma > 0$ is the regularization parameter. The solution is unique [8][9][10] and for the three class problem in fig. 9.1 there are three parts of such RN and three weight vectors one for each class A, B, C, and their weights $[\mathbf{w}_A\ \mathbf{w}_B\ \mathbf{w}_C]^T$ are given by solving the linear systems:

$$\mathbf{w}_A = (\mathbf{K} + N_p\, \gamma\, \mathbf{I})^{-1}\, \mathbf{y}_A \tag{9.3a}$$

$$\mathbf{w}_B = (\mathbf{K} + N_p\, \gamma\, \mathbf{I})^{-1}\, \mathbf{y}_B \tag{9.3b}$$

$$\mathbf{w}_C = (\mathbf{K} + N_p\, \gamma\, \mathbf{I})^{-1}\, \mathbf{y}_C \tag{9.3c}$$

where $\mathbf{K}$ is the kernel matrix, $\mathbf{I}$ the identity matrix, $\mathbf{y}_A$, $\mathbf{y}_B$ and $\mathbf{y}_C$ are the vectors (of size N_p) that hold the desired labels ($y_1, y_2, \ldots, y_{Np}$) that use hot encoding. In the case where a Peer does not hold any training data from a class, the weights for this class are zero.

9.3.2 Distributed Mutual validation matrix

A coarse-grained fully distributed mutual validation matrix is employed in order to non-negative weighting each Peer for each class and to find which Peers to prune. The procedure must be done without reveal the local training data vectors between Peer modules. In the P2P system every peer has a network address and can send messages to other peers. The messages can have arbitrary delays and can be lost as well. Therefore only asynchronous type computations are feasible. A distributed asynchronous mutual validation matrix we have introduced in a previous work [15] is employed here. However, while our previous work [15] deals with unspecialized RNs, this work addresses the problem of class-specialized peers in order to find which Peers to prune for each class.

The simple mutual validation matrix **V** has size *LxL*, and maps all *L* peers in a pair-wise fashion. The set of *L* peer Regularization Network classifiers are ready to contribute in building the committee. However for privacy reasons the peers cannot share with each other any even smallest part of their datasets. Only classifiers can be sent, as binaries or agents, to other peers to validate their local datasets. The average learning rates are sent back to fill in the mutual validation matrix **V**. So the train set from one Peer becomes the validation set of the other and vice versa.

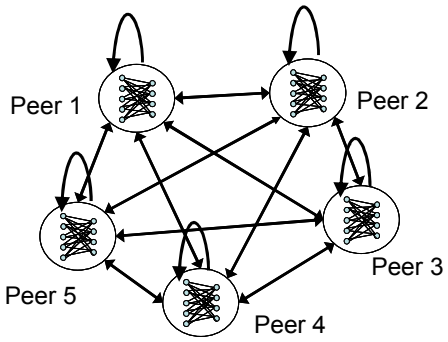


Figure 2. A graphical representation shows all forms of the communications in the P2P system where neural network Peers inter-connected with each other via accuracy measures

The rows of the matrix **V** correspond to local data and the columns of **V** to the travelling RN classifiers. Upon receiving a RN(*j*) the Peer *p* applies it to classify its own local data and sends back a simple average rate equal to:

V(p,j) = (positive hits of Regularization Network *j* when classifying samples of *p* peer) / (local train size of peer *p*)

where positive hits are the number of correctly classified local samples of *p* and local train size is their N_p number. In this way privacy-preserving is achieved. As only average learning rates are been send back no peer reveals any of its local data. The training phase is secure since at the end of the computation, no party knows anything except its own input and the results.

9.3.3 Supervised Peer selection using non-negative weighting

We propose a supervised ensemble selection by solving a non-negative regularized regression problem for finding non negative weight values and thus keeping only the peers with non-zero weights. Pruning classifiers or neurons with zero weights is a well known strategy in the Neural Network literature. The main issue here is which system to solve for finding these weights in a privacy-preserving way.

We follow the steps of our earlier work [15] in order to define the regression problem in terms of the previously found mutual validation matrix **V**. Then we add a non-negative constraint and compute the combining weights via non-negative regularized least squares. Finally every peer *p* with class weight zero is excluded. Thus for the three-class example in fig. 9.1 a peer *p* with $\mathbf{g}_{A,p}=0$ is pruned from class A. Similarly the peers with zero weights $\mathbf{g}_{B,p}$ in class B, and zero weights $\mathbf{g}_{C,p}$ in class C are pruned, thus forming a set of specialized peers for every class. Then the PNN committee uses only these selected peers.

The $\mathbf{g}_A$, $\mathbf{g}_B$ and $\mathbf{g}_C$ peer combining weight vectors, one for each class, can be found, in terms of the vectors $\mathbf{u}_A$, $\mathbf{u}_B$ and $\mathbf{u}_C$ which are used to hold the desired averaged label outputs.

$$(\mathbf{V} + \lambda \mathbf{I})\, \mathbf{g}_A = \mathbf{u}_A \quad , \quad \mathbf{g}_A \geq 0 \tag{9.4a}$$

$$(\mathbf{V} + \lambda \mathbf{I})\, \mathbf{g}_B = \mathbf{u}_B \quad , \quad \mathbf{g}_B \geq 0 \tag{9.4b}$$

$$(\mathbf{V} + \lambda \mathbf{I})\, \mathbf{g}_C = \mathbf{u}_C \quad , \quad \mathbf{g}_C \geq 0 \tag{9.4c}$$

where $\mathbf{I}$ is the identity matrix and λ a regularization parameter. The vectors $\mathbf{u}_A$, $\mathbf{u}_B$, and $\mathbf{u}_C$ of the desired labels that correspond to classes have size equal to the population L of the RN Peers. A value $\mathbf{u}_A(p)$ is the number of positive local hits for A class portion of Peer classifier p per total train size, that must be produced by applying it to its own p^{th} dataset. All other desired labels are similarly defined. A simple non-negative quadratic optimization solver is used for computing the weights. We have found that by progressively lowering the λ parameter fewer p Peers end up with non zero weight values $\mathbf{g}_{A,p}$, $\mathbf{g}_{B,p}$, $\mathbf{g}_{C,p}$ and in consequence fewer Peers can be selected, thus permitting a tuning method for their final number k. By lowering λ the optimum k Peer population is chosen by simple search for turning points, or the knee, in the (k,λ) chart.

9.4 Experiments

Experiments are performed on publicly available real-world benchmark datasets from the UCI repository (http://archive.ics.uci.edu/ml). Each dataset is divided into a train set and a test set. We measure the error ratio (falsely classified samples)/(total) on the test set. A comparison with majority voting is also made.

The experimental design has six steps:

(1) A dataset is randomly split into a train set (70%) and a test set (30%) with stratification.

(2) The train set is distributed unevenly, randomly and without stratification across the Peers.

(3) Every Peer trains a local Regularization Network classifier.

(4) An asynchronous distributed computing cycle among Peers is executed to find the mutual validation matrix.

(5) The specialized Peers are selected using the proposed weighted-based selection.

(6) The final PNN committee machine is tested on the test set. This six step procedure is repeated 10 times for each benchmark dataset and each Peer number, and the results are averaged.

In step 2 to achieve the uneven, random and without stratification distribution of data across Peers we allow a quarter of processors to randomly choose their population size between 5 and 300 samples. Similarly another quarter of Peers choose a size between 5 and 100 and the remainder half of Peers choose a size between 5 and 20. After that these sizes are rescaled based on the total number of training samples. Following these divisions the step 2 creates fairly uneven un-stratified distributions of data.

Table 9.1 summarizes the results. From the Iris dataset it can be clearly seen that the proposed method manages to select fewer class specialized peers. Using only the selected Peers the PNN committee outperforms majority voting and in the same time increases its speed and accuracy. All benchmark datasets deliver similar results and substantial accuracy improvements while using much fewer peers than the initial population.

Table 9.1. Comparison results

	Initial RN Peers	Specialized Peers selected	Majority Voting Error	PNN Committee Machine Error
Iris	10	3.3	4.0	**3.7**
	12	3.4	5.1	**4.2**
	14	3.9	4.3	**3.3**
	16	4.0	5.6	**3.6**
Wine	10	3.4	3.5	**2.7**
	12	4.2	3.2	**2.4**
	14	4.6	4.2	**3.3**
	16	4.9	4.0	**2.9**
Diabetes	50	8.0	28.3	**22.8**
	60	8.4	28.8	**23.3**
	70	9.6	31.1	**24.1**
	80	10.9	31.9	**24.4**
Glass	10	5.2	39.0	**33.7**
	12	6.0	37.5	**33.0**
	14	7.1	38.8	**33.4**
	16	7.6	39.2	**33.9**
Spam	20	6.2	9.0	**7.5**
	40	7.8	10.5	**7.7**
	60	9.0	11.2	**8.3**
	100	12.0	12.7	**8.4**

The Wine dataset results show that the committee error is smaller than majority voting error and once more the method selects much fewer specialized peers that the total population. From the Diabetes dataset one can see that the PNN committee improves the accuracy of majority voting by a rather large amount. The Glass dataset results reveal that the proposed method selects few specialized peers and the improved accuracy delivered by the PNN committee is substantial. The Spam Emails dataset results show once more that the improvement achieved by the PNN committee is significant. All the groups intend to show that the number of the selected peers is sufficient to guarantee a very accurate committee. We vary the number of initial peers to study the range of the proposed method applicability.

9.5 Summary

For large scale distributed Peer-to-Peer data mining the present study deals with the challenging case to train a committee machine asynchronously and privacy-preserving with no local data exchange among the Neural Network Peer classifiers. To this task we employ the simple idea of the mutual validation matrix which studied in an earlier work, and we further present two distinct concepts that deserve further notice. The first concept is the Probabilistic Neural Network that is proposed as the combiner committee machine of the module outputs, which is not so common strategy in typical committees, since their modules are not class specialized. The second concept is a new Neural Network ensemble selection technique based on regularized non-negative weighting in order to sampling the class specialized Peers. Based on experimental results the whole scheme manages to use fewer Peers thus increasing classification speed and furthermore improves significantly the accuracy of the PNN committee. Owning to the evident effectiveness of the method in distributed privacy-preserving P2P systems we plan to study it for conventional Neural Network ensemble selection.

References of chapter 9

[1] Wang L. and Fu X., (2005) Data Mining with Computational Intelligence. Springer-Verlag.

[2] Kargupta H. and Sivakumar K., (2004) Existential Pleasures of Distributed Data Mining. Data Mining: Next Generation Challenges and Future Directions, AAAI/MIT press.

[3] Datta S., (2006) Bhaduri K., Giannella C., Wolff R., and Kargupta H., Distributed data mining in peer-to-peer networks. IEEE Internet Computing, vol. 10, no. 4, pp. 18–26.

[4] Wu, X. (2011) Research on Privacy Preservation in P2P Systems. International Journal of Advancements in Computing Technology, vol. 3, no. 8, pp. 324 -330.

[5] Tresp V., (2002) Committee Machines. Chapter in Handbook of neural network signal processing, (Hu Y. H., and Hwang J.N. eds.) CRC Press LLC.

[6] Hansen L.K., and Salamon P., (1990) Neural network ensembles. IEEE Transactions on Pattern Analysis and Machine Intelligence, vol. 12, pp. 993–1001/

[7] H. Drucker, (1997) Fast Committee Machines for Regression and Classification. Proc. KDD-97.

[8] Poggio T. and Girosi F., (1990) Regularization algorithms for learning that are equivalent to multilayer networks. Science, 247, pp. 978–982.

[9] Girosi F., Jones M. and Poggio T., (1995) Regularization theory and neural networks architectures", Neural Computation, vol. 7, pp. 219-269.

[10] Evgeniou T., Pontil M., and Poggio T., (2000) Regularization Networks and Support Vector Machines. Advances in Computational Mathematics.

[11] Seni G. and Elder J., (2010) Ensemble Methods in Data Mining. Morgan & Claypool publishers

[12] Kantarcioglu M. and Vaidya J. (2003) Privacy-Preserving Naive Bayes Classifier for Horizontally Partitioned Data. Proc. IEEE Workshop on Privacy- Preserving Data Mining

[13] Yu H., Jiang X., and Vaidya J. (2006) Privacy-Preserving SVM using nonlinear Kernels on Horizontally Partitioned Data. Proc. SAC Conference.

[14] Xiong L., Chitti S., and Liu L., (2006) k nearest neighbour classification across multiple private database. Proc. ACM Fifteenth Conference on Information and Knowledge Management.

[15] Kokkinos Y. and Margaritis K.G. (2012) A Regularization Network committee machine of isolated Regularization Networks for distributed privacy-preserving data mining. In Iliadis L. et al. eds., (AIAI 2012), IFIP AICT 381, pp 97–106.

[16] Specht D., (1990) Probabilistic neural networks. Neural Networks, vol. 3, pp.109-118.

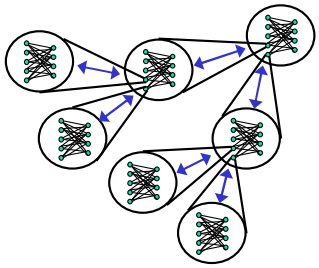
[17] Traven H.G.C., (1991) A neural network approach to statistical pattern classification by "semi-parametric" estimation of probability density functions. IEEE Transactions on Neural Networks, vol. 2, pp. 366–377

[18] Grim J., Kittler J., Pudil P., and Somol P., (2002) Multiple classifier fusion in probabilistic neural networks. Pattern Analysis & Applications, vol 5, pp. 221–233,

[19] Georgiou V.L., Alevizos P.D., and Vrahatis M.N., (2008) Novel approaches to probabilistic neural networks through bagging and evolutionary estimating of prior probabilities. Neural Processing Letters, vol. 27, pp. 153–162.

10 Hierarchical Markovian Radial Basis Function Neural Network classifier

This chapter presents a Hierarchical Markovian Radial Basis Function Neural Network (HiMarkovRBFNN) model that enables recursive operations. The hierarchical structure of this network is composed of recursively nested RBF Neural Networks with arbitrary levels of hierarchy. All hidden neurons in the hierarchy levels are composed of truly RBF Neural Networks with two weight matrices, for the RBF centers and the linear output weights, in contrast to the simple summation neurons with only linear weighted combinations which are usually encountered in ensemble models and cascading networks. Thus the neural network operation is exactly the same at all levels of the hierarchical integration. The hidden RBF response units are recursive. The training methods also remain the same for all levels as in a single RBFNN.



We present a Hierarchical Learning structure (a nested system that allows data access in a recursive fashion) which derived from a classical divide-and-conquer parallel programming strategy that can use tree-based message passing (recursive) to create one hierarchical model. Arrows represent the messages passing from and to processor nodes that hold neurons and data.

The simplicity in the Neural Network construction process is demonstrated by means of three textbook algorithms, namely the well known k-means clustering, the classical tree-based recursion function and a standard regularized least squares solver. Comparisons with the two standard model meta-learning architectures, namely Committee Machines and Cascaded Machines, reveal that the proposed method produces similar results and compares well as a combiner that can merge many HiMarkovRBFNN child nodes into one higher level parent HiMarkovRBFNN node of the same functionality.

10.1 Introduction to hierarchical models

In parallel processing [1] hierarchical structures intrinsically facilitate parallelism, and naturally alleviate many common scalability problems through the divide-and-conquer strategy [2]. For example in order to manage a large and complex dataset we can divide the data so as to create a structure of hierarchical nested data clusters. Similarly, a hierarchical neural network [3] tries to break up a complex task into a series of

simpler and faster computations at each level of the hierarchy. When the neural network is in operation, the outputs of the lower levels are combined into the higher levels. Scalability issues can be alleviated, if the problem in question can be divided and solved in several independent sub-tasks [2]. In this way computational resources can be economized when the neuron connections break up into smaller hierarchical groups that comprise modules with fewer sub-connections. During training the hierarchical decomposition inherently facilitates parallelism. Overall computation can be faster when the simpler lower level operations can be performed independently in parallel. Although hierarchies offer a major advantage for efficient implementations there are fundamental issues concerning sample complexity, functional decomposition, hierarchical representation, and training algorithms.

Thus, the main target of this study is to build a hierarchical modular nested RBF Neural Network classifier which has levels of training that make possible a faster hierarchical learning. The RBF units are organized in a hierarchical fashion and the network adapts itself into a structure of hierarchical nested data clusters. We demonstrate the simplicity in the building process by using three textbook algorithms, namely the well-known k-means clustering, the classical tree-based recursion function for the hidden response units and the standard regularized least squares solver for the linear output weights.

The Radial Basis Function Neural Network (RBFNN) [4-13] is one of the most representative types of Artificial Neural Network models, well established over the years, and widely applied in many science and engineering fields. Owning to their compact topology and fast learning RBFNNs have been extensively used for function approximation, classification, system identification and control tasks. At present, the many recent studies [14] [15] [16] [17] [18] [19] on RBFNNs and their variations prove the continuing interest on them. A single RBFNN uses Radial Basis Functions (RBF) as local activation units. This approach was inspired by the presence of many locally response units that exist in the human brain. Typically Gaussian functions are employed as hidden RBF units. Those RBFs can facilitate hierarchical learning.

In view of the fact that the locality principle operation of RBF units implies that a hierarchy is feasible some very interesting hierarchical RBF neural network models have appeared during the last years. Pioneering works include the hierarchical RBF model of Ferrari and co-workers [20][21][22][23] that was specialized for multi-scale approximation in 3-D meshing. The hierarchical RBF model of Mat Isa et al. [24] was proposed for clinical diagnosis. The hierarchical RBF model of Van Ha [25] that employs the k-nearest rule and several stage networks was proposed to solve medical diagnosis problems. The flexible hierarchical RBF model by Chen and co-workers [26] [27] [28] was applied in system identification, classification and time-series forecasting. Although not proposed for RBFNN we must mention the well-known hierarchical mixture of experts of Jordan and Jacobs [29] suitable for hierarchical mixing experts with gating networks. The existing hierarchical models are applied for different purposes and use different training algorithms than those in the original RBFNN since the first is a multi-gridding approach, the second is a cascading approach of two neural network modules, the third is a cascading of several modules, the fourth is an evolving neural tree and the fifth is a mixture of experts. However, their common ground is that they all built the prediction function through some form of hierarchical iterative approximation.

Several types of hierarchies in the previous models can be recognized. At this point we attempt a primary categorization of the hierarchical types, in order to create a small taxonomy and use a common terminology. For example, multi-resolution is the use of multiple scales of resolution. Modular usually refers to a system composed of many neural network modules. Cascading typically means that the outputs of the first level are used as inputs into the second level in the hierarchy. In the model of Ferrari and co-workers there are many RBF units, with different in scale width parameters, aligned in a 3D grid multi-resolution hierarchy and act as tree nodes and one RBFNN that composed of all of them. Ferrari et al. uses a linear combining function, the weighted sum of all RBFs. Mat Isa et al. uses two RBFNN modules cascading together in which the outputs of the first become inputs of the second. In the model of Chen and co-workers there are many small RBFNNs arranged in a hierarchy to form an acyclic graph, whose RBFNN nodes have different input features and are composed of as many hidden RBF neurons as their input neurons. Chen et al. model also uses cascading, in which outputs of the previous level RBFNNs become inputs of the next. Within this context one can detect many types of hierarchies, like hierarchical multi-resolution (Ferrari et al. model), two-level modular cascading (Mat Isa et al. model), hierarchical modular cascading (Chen et al. model, Van Ha model), hierarchical mixture of experts (Jordan and Jacobs model) and hierarchical modular nested (this work). The last type, which is the subject of our study, has many RBFNN modules hierarchically nested together with a recursive operation. Section 2 will further elaborate on the existing types of hierarchies.

The contribution of the study and the essence of the proposed solution are the following. For a classification function one can observe that the main problem for an intrinsically hierarchical structure is to support recursion, meaning same operation at all levels. Without defining and using recursion it is difficult to build a truly hierarchical nested structure with the same functionality from top to bottom. In view of this problem we employ the Markov rule in order to define a recursive RBF response function. Accordingly, we proposed a Hierarchical Markovian RBF Neural Network (HiMarkov RBFNN) topology and a training framework that can use the same training stages and algorithms from the conventional RBFNNs. In the present work (which elaborates and improves on our earlier work [30]) the HiMarkovRBFNN has a tree structure with a clear recursive operation in every node. Generally, in a tree structure of data one can take many close to each other data clusters and merge them into a single higher level cluster that contains all of them nested. In the same way, a parent HiMarkovRBFNN node takes many children HiMarkovRBFNN nodes and merges them into a single higher level node that contains all of them nested. Thus a distinctive feature of the proposed approach is that each hidden 'neuron' in every level of the hierarchy is another fully-functional HiMarkovRBFNN having inside other HiMarkovRBFNNs nested, and not just simple units that combine the previous level outputs. The classical two synaptic weight sets, for the RBF centers and the linear output weights, are present in each module and the topology is the same for all. The operation of all the modular nodes starting from the bottom level in the hierarchy and moving towards the top is also the same, as a result of the recursion. In addition, all those HiMarkovRBFNNs are individually trained in the same way, using conventional RBFNN training algorithms, well known from the literature. The selection of the hierarchically arranged RBF centers can be performed top-down from coarse-grained to fine-grained levels, using conventional clustering algorithms. Consequently, the calculation of the linear output weights at each module starts from the bottom by using

traditional regularized least squares and moves up towards the higher level modules. While the HiMarkovRBFNN uses hierarchical learning, the final network outputs are in essence a linear combination of the RBF units in the leaf nodes. Thus the internal symmetry of the hierarchical topology and the simplicity of the training strategy renders the method promising.

The rest of the chapter is organized as follows. Section 2 provides a literature review and background knowledge on hierarchical learning neural networks. A small taxonomy is also attempted here. Section 3 gives the basics of a Radial Basis Function Neural Network. Section 4 describes the proposed Hierarchical Markovian RBF Neural Network (HiMarkovRBFNN) topology. Section 5 presents the HiMarkovRBFNN training framework. Specifically section 5 elucidates on the top-down training phase for selecting the RBF centers, as well as the bottom-up training phase for solving the linear output weights. Section 6 discusses some topics on the hierarchical computations involved, their complexity and their nature. Section 7 analyses the Committee Machines and Cascading Machines we use for comparison. Section 8 presents experimental simulations on the proposed HiMarkovRBFNN, and section 9 gives summaries and possible future research.

10.2 Related work in hierarchical neural networks

While supervised hierarchical learning approximation ideas like the '*multi-resolution hierarchy*' in [3] are rather old in the neural network community, extensive studies in such hierarchical models have emerged only during the past few years. In the introduction we have tried, for simple taxonomy purposes, to label the different existing hierarchical models with terms such as '*modular*', '*nested*' and '*cascading*', which are also quite old concepts. Modularity is a key concept in engineering. Many biological networks, from neural networks in the brain to gene regulatory networks, are also organized in a '*modular*' fashion into smaller modules, which have a lower cost of sub-network connections [31]. Like in engineering, a module can be composed of many other sub-modules, and each of them could have inside other sub-modules in a '*nested*' fashion. The '*cascading*' expresses the idea of viewing the outputs of one or more module predictors as a new feature vector for another predictor, and could be traced back to 1962 in the book of Sebestyen [32], who proposes cascade machines in which the output of a predictor is used as input in the next predictor in the sequence, and so on.

From this point of view at least five categories exist, namely hierarchical multi-resolution (Ferrari et al. model), two-level modular cascading (Mat Isa et al. model, Yu et al. model), hierarchical modular cascading (Chen et al. model), hierarchical mixture of experts (Jordan and Jacobs model) and hierarchical modular nested (the HiMarkovRBFNN in the current work), which are analysed next.

Ferrari and co-workers proposed for function approximation and 3-D meshing an innovative multi-resolution model called Hierarchical Radial Basis Function (HRBF), which was extensively studied in [20][21][22][23]. It is one RBFNN (not modular) with many hierarchically arranged RBF units, located in many levels of a 3-D grid. All RBF units share the same feature space of the unknown input vector. The model is additive and the sum of all the weighted RBF units gives the final output. Some distinct differences should be mentioned for clarity. The model of Ferrari et al. has hierarchical layers, each containing a Gaussian grid at a decreasing scale. That is why it is suitable for 3D processing. The density of Gaussian RBFs increases at the lower levels while

their combination is their weighted sum. For scaling the Gaussian width parameter is the same for all Gaussians in a particular layer in the gridding approach. The training phase is based on a hierarchical gridding of the input space where additional layers of Gaussians at lower scales are added when the residual error is higher. The residuals of the RBF units in one level are re-weighted in the next level by the Nadaraya-Watson estimator carried out locally, to find the local weighted mean, on a sub-set of the data points. During the reconstruction phase all the multi-resolution different RBF layers work on the same input data and the same hidden layer. This model which is proven successful for multi-resolution analysis could be placed into the hierarchical multi-resolution category.

Intended for clinical diagnosis the model of Mat Isa et al. [24] uses two RBFNN network modules cascading together, where the first module classifies and filters the data and the second uses only the particular feature vector that is provided by the first. This model fits well into the two-level modular cascading category. Although this category is not hierarchical in the broad sense, any model with two-level RBFNN modules can fall into here. Thus, another elucidative example of a two-level modular RBF model is the multistage RBF neural network ensemble by Yu and co-workers [33], created for exchange rates forecasting. Like the previous model it is restricted to only two levels cascading together. The first level produces a large set of individual RBFNN modules for regression, whose outputs are fed as inputs into the second level RBFNN that acts as a meta-learner combiner. Typically, like any meta-learner, the second level RBFNN has input neurons as many as the ensemble modules. While, in general, the hidden neuron number could vary, a fixed such number can simplify the meta-learner training. Hence the work in [33] presets the number of hidden neurons in the second level RBFNN equal to its input neurons. The training in [33] is performed by first creating the well known stacking level-1 set, using the outputs of the ensemble members on a separate validation set. In stacking the number of attributes in a meta-level (level 1) instance is equal to the number of RBFNNs. Then these meta-level instances are used to train the second level RBFNN. Since the outputs of the ensemble modules are passed on as inputs into the next module, this paradigm also belongs into the two-level modular cascading category.

The flexible hierarchical RBF neural network model was proposed by Chen and co-workers [26][27], also called flexible neural tree [26][27] from which it was originated. This model is consisted of multiple neural networks assembled in the form of an acyclic graph. It has many hierarchically arranged small RBFNNs, each one having different input neurons, thus working much like the ensembles of features. Chen and co-workers preset the number of hidden neurons at each RBFNN module equal to the number of its input neurons. Like the cascading machines, the outputs of the RBFNN nodes (or neural tree nodes) in one level become the inputs received in the next level. The parameters for RBF centers, weights and widths are all treated in the same way as in the neural tree model [26]. The hierarchical neural structure in these studies was evolved through various evolutionary training algorithms, which share no resemblance with the conventional RBF training. This model may fit in the category of hierarchical modular cascading.

Another modular cascading HRBFN model was proposed by Van Ha in [25], where some of the input data were rejected based on an error criterion at the end of each level. It uses the k nearest neighbour to detect the classification error. The rejected data are converted into other vectors by a nonlinear transform before they were sent to the next

stage network. A new stage network then is constructed to classify them using the nearest neighbour method as decision rule. The rejected data become input to the next level where the number of neurons (Gaussian units) is determined as a logarithmic function of the number of rejected data.

For hierarchical dynamic domain decomposition in gating networks, the well-known hierarchical mixture of experts was studied by Jordan and Jacobs [29]. This is a classical neural network committee machine [6] that serves as a neural paradigm that is composed of several neural network modules. The mixture of experts [29] is an ensemble of several neural network experts, which can operate differently, and each one can specialize in a different sub-region of the input (or feature) space. The input space is divided and conquered by a gating network that dynamically allocates input-dependent weights for these experts. The experts located in leafs of this hierarchy are dynamically combined together from the bottom to the top via the gating networks. A gating network at each level examines the input vector and computes the linear weights for the particular expert networks. Only the lowest level experts can be RBFNN modules. Thus, hierarchical modular dynamic cascading is one possible operation that is performed here. Over the years this pioneering model has been an excellent text-book paradigm for committees, and preserves its own category, that of the hierarchical mixture of experts.

Up to this point we have tried to cover existing supervised hierarchical learning algorithms based on Radial Basis Functions. Differently from supervised tasks there are several noteworthy un-supervised competitive learning hierarchical methods which have been proposed. Such competitive learning methods can provide non-linear mappings from high-dimensional input spaces (with several features possibly arranged spatially, like in images) to low-dimensional output spaces. Out of them, the Neocognitron model [34], suitable for hierarchical 2D image representations, is the first hierarchical feed-forward neural network that uses feature detectors of increasing complexity at each level to extract features of different spatial resolution. The neuro-biologically inspired, pyramidal with local recurrent connectivity, improved version of the Neocognitron type is the Neural Abstraction Pyramid (NAP) Architecture [35]. NAP is a hierarchical locally recurrent network architecture [35] that uses unsupervised competitive learning and yields a hierarchy of sparse feature maps at multiple abstraction levels ([35] surveys many hierarchical image interpretation methods like Convolutional Networks for feature extraction, Helmholtz machines, hierarchical products of experts, and hierarchical Kalman filters). Besides local recurrence, the self-organization can also be implemented hierarchically. Note here that while the aforementioned methods for image representation use one neural network with several hierarchically arranged neurons, the following self-organized hierarchies use several hierarchically arranged modules. Existing self-organized Neural Networks that are based exclusively on competitive learning are the hierarchical versions of the Self-Organizing Feature Map (SOM) [36]. These special architectures are composed of independent SOM modules arranged in layers. Another competitive learning method is the hierarchical fast learning artificial neural network (HieFLANN) [37] that clusters a feature similarity matrix, in order to partition the feature space into several homogeneous feature subspaces. HieFLANN uses many collaborative modules of k-means fast learning artificial neural networks, which are single-layer self-organized clustering networks, designed for creating input-output mappings similarly to Adaptive Resonance Theory competitive learning.

The previous list, far from being complete, intends to show central dissimilarities that exist between the several approaches of hierarchically arranged neural networks which might sound similar to each other in the first place, but they are actually quite different.

By endorsing hierarchies in supervised learning, we propose a hierarchical Markovian RBF NN (HiMarkovRBFNN) that consists of nested nodes that have the same operation in every level. All RBFNNs share the same input space and no ensemble oriented training method is required. The proposed topology is symmetric, theoretically sound and the training phase for computing the weights is conducted hierarchically, level to level, starting from the bottom and moving up via any learning algorithm.

Using the proposed HiMarkovRBFNN, the computationally intensive calculations are likely to be the inversion of large Gram matrices ($\mathbf{G}^T\mathbf{G}$), or long gradient descent run times, since small sets of RBF centers are now existent in the nodes of every level. Thus the training procedure is basically transformed into simple Gaussian summations needed for the several small regressor matrices $\mathbf{G}$. Such summations are especially known for their ability to be efficiently performed in parallel.

In our model there are many RBFs hierarchically placed in a tree as a consequence of many hierarchically arranged RBFNNs. Since only the RBFs in the leaf nodes depend on the unknown $\mathbf{x}$, an outsider sees one RBFNN with radial basis functions as many as their leaf population. Only an insider can see the internal hierarchical structure which is fixed. Thus, even though the HiMarkovRBFNN has hierarchical levels of training, and in this way facilitates hierarchical learning, the network output is a linear combination of the RBF units in the leafs.

10.3 Radial Basis Function Neural Network basics

In the mathematical formulas that follow an uppercase bold letter will symbolize a matrix, while a lowercase bold letter will symbolize a vector, and an italic letter will denote a scalar variable. Fig. 10.1 illustrates the meaning of some symbols and notations used.

The conventional RBFNN has d input neurons, which correspond to the data features, K hidden neurons that hold the RBF units and M output neurons that represent the data classes. An RBF Neural network is composed of three layers, namely the input, hidden and output, with two synaptic weight matrices $\mathbf{C}$ and $\mathbf{W}$ in between them.

N – the number of training data
M – the number of data classes
$\mathbf{x}$ – a single training example or input vector
K – the number of root RBF centers and disjoint data partitions
$f_m(\cdot)$ – the output for class m (output neuron m)
$\boldsymbol{\sigma}$ – the width (radius) for a Gaussian function
$\mathbf{C}$ – the matrix of the RBF centers in a RBF Neural Network
$\mathbf{c}_i$ – the i^{th} RBF center
$\mathbf{W}$ – the matrix of the linear output weights in a RBF Neural Network
$w_{m,i}$ – the linear weight value for m^{th} class and i^{th} RBF unit
$\mathbf{G}$ – the regression matrix
$\mathbf{I}$ – the identity matrix
λ – the regularization parameter
$\mathbf{y}$ – the vector of the desired labels (used for a training process)

Figure 10.1. Common symbols and notations.

In the conventional RBFNN (see also the nodes of the HiMarkovRBFNN in fig. 10.2) the hidden RBF layer is nonlinear whereas the output layer is linear. The hidden neurons form a layer of K receptive fields, which have localized response Radial Basis Functions and map the input space onto a new space. For RBF responses in the hidden layer popular choices are Gaussian functions with appropriate centers $\mathbf{c}$ and Gaussian width $\boldsymbol{\sigma}$. The hidden RBF neurons are fully connected to all input neurons, with their synaptic weight matrix $\mathbf{C}$ to be formed by the centers $\mathbf{c}_i$ of the radial basis functions. Here an i^{th} hidden neuron computes the Euclidean distance between an unknown input vector $\mathbf{x}$ and the center vector $\mathbf{c}_i$. A radial basis function is applied to this distance to form the RBF response. The resulting outputs of the hidden neurons are communicated via linear weights $\mathbf{W}$ to the M class-output neurons of the output layer where the sums are calculated. The matrix $\mathbf{W}$ which equals $[\mathbf{w}_A\ \mathbf{w}_B\ \mathbf{w}_C]^T$ for the three-class case in fig. 10.2 has size $3\times K$. An output unit for the m^{th} class is given by the summation (eq. 10.1):

$$f_m(\mathbf{x}) = \sum_{k=1}^{K} w_{m,k} \cdot \exp(-||\mathbf{c}_k - \mathbf{x}||^2/\boldsymbol{\sigma}_k^2) \tag{10.1}$$

where $w_{m,k}$ is the linear weight of the k^{th} hidden neuron for the m^{th} class, and the Gaussian function is used to model each hidden RBF response.

With a given training set $\{\mathbf{x}_i, y_i\}_{i=1}^{N}$, the training algorithm of the network searches the parameter space to find the best of all centers $\mathbf{c}$, widths $\boldsymbol{\sigma}$ and weights w (the center coordinates, widths and heights of the Gaussian bells). There are several different algorithms for optimized training [9] of such RBF neural networks, and there is a vast literature on their training methods. RBFNN training is faster than that in MLP neural networks [5], and the RBF hidden layer is much easier to interpret, by following simple locality principals. The typical and the fastest RBF learning process, on which we focus here, is divided into three consequent steps [8][9][12], corresponding to the three distinct sets of the network parameters. The first step consists of determining the hidden unit centers, while in the second step their width parameters are estimated. During the third step, the linear output weights are determined and optimized by regularization.

10.4 The Hierarchical Markovian RBFNN Topology

The HiMarkovRBFNN idea originates from the following observation. Typically, by dividing the training set into several domains of possibly disjoint subsets, one can train a Neural Network model for each data partition. This first step can produce an ensemble of predictors. In our case, each predictor is realized by a local RBFNN. If one is able to extend this simple approach and manage to utilize another RBFNN as a combiner of the predictors then this strategy can create the first building block towards a hierarchical RBFNN. The difficultly that prevents this hierarchy to continue into higher levels can be overcome if the combiners, the parent nodes, have exactly the same characteristics of the child node predictors they combine. This necessity means that any parent node must have the same topology and the same RBF response functions with their linear weights as its child nodes. This key requirement implies an embedded architecture. In addition, if it is possible to train the higher level RBFNN with the same training algorithms used in the lower levels RBFNNs then the second key requirement is clear. The previous two requirements have a common factor, which is the recursive operation, like the familiar one used in a binary tree search. Therefore, by defining a mechanism (in our case a recursive RBF response) that supports a clear

recursion, all the key requirements can be met. Consequently, the main problem for the hierarchy to be created is to implement the recursive operation inside the RBF Gaussian response functions. This is done by using the Markovian property that specifically sums Gaussian units to create another Gaussian. By placing these Gaussian units in a tree (see fig. 10.2), each unit sums the units below it. In the continuum space these sums correspond to Gaussian integrals like the following (eq. 10.2):

$$\exp(-||\mathbf{c}'-\mathbf{x}||^2/(\sigma'+\sigma'')^2) \sim \int \exp(-||\mathbf{c}'-\mathbf{c}''||^2/\sigma'^2)\exp(-||\mathbf{c}''-\mathbf{x}||^2/\sigma''^2)\,d\mathbf{c}'' \qquad (10.2)$$

Fig. 10.2 illustrates the symmetric architecture of the proposed HiMarkovRBFNN, as well as its operation in which the Markovian property is applied. At the bottom levels any neural network located in a leaf of the tree is RBFNN in nature. The same holds also for all the networks in the other levels. They all use Gaussian functions with specific RBF centers that form the hidden neurons which are connected with the output neurons via the linear weighted links. The RBF response functions $\varphi()$ and $\varphi'()$ are recursive. The nested hierarchy is also demonstrated in fig. 10.2.

$$Class(\mathbf{x}) = \arg\max_m (f_m(\mathbf{x}))$$

$$f_A(\mathbf{x}) = \sum_{k=1}^{K} w_{A,k}\cdot\varphi_A(\mathbf{c}_k,\mathbf{x})$$

$$f_B(\mathbf{x}) = \sum_{k=1}^{K} w_{B,k}\cdot\varphi_B(\mathbf{c}_k,\mathbf{x})$$

$$f_C(\mathbf{x}) = \sum_{k=1}^{K} w_{C,k}\cdot\varphi_C(\mathbf{c}_k,\mathbf{x})$$

$$\varphi_m(\mathbf{c}_k,\mathbf{x}) = \sum_{i=1}^{K'} h_m(\mathbf{c}_k,\mathbf{c}'_i)\cdot w'_{m,i}\cdot\varphi'_m(\mathbf{c}'_i,\mathbf{x})$$

$$f'_A(\mathbf{x}) = \sum_{i=1}^{K'} w'_{A,i}\cdot\varphi'_A(\mathbf{c}'_i,\mathbf{x})$$

$$f'_B(\mathbf{x}) = \sum_{i=1}^{K'} w'_{B,i}\cdot\varphi'_A(\mathbf{c}'_i,\mathbf{x})$$

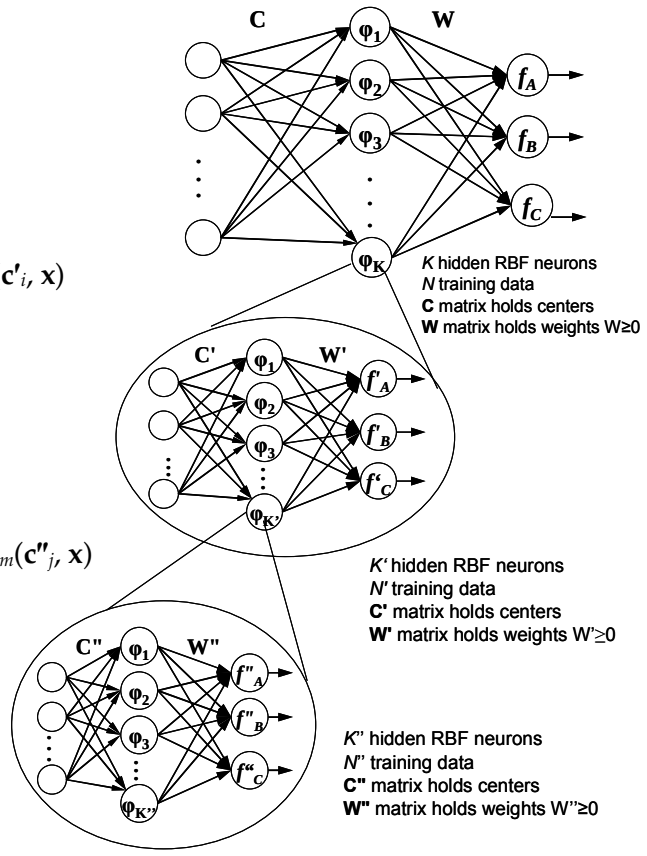
$$f'_C(\mathbf{x}) = \sum_{i=1}^{K'} w'_{C,i}\cdot\varphi'_C(\mathbf{c}'_i,\mathbf{x})$$

$$\varphi'_m(\mathbf{c}'_i,\mathbf{x}) = \sum_{j=1}^{K''} h'_m(\mathbf{c}'_i,\mathbf{c}''_j)\cdot w''_{m,j}\cdot\varphi''_m(\mathbf{c}''_j,\mathbf{x})$$

$$f''_A(\mathbf{x}) = \sum_{j=1}^{K''} w''_{A,j}\cdot\varphi''_A(\mathbf{c}''_j,\mathbf{x})$$

$$f''_B(\mathbf{x}) = \sum_{j=1}^{K''} w''_{B,j}\cdot\varphi''_B(\mathbf{c}''_j,\mathbf{x})$$

$$f''_C(\mathbf{x}) = \sum_{j=1}^{K''} w''_{C,j}\cdot\varphi''_C(\mathbf{c}''_j,\mathbf{x})$$



$$\varphi''_m(\mathbf{c}''_j,\mathbf{x}) = \exp(-||\mathbf{c}''_j-\mathbf{x}||^2/\sigma''^2_j)$$

$$h_m(\mathbf{c}_k,\mathbf{c}'_i) = \exp(-||\mathbf{c}_k-\mathbf{c}'_i||^2/\sigma_k^2)$$

$$h'_m(\mathbf{c}'_i,\mathbf{c}''_j) = \exp(-||\mathbf{c}'_i-\mathbf{c}''_j||^2/\sigma'^2_i)$$

Figure 10.2. Hierarchical Markovian RBF Neural Network topology and operation for the three-class problem. The network spreads in a radial fashion. All the functions $\varphi''()$, $h()$ and $h'()$ are Gaussians. Only those are stored in the tree. The functions $\varphi()$ and $\varphi'()$ symbolize the recursive responses via the Markov summations of these Gaussian functions.

The hierarchy is based on the divide-and-conquer strategy which is widely used in many areas in machine learning [2] for dividing a complex concept into a set of simpler models. Example applications of this strategy for speech recognition are the Hidden Markov Models that consists of a series of naïve Bayes classifiers whose decisions are merged by a transition matrix [2] to learn a complex concept. Markov Models take advantage of the Markov property which simply says that contextual constraints are given by local interactions.

The proposed HiMarkovRBFNN structure (fig. 10.2) exploits the simple Markov chain property that sums over all intermediate centers $\mathbf{c}'_i$ between the vector $\mathbf{x}$ and the center $\mathbf{c}_k$ in order to explicitly output the recursive RBF response function $\varphi(||\mathbf{c}_k-\mathbf{x}||)$ given by eq. 10.3:

$$\varphi(||\mathbf{c}_k-\mathbf{x}||) = \sum_{i=1}^{K'} h'(||\mathbf{c}_k-\mathbf{c}'_i||)\cdot w'_i \cdot \varphi'(||\mathbf{c}'_i-\mathbf{x}||) \quad (10.3a)$$

$$h(||\mathbf{c}_k-\mathbf{c}'_i||) = \exp(-||\mathbf{c}_k-\mathbf{c}'_i||^2/\boldsymbol{\sigma}_k^2) \quad (10.3b)$$

$$\varphi'(||\mathbf{c}'_i-\mathbf{x}||) = \sum_{j=1}^{K''} \exp(-||\mathbf{c}'_i-\mathbf{c}''_j||^2/\boldsymbol{\sigma}'^2_i)\cdot w''_j \cdot \exp(-||\mathbf{c}''_j-\mathbf{x}||^2/\boldsymbol{\sigma}''^2_j) \quad (10.3c)$$

The Gaussian functions are recursively summed up. A center $\mathbf{c}_k$ in the first level of fig. 10.2 has nested other centers $\mathbf{c}'_i$ which by their part have nested other centers $\mathbf{c}''_j$. For any given pair of a center $\mathbf{c}$ and an unknown $\mathbf{x}$ each $\varphi()$ outputs a likelihood value through out a recursive summation. For such Gaussian distributions the Markovian property in the continuum space says that $P(m|\mathbf{c},\mathbf{x})=\int P(m|\mathbf{c},\mathbf{c}')P(\mathbf{c}')P(m|\mathbf{c}',\mathbf{x})d\mathbf{c}'$, in which the assumption is that the intermediate transitions between the steps $\mathbf{c}$ and $\mathbf{c}'$ are based on Gaussians. Thus, eq. 10.3 presents the discretized approximation of the Markovian property. Eq. 10.3c is the discrete version of eq. 10.2.

In eq. 10.3 the right side of the function is a root node (parent) Gaussian unit that is produced by summing all the intermediate Gaussian units between $\mathbf{c}$ and $\mathbf{x}$. In eq. 10.3 a parent center $\mathbf{c}_k$ has K' child centers $\mathbf{c}'_i$. The first Gaussian term $h(||\mathbf{c}_k-\mathbf{c}'_i||)$ which equals $\exp(-||\mathbf{c}_k-\mathbf{c}'_i||^2/\boldsymbol{\sigma}_k^2)$ is the typical cost of choosing $\mathbf{c}_k$ when the true pattern is $\mathbf{c}'_i$. Since other Gaussian centers $\mathbf{c}'_i$ exist between $\mathbf{c}_k$ and $\mathbf{x}$, their contributions are summed up. Thus, the second term $\varphi'(||\mathbf{c}'_i-\mathbf{x}||)$ is the next level recursive response to $\mathbf{x}$. Hence, the summation is over all contributions from the child centers that belong to the same cluster group of the parent center $\mathbf{c}_k$. As a result, the HiMarkovRBFNN allows for an explicit hierarchically structured framework where several classical RBFNN training algorithms can be applied.

Recall that the HiMarkovRBFNN tries to overcome common scalability problems encountered in the prohibitively slow neural network training for large datasets. Thus, for scalability reasons any RBFNN must have few K hidden neurons with Gaussian response functions. While few responses $\varphi()$ can capture only a coarse-grained probability map between $\mathbf{c}$ and $\mathbf{x}$, the accuracy can be enhanced if these responses encapsulate indirectly other intermediate calculation steps in a more fine-grained level (eq. 10.3). The decision profile becomes more detailed while moving from one level to the next.

Notations and symbols in fig. 10.2 are essential. The single RBFNN in the first level (root) uses four symbols, namely K for the number of its hidden RBF neurons, N for the number of its training data, $\mathbf{C}$ for the RBF centers matrix and $\mathbf{W}$ for the linear output weights matrix. Less subscripts and superscripts can reduce notational complexity.

Thus for convenience, and to avoid unnecessary complex indices, a symbol or function with a prime ′ will indicate that this symbol belongs to the second level of the hierarchy. A double prime ″ will indicate that the symbol or function is from the third level. The population of the training examples that the root RBFNN node holds in the first level is N, which is the total dataset.

While the K children nodes of the root node that form the second level nodes actually hold N'_1, N'_2, …, N'_K examples in their own data partitions, we stress again that for simplicity a plain N' will be used to denote the number of examples inside a data partition at *any* node of the second level, and not a particular one. Likewise N'' represents the number of examples in a data partition that any node holds in the third level. This notation avoids unnecessary complex numbering in the subscripts, since the functions are complex enough. In the same way $\mathbf{C}$, $\mathbf{C}'$ and $\mathbf{C}''$ symbolize the matrices that hold the corresponding Gaussian centers $\mathbf{c}$, $\mathbf{c}'$ and $\mathbf{c}''$ at *any* node in the first, second and third level respectively. Similarly, K, K' and K'' indicate the number of Gaussian centers *any* node may hold in the first, second and third level respectively (fig. 10.2). We use $\mathbf{W}$, $\mathbf{W}'$ and $\mathbf{W}''$ for the linear output weight matrices at *any* RBFNN node in the first level, second level and third level respectively.

In fig. 10.2 all the functions $\varphi''()$, $h()$ and $h'()$ that exist in the hierarchy are Gaussians. These are the building blocks. This is indicated also in the bottom of fig. 10.2. The tree stores only these Gaussian functions with their parameters and their linear weights (and nothing else). The response functions $\varphi()$ and $\varphi'()$ just symbolize the recursive Markov summations of these Gaussian functions. A Gaussian center $\mathbf{c}_k$ has nested other Gaussian centers $\mathbf{c}'_i$ which by their part have nested other Gaussian centers $\mathbf{c}''_j$ located in leafs. Thus the hierarchical structure in fig. 10.2 is specifically composed of Gaussian RBF units $\exp(-||\mathbf{c}_k-\mathbf{c}'_i||^2/\boldsymbol{\sigma}_k^2)$, $\exp(-||\mathbf{c}'_i-\mathbf{c}''||^2/\boldsymbol{\sigma}'^2_i)$ and $\exp(-||\mathbf{c}''_j-\mathbf{x}||^2/\boldsymbol{\sigma}''^2_j)$ at the three levels, with the Gaussian widths related to their centers. These Gaussian RBF units in the three levels of fig. 10.2, are the ones that actually stored inside the network levels and are the ones that hierarchically summed up using the recursive function (see also fig. 10.3), when a new example $\mathbf{x}$ is presented.

The notation $\varphi_m(\mathbf{c}_k, \mathbf{x})$ in fig. 10.2 denotes the response of k^{th} hidden unit for class m. If a hidden neuron of the first level RBFNN in fig. 10.2 has only one Gaussian center $\mathbf{c}_k$ without a second level, then all the Gaussian responses $\varphi_A(\mathbf{c}_k, \mathbf{x})$, $\varphi_B(\mathbf{c}_k, \mathbf{x})$ and $\varphi_C(\mathbf{c}_k, \mathbf{x})$ would be equal to $\exp(-||\mathbf{c}_k - \mathbf{x}||^2/\boldsymbol{\sigma}_k^2)$. Otherwise, if each Gaussian center $\mathbf{c}_k$ in the first level has a second level of nested Gaussian centers $\mathbf{c}'_i$ which by their part have another level of nested Gaussian centers $\mathbf{c}''_j$ then the responses are computed recursively.

The partitioning of the training dataset can be performed either top-down or bottom-up. Following a top-down description in fig. 10.2 the top level HiMarkovRBFNN is composed of K clusters. Each cluster has a corresponding center $\mathbf{c}_i$, and all these centers are in the $\mathbf{C}$ matrix. By iteratively partitioning the training points inside any one of these K cluster groups, one can recursively define the clusters in the next level in the hierarchy producing the second level RBFNN nodes. Although K'_1, K'_2, …, K'_K are the number of centers formed in the second level nodes, we use simple K' to denote the number of centers at any second level node. These centers are stored in their corresponding $\mathbf{C}'$ matrices. In consequence, by partitioning the points inside the K' clusters gives the data groups for the next nodes, and so on. We use K'' to indicate the number of centers that are stored in the $\mathbf{C}''$ matrix at *any* node that lays in the third level (bottom) in fig. 10.2.

Computing the linear weights **W**, **W′**, **W″** at each level can be performed only in a bottom-up manner, because the higher levels need the weights of the lower-level nodes. Following bottom-up in fig. 10.2 the bottom-level nodes produce the $f''()$ output functions, the middle level nodes produce the $f'()$ output functions and the top level nodes produce the $f()$ ones. In order to find the linear output weights the computation starts from the bottom level for each RBF neural network using the training examples it holds. During operation in the HiMarkovRBFNN each level requires the RBF responses of the previous level in order to work. Subsequently, optimization is done with the constraint that all the **W**, **W′**, and **W″** matrix entries to be nonnegative, so as to produce nonnegative outputs for each class. To this end, we use a standard non negative least squares solver to compute the linear weights.

In fig. 10.2 there are different Gaussian widths per neural network. The root network stores K, in number, Gaussian centers $\mathbf{c}_k$ with their widths $\boldsymbol{\sigma}_k$. In the implementations we define the width $\boldsymbol{\sigma}_k$ as the minimum distance between the centers $\mathbf{c}_k$. The second level has K neural networks where each one stores K'_1, K'_2, …, K'_K in number Gaussian centers $\mathbf{c}'_i$ respectively. In each one of the second level neural networks we can set their Gaussian widths $\boldsymbol{\sigma}'_i$ equal to the minimum distance between the centers each network holds. If it is specifically required, the width parameters $\boldsymbol{\sigma}''_j$ within the RBF functions at the bottom level can become different for each class, namely class-conscious (a class-related variance inside the cluster), by using some average distance of the center from the same class points inside the cluster.

Like all methods that are based on modular classifiers, the HiMarkovRBFNN in fig. 10.2 has one requirement that needs special care. For symmetry reasons, all the classifiers in the leaf RBFNN nodes must output M classes. However, the dataset is hierarchically partitioned and some disjoint data partitions in the leaf nodes may have examples from fewer classes than the M classes of the dataset. For instance, if a leaf RBFNN node is based on a data cluster that holds no data points of class A then this leaf RBFNN node has no internal knowledge of the particular class A (and consequently the local separating boundaries of A with the other classes are unknown). In this case, weights equal to $1/M$ (M is the number of classes) are assigned to connect the hidden neuron RBFs with the output neuron of class A. Thus, if a class is missing from a leaf RBFNN node then the weights for the corresponding class-output neuron will not be zero, and hence this leaf RBFNN node will not output a zero probability for this class by default. Otherwise, since the Markov chain rule is a product rule of probabilities, any node that has all weights zero in a class would have been excluded from the computations by definition. Now by setting the weights for the missing classes equal to $1/M$ solves the problem. This strategy has a deeper meaning that works. Leaf RBFNNs that are based on dense clusters can specialise in some classes, but must keep an open mind for the missing ones.

```
double classOutput(treeNode root, example x, class m)
{       // implements f_m(x) = Σ_{k=1}^{K} w_{m,k} · φ_m(c_k, x), as well as any f'_m(x) or f''_m(x)
   sum = 0.0 ;
   for each child k  in root              // for k=1 to K  since the root has K child centers
     sum = sum + root.child[k].classWeight[m] *recursiveResponse(root.child[k], x, m);
   return sum ;
}
```

```
double recursiveResponse (treeNode parent, example x, class m)
{       // implements the recursive RBF response units φm(), φ'm(), or φ''m()
  sum = 0.0 ;
  if (parent.childsSize == 0) return exp(–||parent.c – x||2/ parent.σ2) ;
  for each child ch in parent
    sum = sum + exp (–||parent.c – parent.child[ch].c ||2/ parent.σ2) *
      parent.child[ch].classWeight[m] * recursiveResponse (parent.child[ch], x, m) ;
  return sum ;
}
```

Figure 10.3. The function classOutput() computes the typical class output $f_m(\mathbf{x})$ for the m^{th} class as a linear weighted combination of the recursive RBF responses $\varphi_m()$. The function recursiveResponse() implements the recursive responses $\varphi_m()$ which are simple summations of the Gaussians functions stored in the tree levels.

The function classOutput() illustrated in fig. 10.3 implements a traditional class output unit $f_m(\mathbf{x})$ for a particular class m. The classOutput() calls the recursive response functions. The class outputs $f_m(\mathbf{x})$, $f'_m(\mathbf{x})$ or $f''_m(\mathbf{x})$, for any particular class m, can be used for optimizing the parameters inside any individual neural network by minimizing their classification error computed from their own data partitions. These outputs are not recursive. Thus, $f_m(\mathbf{x})$ cannot be expressed in terms of $f'_m(\mathbf{x})$ and $f''_m(\mathbf{x})$ because these represent the output units in the output layer and the networks are not cascaded. However, as illustrated in figs. 10.2 and 10.3, each class output is a linear combination of the recursive RBF responses $\varphi_m()$ in the hidden layer.

The responses $\varphi_m()$ or $\varphi'_m()$ in a hidden layer are the recursive functions. $\varphi''_m()$ can also be considered as a recursive response terminated instantly (see eq. 10.11 in the discussion). Note the second function in fig. 10.3 that actually implements the recursiveResponse() which uses the recursion throughout the hierarchy. The function recursiveResponse() internally calls itself. For speed reasons, one can implement the iterative summations without actually using recursion.

Given a hierarchical structure of data clusters, the HiMarkovRBFNN tries to adapt into it. A parent node (master) tells to his childs (workers) to find the responses $\varphi_m()$, the childs tell to their childs to find the responses $\varphi'_m()$ and so on. The Gaussian function $\exp(-||\mathbf{c}_k-\mathbf{c'}_i||^2/\boldsymbol{\sigma}_k^2)$, is the cost of choosing $\mathbf{c}_k$. We set each Gaussian width $\boldsymbol{\sigma}_k$ equal to the minimum distance between all the parent centers $\mathbf{c}_k$. This shows major improvements in the performance of the HiMarkovRBFNN. The use of Markov rule to sum up probabilities does not require from the network outputs to be normalized in order to represent posteriors. The event (or joint) probabilities are multiplied two-by-two at each level during the transition stage when moving to the next level. Computed from the data using non-negative least squares, the weights of the next level re-normalize the class outputs of the previous level.

10.5 Hierarchical RBF Neural Network Training

10.5.1 Top-down selection of RBF centers

In order to determine the RBF centers this selection step can be performed either top-down or bottom-up through the hierarchy. The task is to create a tree structure composed of hierarchically arranged clusters from the training data that divide the initial dataset into partitions. Data distributions can hide multiple levels of analysis

from coarse-grained to fine-grained scales. Thus a tree structure like the paradigm in fig. 10.2 is needed to be formed with the particular RBF centers placed in their corresponding matrices **C**, **C′** or **C″** in the first, second and third level RBFNNs. Note here that the K second level nodes in fig. 10.2 actually will hold N'_1, N'_2, …, N'_K examples from which they must found $\mathbf{C}'_1$, $\mathbf{C}'_2$, …, $\mathbf{C}'_K$ center matrices respectively. Again we will use a simple **C′** to symbolize such a matrix of *any* second level node.

During the top-down RBF centers selection, several algorithms can be applied iteratively, starting from the first level, and continuing towards the next levels. Popular algorithms select the RBF centers randomly, or perform supervised center selection or employ unsupervised clustering for center selection, which is the most common strategy. Clustering algorithms such as k-Means, fuzzy c-Means or Subtractive Clustering are all well-documented for this RBF training stage. Once the clustering is completed, the mean of each cluster can be used as the center. To create the paradigm in fig. 10.2, initially, K clusters are formed in the top level, having centers that are stored in the related **C** matrix. Then, the clustering procedure is recursively repeated inside the clusters of the top level in order to find other K' clusters inside them and to store their centers in the corresponding **C′** matrices. In consequence, the same clustering is repeated inside all the K' clusters of the second level to find other K'' clusters inside them and to store their centers to the **C″** matrices of the third level. A typical stopping criterion can be defined here for terminating the recursive clustering procedure if any cluster has less than a predefined number of points, or at the opposite to further continue clustering into more than three levels in depth, if a cluster holds more than another predefined number of points.

As a demonstration to determine the clusters we utilize a textbook algorithm, which is the classical k-means clustering. The goal for the top-down version is to perform simple divide-and-conquer by selecting few cluster centers at a low granularity, and iteratively clustering the data inside them to find more clusters at a higher granularity. In other words, the algorithm finds a set of clustering solutions such that the clusters are nested across the levels of the hierarchy. A bottom-up hierarchical version is also possible for the k-means clustering where the algorithm can cluster means in the lowest layer, and then post-processing the cluster centers to find their nearest points, before proceeding to the next layer and merging the clusters together. Note that our goal here is not unsupervised learning that tries to explain the data, but rather plain and simple divide-and-conquer in order to create a hierarchical nested structure. The hiMarkovRBFNN will then adapt itself into this structure.

10.5.2 Bottom-up linear weights finding

In fig. 10.2, the corresponding linear weights **W**, **W′** or **W″** need to be determined for any RBF neural network at the first, second or third level. This calculation step is performed bottom-up through the hierarchical levels by following conventional regularized linear least squares [5][6][38]. Note that this bottom-up merging of the lower level outputs into the higher ones is possible since the network supports the same recursive operation at all levels.

Standard regularized linear least squares work as follows. Assuming K hidden neuron outputs $\varphi_j()$ and N training data examples then for a weight vector **w** and a target vector **y** that has elements the desired labels (hot encoding is used which sets y=1 for the same class labels and y=0 for the other classes) one can solve the linear system $\mathbf{Gw}=\mathbf{y}$. The regressor matrix **G** of size $K \times N$ holds the outputs of the RBF units from all

N samples as inputs. Specifically $\mathbf{G}$ has entries $\mathbf{G}[i][j]=\varphi_j(||\mathbf{c}_j-\mathbf{x}_i||)$ $\{i=1,..,N, j=1,..,K\}$ which are the corresponding hidden neuron outputs. All the weight vectors for all the classes compose the matrix $\mathbf{W}$. Hence a matrix $\mathbf{W}$ equals $[\mathbf{w}_A\ \mathbf{w}_B\ \mathbf{w}_C]^T$ for the three-class case. As there are typically more training examples than hidden neurons, N is larger than K and the system is over-determined. The weights for each class can be computed as $\mathbf{w} = \mathbf{G}^+\mathbf{y}$ where $\mathbf{G}^+= (\mathbf{G}^T\mathbf{G})^{-1}\mathbf{G}^T$ is the pseudo-inverse (Moore-Penrose inverse). However, by following Tikhonov regularization theory one can redefine the pseudo-inverse as $\mathbf{G}^+ = (\mathbf{G}^T\mathbf{G} + \lambda\mathbf{I})^{-1}\ \mathbf{G}^T$ where $\mathbf{I}$ is the unit matrix and λ a small positive regularization parameter [5][6].

Using this bottom-up linear weights finding we are going to train the three level hierarchical Markovian RBF example model in fig. 10.2, level by level. Recall that in fig. 10.2 any $\varphi''()$ indicates the RBF response function in the bottom level (third level), while $\varphi'()$ and $\varphi()$ symbolize the recursive RBF response functions for the second level and first level respectively.

In the bottom level (fig. 10.2), any RBFNN can compute the linear weights $\mathbf{w''}$ vector (where $\mathbf{w''}$ denotes $\mathbf{w''}_A$ $\mathbf{w''}_B$ or $\mathbf{w''}_C$) for any particular class of A, B or C, in terms of the $\mathbf{G''}$ matrix of size $K''\times N''$ and the vector $\mathbf{y''}$ of size N'' that has the desired labels for this class as:

$$\mathbf{w''} = (\mathbf{G''}^T\mathbf{G''} + \lambda\mathbf{I})^{-1}\mathbf{G''}^T\mathbf{y''} \qquad \text{s.t. } \mathbf{w''} \geq 0 \tag{10.4}$$

where a $\mathbf{G''}[i][j]$ element is

$$\mathbf{G''}[i][j]= \varphi''(||\mathbf{c''}_j-\mathbf{x}_i||) = \exp(-||\mathbf{c''}_j-\mathbf{x}_i||^2/\boldsymbol{\sigma}_j^2) \tag{10.5}$$

The solution has an additional constraint that the $\mathbf{w''}$ entries must be non-negative ($\mathbf{w''}\geq 0$), so as to produce non-negative outputs for each class. To this end we utilize a non negative least squares solver.

In the same way, for any middle level (second level in fig. 10.2) RBFNN node the weights $\mathbf{w'}$ vector (where $\mathbf{w'}$ denotes $\mathbf{w'}_A$ $\mathbf{w'}_B$ or $\mathbf{w'}_C$) for any particular class of A, B or C would be given, in terms of the $\mathbf{G'}$ matrix of size $K'\times N'$ and the vector $\mathbf{y'}$ of size N', that holds the desired labels for this class, by:

$$\mathbf{w'} = (\mathbf{G'}^T\mathbf{G'} + \lambda\mathbf{I})^{-1}\mathbf{G'}^T\mathbf{y'} \qquad \text{s.t. } \mathbf{w'} \geq 0 \tag{10.6}$$

where

$$\mathbf{G'}[i][j]= \varphi'(||\mathbf{c'}_j-\mathbf{x}_i||) = \sum_{k=1}^{K''} \exp(-||\mathbf{c'}_j-\mathbf{c''}_k||^2/\boldsymbol{\sigma}_j^2)\ w''_k \exp(-||\mathbf{c''}_k-\mathbf{x}_i||^2/\boldsymbol{\sigma}_k^2) \tag{10.7}$$

and the summation is estimated for the K'' in number child centers $\mathbf{c''}_k$ that the parent Gaussian center $\mathbf{c'}_j$ holds. Again the optimization is subject to the constraint that the weights must be nonnegative ($\mathbf{w'}\geq 0$). Note that there is a different $\mathbf{G'}$ matrix for each class since the output functions of the previous level are class conscious.

At the top level (first level in fig. 10.2) RBFNN the $\mathbf{w}$ vector of linear weights for a class output would be given, in terms of the $\mathbf{G}$ matrix of size $K\times N$ and the desired labels vector $\mathbf{y}$ of size N for this class, by:

$$\mathbf{w} = (\mathbf{G}^T\mathbf{G} + \lambda\mathbf{I})^{-1}\mathbf{G}^T\mathbf{y} \qquad \text{s.t. } \mathbf{w} \geq 0 \tag{10.8}$$

where

$$\mathbf{G}[i][j]= \varphi(||\mathbf{c}_j-\mathbf{x}_i||) = \sum_{k=1}^{K'} \exp(-||\mathbf{c}_j-\mathbf{c'}_k||^2/\boldsymbol{\sigma}_j^2)\cdot w'_k \cdot \varphi'(||\mathbf{c'}_k-\mathbf{x}_i||) \tag{10.9}$$

and the sum is running over the K' in number child $\mathbf{c}'_k$ RBF centers of the parent RBF center $\mathbf{c}_j$. The top level RBFNN uses all $N = N'_1 + N'_2 + .. + N'_K$ examples from the lower K partitions.

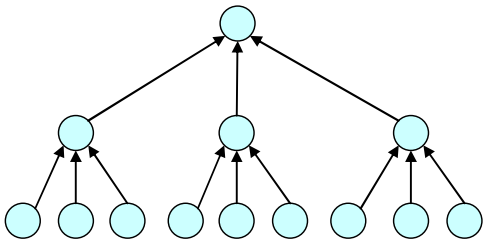


Figure 10.4. Bottom-up linear weights finding paradigm. All nodes in the tree are RBFNNs, while arrows show the order of calculations.

Each level requires all the linear weights of the previous levels, and thus this training stage starts from the bottom and continues towards the top. In fig. 10.4 there is a paradigm that shows the order of calculations as well as and the dependencies. All the child nodes under a parent are required to finish their weight calculations before the corresponding parent node can start its own.

10.6 Discussion

Here we discuss some topics on the hierarchical computations involved, their complexity, and their nature. The computational complexity of the HiMarkovRBFNN varies depending on the algorithms at each training stage. The first stage of the training is the partitioning of the dataset into cluster nodes in order to form levels of the hierarchy. If a top-down partitioning strategy is adopted then the complexity equals that of the clustering algorithm that is applied. In the second stage of the training, namely the bottom-up linear weights finding, the computational complexity depends on the number of hidden units. If a node has K hidden units and holds N training samples, then the complexity is $O(K \cdot N)$ to compute the regressor matrix $\mathbf{G}$, that has size $K \times N$, and $O(K^3)$ to invert the matrix $(\mathbf{G}^T\mathbf{G} + \lambda\mathbf{I})$ similarly to the single RBFNN case. However, the work here is distributed. The number of RBFs in every node is much smaller than N, and can be regulated. As a rule, inverting a single matrix of size $K \times K$ is cubic in K. When K is large, for example K=10000, the inversion is rather difficult to implement as well as quite slow. On the other hand, having a HiMarkovRBFNN tree with 100 nodes in the hierarchy, and solving 100 matrices with K=100 is easier and much faster.

When the network is in operation, simple Gaussian summations are used recursively. The computational complexity is bounded by the number of the Gaussian centers that are located in the leaf nodes. Every one of the $\varphi''(||\mathbf{c}''-\mathbf{x}||)$ functions in the leafs, that depend on $\mathbf{x}$, receives an unknown $\mathbf{x}$ and using the centers $\mathbf{c}''$ computes $\exp(-||\mathbf{c}''-\mathbf{x}||^2/\boldsymbol{\sigma}^2)$. The other Gaussian computations $\exp(-||\mathbf{c}'-\mathbf{c}''||^2/\boldsymbol{\sigma}^2)$ between two centers $\mathbf{c}'$ and $\mathbf{c}''$ at two different levels are fixed, and thus can be pre-computed and permanently stored. Thus in operation, if the population of all the hidden RBF units in the leafs is L then the network has $O(L)$ complexity to classify an unknown $\mathbf{x}$. This is equal to that of a single RBFNN that has L hidden RBF units. In other words, internally it is a hierarchical Markovian structure with a hierarchical training phase. However, externally an outsider can only see the L hidden units in the leaf nodes like in a single RBFNN.

Besides the model may work for practical cases, the convergence of the hierarchical model can be based on the fact that is a linear combination of the Gaussian functions $\varphi''(||\mathbf{c}''-\mathbf{x}||)$ in the leafs which are the ones that depend on $\mathbf{x}$. However like other hierarchical models the theoretical properties of this must be proven.

A first question that seems to emerge concerns the potential of any single RBFNN located in a leaf to be considered as a HiMarkov one. Thus the question is the following: can a single RBFNN classifier in a leaf be directly derived from a HiMarkovRBFNN? Yes if a parent simply has one child, itself only (which actually is the stopping criterion for the hierarchical decomposition). By using the cost $\exp(-||\mathbf{c}-\mathbf{c}'||^2/\boldsymbol{\sigma}^2)$ which is expressed by a Gaussian function, one can reproduce the final RBF response function $\varphi(||\mathbf{c}-\mathbf{x}||)$, which is also expressed by a Gaussian function, from the integral:

$$\int \exp(-||\mathbf{c}-\mathbf{c}'||^2/\boldsymbol{\sigma}^2) \cdot \varphi(||\mathbf{c}'-\mathbf{x}||)\, d\mathbf{c}' \qquad (10.10)$$

As the cost $\exp(-||\mathbf{c}-\mathbf{c}'||^2/\boldsymbol{\sigma}^2)$, of choosing $\mathbf{c}$ when the true center is $\mathbf{c}'$, meaning is that the parent center $\mathbf{c}$ has one child center $\mathbf{c}'$, itself only, the first Gaussian term in eq. 10.10 reduces to the simple delta function $\delta(\mathbf{c}-\mathbf{c}')$ (if the width limit approaches $\boldsymbol{\sigma}$-->0 then the Gaussian function becomes a delta function) and the integral becomes:

$$\int \delta(\mathbf{c}-\mathbf{c}') \cdot \varphi(||\mathbf{c}'-\mathbf{x}||)\, d\mathbf{c}' = \varphi(||\mathbf{c}-\mathbf{x}||) \qquad (10.11)$$

which gives one Gaussian RBF unit.

A second question is the following: Can a traditional single RBFNN located in a leaf be trained via the Markov summations of the series of joint probabilities? Actually, integration is what it does (see also [5]). Now let us elaborate on this by reviewing the single RBFNN training. Given K RBF centers $\mathbf{c}_j$ (j=1,..K), we know that the regressor matrix $\mathbf{G}$ of size $K \times N$ holds the outputs of the RBF units from all N training examples as inputs. In particular, $\mathbf{G}$ has entries $\mathbf{G}[i][j]=\varphi_j(||\mathbf{c}_j-\mathbf{x}_i||)$ $\{i$=1,..,N, j=1,..,$K\}$. We need to find the pair-wise joint probability distributions of the RBF centers. Using the Markov property, such a pair-wise joint probability between the center $\mathbf{c}_1$ and the center $\mathbf{c}_2$ is given by:

$$\varphi(\mathbf{c}_1, \mathbf{c}_2) = \int \varphi(\mathbf{c}_1, \mathbf{x})\, \varphi(\mathbf{c}_2, \mathbf{x})\, d\mathbf{x} \qquad (10.12)$$

which cannot be computed analytically but can be approximated with N training examples as:

$$\varphi(\mathbf{c}_1, \mathbf{c}_2) = \sum_{i=1}^{N} \varphi(\mathbf{c}_1, \mathbf{x}_i)\, \varphi(\mathbf{c}_2, \mathbf{x}_i) \qquad (10.13)$$

and because every parent RBF unit has one child, itself only, with a known Gaussian function the $\varphi(\mathbf{c}_1, \mathbf{c}_2)$ becomes:

$$\varphi(\mathbf{c}_1, \mathbf{c}_2) = \sum_{i=1}^{N} \exp(-||\mathbf{c}_1-\mathbf{x}_i||^2/\boldsymbol{\sigma}_1^2) \exp(-||\mathbf{c}_2-\mathbf{x}_i||^2/\boldsymbol{\sigma}_2^2) \qquad (10.14)$$

eq. 10.13 or eq. 10.14 simply gives the cell of the traditional matrix $\mathbf{G}^T\mathbf{G}$ (of size $K \times K$) that corresponds to the pair of centers $\{\mathbf{c}_1,\mathbf{c}_2\}$, meaning the row 1 and column 2 of $\mathbf{G}^T\mathbf{G}$. Thus the matrix $\mathbf{G}^T\mathbf{G}$ actually holds the pair-wise joint probabilities between all the RBF centers. Before using regularized least squares we must find the target values of each center.

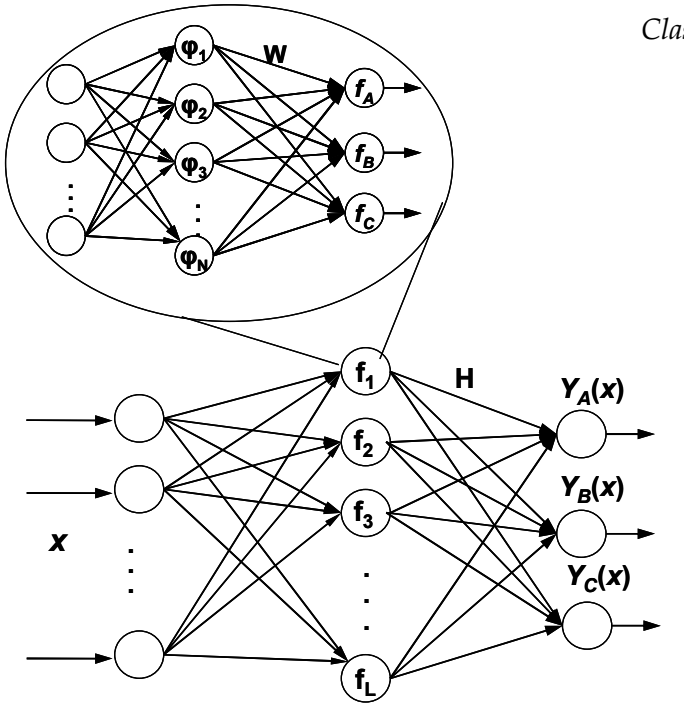
The target value $y(\mathbf{c}_j)$ for a particular center $\mathbf{c}_j$ is given by the integral $\int \varphi(\mathbf{c}_j, \mathbf{x})\, y(\mathbf{x})\, d\mathbf{x}$, where $y(\mathbf{x})$ is the desired label of variable $\mathbf{x}$ in the continuum space. Approximating this target value with the training set of N examples gives:

$$y(\mathbf{c}_j) = \sum_{i=1}^{N} \varphi(\mathbf{c}_j, \mathbf{x}_i)\, y_i = \sum_{i=1}^{N} \exp(-||\mathbf{c}_j - \mathbf{x}_i||^2 / \sigma_j^2)\, y_i \qquad (10.15)$$

where y_i is the desired label of example $\mathbf{x}_i$. This summation that corresponds to the center $\mathbf{c}_j$ equals the j^{th} element of the well known $\mathbf{G}^T\mathbf{y}$ target vector. By combining all we get for the weights $\mathbf{w} = (\mathbf{G}^T\mathbf{G} + \lambda\,\mathbf{I})^{-1}\,\mathbf{G}^T\mathbf{y}$. Hence, we end up with the same expression.

10.7 Comparison with Committee Machines and Cascading Machines

Standard model architectures of Committee Machines and Cascaded Machines which are depicted in fig. 10.5 and fig. 10.6 are strong paradigms for combining many neural network modules together.



$$Class(\mathbf{x}) = \arg\max_{m} (Y_m(\mathbf{x}))$$

$$f_{j,A}(\mathbf{x}) = \sum_{n=1}^{N_j} w_{A,n} \cdot \varphi(\mathbf{x}_n, \mathbf{x})$$

$$f_{j,B}(\mathbf{x}) = \sum_{n=1}^{N_j} w_{B,n} \cdot \varphi(\mathbf{x}_n, \mathbf{x})$$

$$f_{j,C}(\mathbf{x}) = \sum_{n=1}^{N_j} w_{C,n} \cdot \varphi(\mathbf{x}_n, \mathbf{x})$$

$$Y_A(\mathbf{x}) = \sum_{j=1}^{L} h_{A,j} \cdot f_{j,A}(\mathbf{x})$$

$$Y_B(\mathbf{x}) = \sum_{j=1}^{L} h_{B,j} \cdot f_{j,B}(\mathbf{x})$$

$$Y_C(\mathbf{x}) = \sum_{j=1}^{L} h_{C,j} \cdot f_{j,C}(\mathbf{x})$$

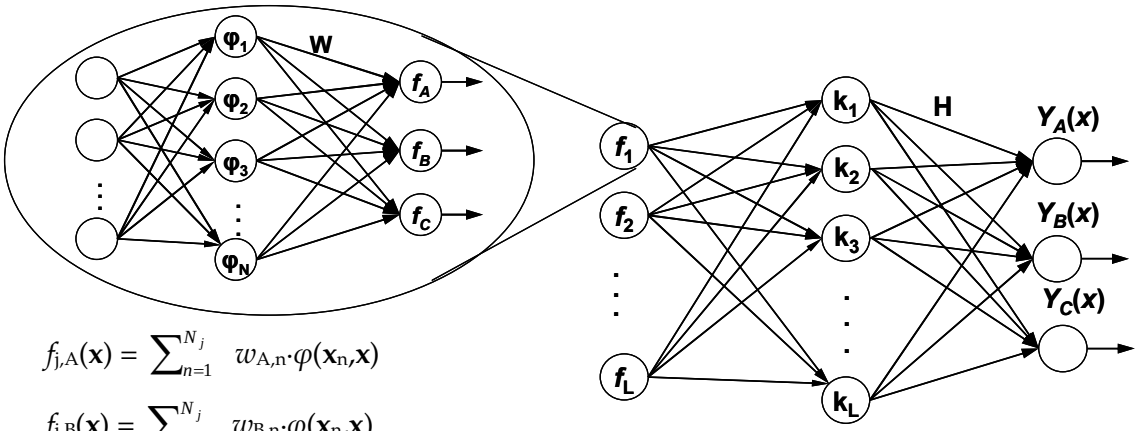
Figure 10.5. A committee machine with L neural network modules for the three-class problem.

The Committee machine [5][6][39][40] illustrated in fig. 10.5, combines multiple independently trained neural networks. Visually, a committee machine can be realized as a modular Neural Network which has L Neural Network modules in place of its neurons. When these Neural Network modules share the same input space and divide the training dataset among them into disjoint partitions (N_1, N_2,.., N_L) then they are situated in the hidden layer neurons of the committee. After finding the committee weights $\mathbf{H}$, in order to classifying an unknown $\mathbf{x}$ the committee machine class outputs $Y_m(\mathbf{x})$ are implemented as a weighted sum of the individual module outputs $f_{j,m}(\mathbf{x})$ for each class m (see also fig. 10.5). The maximum value of the committee class output wins. Such a committee machine having the neural network modules nested in the

hidden neurons is closely related with a two level HiMarkovRBFNN. The only difference here is that the higher level centers are missing and thus it is difficult to generalize into more than two levels.

The combining linear weights **H** of the Committee Machine are found by using multi-response regularized linear regression, which is class-conscious (different for each class that exploits the full decision profile). That is for L neural networks $f()$ in the leaf nodes and a training dataset of N examples a regression matrix $\mathbf{G}_m$ of size $L \times N$ is formed for each class m. The leaf nodes act as the regressors. For instance, an entry for the class A of the j^{th} regressor output and the i^{th} example $\mathbf{x}_i$ is $\mathbf{G}_A[i][j] = f_{j,A}(\mathbf{x}_i)$. Then the combining weights $\mathbf{h}_A$ are given in terms of the desired labels $\mathbf{y}$ (hot encoding) for class A by $\mathbf{h}_A = (\mathbf{G}_A{}^T \mathbf{G}_A + \lambda \mathbf{I})^{-1} \mathbf{G}_A{}^T \mathbf{y}$.

The Cascaded Machine presented in fig. 10.6 is the other standard model architecture. The Cascaded Machine uses the outputs of one level as inputs into the second level. The summation neurons at each level use weights to combine the outputs they receive from the previous level. As pointed out in [41], in the area of pattern recognition and machine learning the idea of viewing the predictor output as a new feature vector for another predictor is very old. This can be traced back to Sebestyen [32] in his book Decision-Making Processes in Pattern Recognition, published in 1962, where he proposes Cascade Machines, in which the output of one predictor is fed as the input to the next predictor in the sequence. We use an efficient implementation of a class-conscious Cascaded Machine (fig. 10.6).



$$f_{j,A}(\mathbf{x}) = \sum_{n=1}^{N_j} w_{A,n} \cdot \varphi(\mathbf{x}_n, \mathbf{x})$$

$$f_{j,B}(\mathbf{x}) = \sum_{n=1}^{N_j} w_{B,n} \cdot \varphi(\mathbf{x}_n, \mathbf{x})$$

$$f_{j,C}(\mathbf{x}) = \sum_{n=1}^{N_j} w_{C,n} \cdot \varphi(\mathbf{x}_n, \mathbf{x})$$

$$class(\mathbf{x}) = \arg \max_m (Y_m(\mathbf{x}))$$

$$Y_A(\mathbf{x}) = \sum_{j=1}^{L \times 3} h_{A,j} \cdot k_j(\mathbf{f}(\mathbf{x}))$$

$$Y_B(\mathbf{x}) = \sum_{j=1}^{L \times 3} h_{B,j} \cdot k_j(\mathbf{f}(\mathbf{x}))$$

$$Y_C(\mathbf{x}) = \sum_{j=1}^{L \times 3} h_{C,j} \cdot k_j(\mathbf{f}(\mathbf{x}))$$

$$\mathbf{f}(\mathbf{x}) = [f_{1,A}(\mathbf{x})\, f_{1,B}(\mathbf{x})\, f_{1,C}(\mathbf{x})\, f_{2,A}(\mathbf{x})\, f_{2,B}(\mathbf{x})\, f_{2,C}(\mathbf{x}) \ldots\ldots\ f_{L,A}(\mathbf{x})\, f_{L,B}(\mathbf{x})\, f_{L,C}(\mathbf{x})]^T$$

Figure 10.6. A class-conscious Cascaded Machine for the three class problem where all the class outputs of level-0 become inputs of the level-1. The vector $\mathbf{f}(\mathbf{x})$ is the new transformed input vector for the level-1 predictor.

The weights **H** for the Cascaded Machine are found via stacking. Stacked generalization [42], or stacking in sort, learns the combining weights by mapping between the leaf output predictions and the actual correct classes. According to

stacking for computing the weights of the L individual neural network modules (or level-0 models) used in the Cascaded Machine meta-learner combiner (or level-1 model), a global separate validation set must be employed. The examples in this validation set are mapped one by one to level-0 modules for creating the level-1 training set, the **A** set. Fig. 10.6 shows 3 classes. For these 3 classes each level-1 training sample in the **A** set has $L\times3$ attributes, whose values are the class output predictions of each one of the L level-0 modules. Therefore, a level-1 training instance in **A** set is made of $L\times3$ attributes and the desired target class y. See the new transformed input vector $\mathbf{f}(\mathbf{x})$ vector in fig. 10.6 for such a level-1 instance. This multi-response stacking is also class-conscious and exploits the full decision profile. We employ a linear kernel $k()$ in the hidden neurons of the level-1 to perform ridge regression. Then using the transformed level-1 set **A**, the weights $\mathbf{H}_m$ for every class m are obtained for the linear kernel in terms of the desired labels **y** for this class m (hot encoding is used which sets y=1 for class m examples and y=0 for the others) by solving the regularized system $\mathbf{H}_m=(\mathbf{A}^T\mathbf{A}+\lambda\mathbf{I})^{-1}\ \mathbf{A}^T\mathbf{y}$. In contrast to the poor performance of the simple cascading versions, where the level-0 models propagate to the level-1 only one output value (only the predicted label), the class-conscious Cascaded Machine in fig. 10.6 is a very efficient combiner.

10.8 Experimental simulations

Simulations are conducted for various benchmark datasets, and several comparisons have been made. The classification error rate is measured using publicly available benchmark datasets from the UCI machine learning data repository (http://archive.ics.uci.edu/ml). The specific details of these datasets are illustrated in table 10.1.

Table 10.1. Benchmark dataset details

Dataset Name	**instances**	**features**	**classes**
Wine	178	13	3
Wisconsin (Diagnostic)	683	9	2
Diabetes Pima Indians	768	8	2
Sonar	208	60	2
Glass	212	9	6
Vehicle Silhouettes	846	18	4
Yeast	1479	8	9
Spambase	4601	57	2
Page Blocks	5473	10	5
Satellite Image	6435	36	6

A dataset is randomly split into a training set (80%) and a test set (20%) with stratification. For each dataset we measure the classification error rate, which counts the number of incorrectly classified examples of the test set and divides by their population number. This procedure is repeated 40 times and the results are averaged.

The purpose of the experiments is to train many RBFNNs and merge them into one HiMarkovRBFNN. This higher level parent HiMarkovRBFNN will have many nested children. The simulations intend to reveal via the comparisons if the error rate of this parent is similar to the error rate produced by the other meta-learner combiners. This way, one can examine if the proposed method suffers from any accuracy losses. If the error rates are similar then we are practically done, since generalizing the HiMarkovRBFNN into many levels is straightforward, in contrast to the other methods.

In table 10.2 we compare the error rates on the test set by using a single Radial Basis Function Neural Network (full RBFNN), the Parent RBFNN only (parent RBFNN), the ensemble average of the child RBFNNs class outputs (Child RBFNNs), the Hierarchical Markovian RBFNN that combines the parent with their children, the Committee Machine of RBFNNs and the Cascaded Machine of RBFNNs.

Table 10.2. Classification error rates for the various methods

	Parent centers	Full RBF NN	Parent RBF NN	Child RBF NNs	Committee Machine of RBFNN	Cascaded Machine of RBFNN	**Hierarchical Markovian RBFNN**
Wine	5	2.8 ±2.2	3.7 ±2.4	2.7 ±2.4	2.4 ±1.9	**2.2** ±2.0	2.3 ±2.1
	10	2.6 ±2.3	3.7 ±2.4	2.5 ±2.2	2.2 ±1.9	2.3 ±1.8	**2.0** ±2.0
Wisconsin	5	4.1 ±1.8	4.2 ±1.4	4.5 ±1.8	4.0 ±2.0	4.5 ±2.1	**3.2** ±1.2
	10	4.3 ±1.5	4.3 ±1.4	5.4 ±2.6	4.4 ±1.8	5.2 ±2.1	**4.0** ±1.7
Diabetes	5	26.5 ±3.0	28.2 ±3.4	26.8 ±2.8	**25.6** ±2.9	25.7 ± 2.8	26.0 ±2.8
	10	26.4 ±3.0	28.3 ±2.8	26.5 ±3.3	26.1 ±3.2	**25.8** ±3.1	26.1 ±2.9
Sonar	5	19.3 ±5.3	34.5 ±6.7	19.6 ±5.3	19.5 ±4.3	19.5 ±4.3	**19.1** ±4.0
	10	18.7 ±5.6	32.4 ±5.4	18.5 ±4.8	17.8 ±4.5	17.9 ±4.5	**17.3** ±4.7
Glass	5	33.1 ±4.2	45.2 ±5.1	36.2 ±5.2	34.2 ±5.1	34.3 ±5.6	**32.9** ±4.6
	10	33.2 ±4.4	46.4 ±6.6	34.7 ±5.1	33.1 ±4.8	33.2 ±4.5	**32.8** ±4.9
Vehicle	5	29.2 ±2.8	43.5 ±4.8	29.7 ±2.8	28.9 ±2.8	29.2 ±2.6	**28.5** ±2.2
	10	29.4 ±3.0	41.6 ±4.5	29.6 ±2.8	28.9 ±2.7	28.9 ±2.9	**28.6** ±2.6
Yeast	10	43.2 ±1.8	44.9 ±2.7	43.5 ±2.8	42.4 ±2.0	42.3 ±2.1	**42.1** ±2.1
	20	43.1 ±2.0	44.7 ±2.9	43.3 ±2.9	41. 7 ±2.2	41.9 ±2.3	**41.3** ±2.1
Spambase	10	8.6 ±2.3	15.5 ±3.7	9.4 ±1.5	7.7 ±0.9	7.8 ±1.0	**7.5** ±0.6
	20	8.7 ±2.2	13.2 ±3.6	9.2 ±1.6	7.8 ±1.0	7.9 ±1.1	**7.4** ±0.9
Page blocks	10	4.2 ±0.4	12.8 ±0.9	5.5 ±0.7	**3.7** ±0.4	3.8 ±0.4	3.9 ±0.5
	20	4.1 ±0.5	11.2 ±1.0	6.2 ±0.9	3.7 ±0.5	3.7 ±0.5	**3.5** ±0.6

Satellite Image	20	12.6 ±0.8	16.3 ±1.1	12.1 ±0.9	10.5 ±0.9	10.8 ±0.9	**9.8** ±0.8
	40	12.5 ±1.0	15.1 ±1.2	11.5 ±1.2	11.2 ±0.9	11.3 ±0.9	**9.7** ±0.8

It is essential to keep the complexity of the experiments as simple as possible, without optimizing any parameters in order to reveal potential advantages and disadvantages of the proposed method. That is to create a default baseline. To this end, we set the regularization parameter equal to λ=0.1 for all implementations. The Gaussian width parameters of the RBF centers in the single full RBFNN as well as of those in all the RBFNN child neural networks were found from their k-nearest neighbours using k=10. For the width parameter in each one of the RBF centers in the parent RBFNN we use the minimum distance between all these parent centers $\mathbf{c}$ (which is much better than their average distance from the points they own inside their clusters). Similarly, for the HiMarkovRBFNN the Gaussian width parameters in the terms $\exp(-||\mathbf{c}-\mathbf{c}'||^2/\boldsymbol{\sigma}^2)$, which are the costs between a parent center $\mathbf{c}$ and a child center $\mathbf{c}'$, are equal to the minimum distance between the parent centers $\mathbf{c}$.

The k-means clustering algorithm is used to divide each dataset into K disjoint data clusters and to create a K population of centers for the Parent RBFNN. We use K equal to 5, 10, 20 or 40. For each cluster a child RBFNN is trained using the data inside it. Thus, one parent RBFNN has K centers and each such center corresponds to one child RBFNN. In the fourth column of table 10.2, the performance of the parent RBFNN is illustrated where only its RBF centers are used to classify the test set examples. In the fifth column there is the performance of the ensemble of the child RBFNNs. In the sixth column there is the performance of the HiMarkovRBFNN that combines the parent RBF centers with their related child RBFNNs and shows improvements in accuracy.

In essence the comparison results in table 10.2 reveal that the simple recursive operation is efficient and the hierarchy is present in all the datasets. The proposed method does not suffer from any accuracy losses and the HiMarkovRBFNN performs fairly better than all the other methods in the comparisons. Table 10.2 shows that the HiMarkovRBFNN ranks first in sixteen cases. However the standard deviations must also be considered. In particular, although the relative improvement is noticeable for almost all the datasets, the distributions of the errors are overlapping. This allows though considering positively the performance on the four cases in table 10.2 for which the other models perform better, because the difference is small considering the standard deviation of the error. Apart from the performance improvements, the main intention of the experiments is to demonstrate the effective functionality of the recursive response functions. This is the key that can permit many levels for the hierarchy while the other models stop in the two-level architecture by definition.

In particular the results in table 10.2 illustrate that the error rates of the HiMarkovRBFNN are smaller than those of the single RBF neural network in the third column. In addition, the comparison results in the seventh column and eighth column of table 10.2 actually show that the HiMarkovRBFNN is similar in accuracy to the Committee machine and the Cascaded Machine, which are very effective meta-level combiners, and in most of the datasets the HiMarkovRBFNN outperforms them. The difference is that the last two meta-level combining methods are limited to two levels, while the HiMarkovRBFNN produces a fully functional higher level RBFNN.

Parallel performance tests are also conducted on a cluster of workstations of Intel Pentium with a clock speed of 2.5 GHz, memory 2GB and Linux system. The machines are interconnected with 1000 Mbps Ethernet. We measure the total parallel execution time versus the number of processors and the speedup S / P, where S the sequential run time in a single processor and P the time that simulates the Network in parallel.

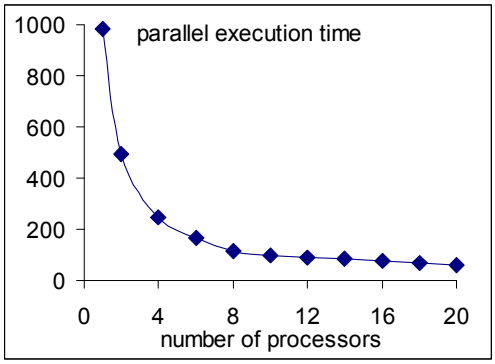


Figure 10.7. Total parallel execution time versus the number of processors

Fig. 10.7 illustrates the performance evaluation using the parallel execution time versus the number of processors for one artificial dataset with N=500000. Using 20 processors the parallel execution time was reduced by 94%.

In fig. 10.8 one can see the ratio of sequential execution time per parallel execution time using 20 processors when tested in variable size datasets. This ratio assists in the scalability analysis. By increasing the dataset size the parallel hierarchical Markovian RBFNN improves its scalability towards linear speedup.

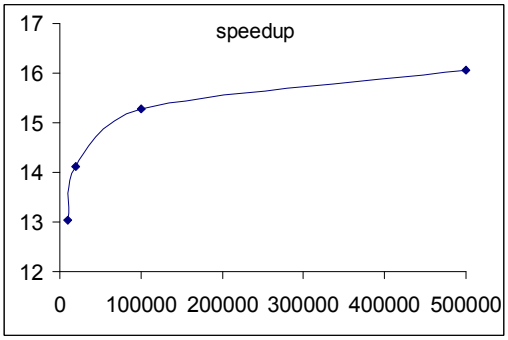


Figure 10.8. Speedup as measured on different datasets

The complexity in the first part of the training phase depends on the type of algorithm used for top-down RBF centers finding and the number of levels. The second part of the training is the bottom-up linear weights finding in every node and depends on the number of nodes.

10.9 Summary

The proposed HiMarkovRBFNN model tries to adapt a hierarchical modular nested neural network into any hierarchical structure of nested data clusters. Thus, the model has hierarchical levels of training. We employ the Markov rule in order to define a recursive RBF response function that permits the hierarchical integration from one level to the next. All the nodes in the hierarchical levels are composed of fully functional RBF in nature Neural Networks which have the two classical synaptic weight sets, namely the **C** matrix that holds their RBF centers and the **W** matrix that

holds their linear output weights. Thus, the Neural Network operation is exactly the same for all nodes. The discussion section shows that when the Markov property is used in the summations for the joint probability distributions of the traditional single RBFNN it produces the same equation for the linear output weights **W**. Thus, the foundations of this idea are actually quite old. How simple can be the constructing phase of the HiMarkovRBFNN is demonstrated by using three textbook algorithms, namely the well-known k-means clustering, the classical tree-based recursion function, and a standard regularized nonnegative least squares solver. This serves as a baseline, which also means that there is room for improvements.

Experimental simulations are conducted in order to compare the HiMarkovRBFNN with a single RBFNN, an ensemble of the child RBFNNs, and the two standard model meta-learning architectures, namely Committee Machines and Cascaded Machines. The results show that the HiMarkovRBFNN has similar error rate with the other meta-learning combiners and it doesn't suffer from accuracy losses due to the Markov rule. Hence, the proposed nested merging method produces a promising fully functional higher level HiMarkovRBFNN node.

Such a hierarchical structure of the HiMarkovRBFNN can potentially become practical in applying it together with a divide-and-conquer strategy to overcome common scalability issues and facilitate hierarchical learning. In addition, within the context of rule extraction in data mining, it is often desirable to convert the interconnection weights into rules to make the discovered knowledge comprehensible for the users. It is possible that a hierarchical neural network which adapts into a given structure of nested clusters could be of assistance in hierarchical rule extraction. Beyond the scalability advantages and the obvious parallel operation, the presented method could be proven useful in classifying highly irregular datasets, such as complex nested datasets. We plan on studying this possibility together with improvements on the training phase, as well as on modifying the proposed hierarchical nested structure that supports a recursion in order to work with other types of neural networks.

References for chapter 10

[1] Wilson G. (1995) Parallel Programming for Scientists and Engineers. MIT Press, Cambridge

[2] Dietterich T.G. (2000) The Divide-and-Conquer Manifesto. Algorithmic Learning Theory, 13–26.

[3] Moody J.E. (1988) Fast learning in multi-resolution hierarchies. Neural Information Processing Systems, 29–39.

[4] Bishop C.M. (1991) Improving the generalization properties of radial basis function neural networks. Neural Computation 3, 579–588.

[5] Bishop C.M. (1995) Neural Networks for Pattern Recognition. New York: Oxford Univ. Press.

[6] Hu Y.H. and Hwang J.N. (2002) Handbook of neural network signal processing. CRC Press LLC.

[7] Er M.J., Wu S., Lu J. and Toh H.L. (2002) Face recognition with Radial Basis Function (RBF) Neural Networks. IEEE Transactions on Neural Networks, 13(3), 697–710.

[8] Sarimveis H., Alexandridis A. and Bafas G. (2003) A fast training algorithm for RBF networks based on subtractive clustering. Neurocomputing 51, 501–505.

[9] Karayiannis N. and Randolph-Gips M. (2003) On the Construction and Training of Reformulated Radial Basis Function Neural Networks. IEEE Transactions on Neural Networks 14(4), 835–844.

[10] Han M. and Xi J. (2004) Efficient clustering of radial basis perceptron neural network for pattern recognition. Pattern Recognition 37, 2059–2067.

[11] Rivas V.M., Merelo J.J., Castillo P.A., Arenas M.G. and Castellano J.G. (2004) Evolving RBF neural networks for time-series forecasting with EvRBF. Information Sciences 165, 207–220.

[12] Wang L. and Fu X. (2005) Data Mining with Computational Intelligence. Springer-Verlag

[13] Oyang Y.J., Hwang S.C., Ou Y.Y., Chen C.Y. and Chen Z.W. (2005) Data classification with radial basis function networks based on a novel kernel density estimation algorithm. IEEE Transactions on Neural Networks 16, 225–236.

[14] Alexandridis A., Chondrodima E. and Sarimveis H. (2013) Radial Basis Function Network training using a nonsymmetric partition of the input Space and Particle Swarm Optimization. IEEE Transactions on Neural Networks and Learning Systems, 24 (2), 219–230.

[15] Fernández-Navarro F., Hervás-Martínez C. and Gutierrez P. A. (2013) Generalised Gaussian radial basis function neural networks. Soft Computing, 17 (3), 519–533.

[16] Babu G.S. and Suresh S. (2013) Sequential Projection-Based Metacognitive Learning in a Radial Basis Function Network for Classification Problems. IEEE Transactions on Neural Networks and Learning Systems 24(2), 194–206.

[17] Yu H., Reiner P.D., Xie T., Bartczak T. and Wilamowski B.M. (2014) An incremental design of Radial Basis Function Networks. IEEE Transactions on Neural Networks and Learning Systems 25(10), 1793–1803.

[18] Hong X., Chen S., Qatawneh A., Daqrouq K., Sheikh M. and Morfeq A. (2014) A radial basis function network classifier to maximise leave-one-out mutual information. Applied Soft Computing 23, 9–18.

[19] Oh S.-K., Kim W.-D., Pedrycz W. and Seo K. (2014) Fuzzy Radial Basis Function Neural Networks with information granulation and its parallel genetic optimization. Fuzzy Sets and Systems 237, 96–117.

[20] Borghese N.A. and Ferrari S. (1998) Hierarchical RBF networks and local parameters estimate. Neurocomputing 19, 259–283.

[21] Ferrari S., Maggioni M. and Borghese N.A. (2004) Multi-scale approximation with hierarchical radial basis functions networks. IEEE Transactions on Neural Networks 15(1), 178–188.

[22] Ferrari S., Frosio I., Piuri V. and Borghese N.A. (2005) Automatic multiscale meshing through HRBF networks. IEEE Trans. on Instr. and Meas., 54(4), 1463–1470.

[23] Ferrari S., Bellocchio F., Piuri V. and Borghese N.A. (2010) A hierarchical RBF online learning algorithm for real-time 3-D scanner. IEEE Transactions on Neural Networks 21 (2), 275–285.

[24] Mat Isa N.A., Mashor M.Y. and Othman N.H. (2002) Diagnosis of Cervical Cancer using Hierarchical Radial Basis Function (HiRBF) Network. In Proceedings of the International Conference on Artificial Intelligence in Engineering and Technology, 458–463

[25] Van Ha K. (1998) Hierarchical radial basis function networks. In IEEE Neural Network Proceedings, 1893–1898.

[26] Chen Y., Yang B., Dong J. and Abraham A. (2005) Time-series forecasting using flexible neural tree model. Information Sciences, 174(3-4), 219–235.

[27] Chen Y., Peng L. and Abraham A. (2006) Hierarchical Radial Basis Function Neural Networks for Classification Problems. In Proceedings of ISNN (1), 873–879.

[28] Chen Y., Yang B. and Meng Q. (2012) Small-time scale network traffic prediction based on flexible neural tree. Applied Soft Computing, 12(1), 274–279.

[29] Jordan M.I. and Jacobs R.A. (1994) Hierarchical mixtures of experts and the EM algorithm. Neural Computation 6, 181–214.

[30] Kokkinos Y. and Margaritis K.G. (2013) A Parallel and Hierarchical Markovian RBF Neural Network: preliminary performance evaluation. In Iliadis L. et al. (eds.), Proceedings of EANN 2013, Springer/LNCS, Part I, CCIS 383, 340–349.

[31] Clune J., Mouret J.-B. and Lipson H. (2013) The evolutionary origins of modularity. Proceedings of the Royal Society B. 280: 20122863, 1–9.

[32] Sebestyen G.S., (1962) Decision-Making Process in Pattern Recognition. The Macmillan Company, N.Y.

[33] Yu L., Lai K.K and Wang S. (2008) Multistage RBF neural network ensemble learning for exchange rates forecasting. Neurocomputing 71, 3295– 3302.

[34] Fukushima K. (1988) Neocognitron: A Hierarchical neural network capable of visual pattern recognition. Neural Network 1, 119-130.

[35] Behnke S. (2003) Hierarchical Neural Networks for Image Interpretation. PhD thesis, in Springer LNCS 2766.

[36] Dittenbach M. (2000) The Growing Hierarchical SOM (GHSOM). In Proceedings of IJCNN.

[37] Tay A.L.P., Zurada J.M., Wong L. and Xu J. (2007) The hierarchical fast learning artificial neural network (HieFLANN) – An autonomous platform for hierarchical neural network construction. IEEE Transactions on Neural Networks 18(6), 1645–1657.

[38] Poggio T. and Girosi F. (1990) Regularization algorithms for learning that are equivalent to multilayer networks. Science 247, 978–982.

[39] Hashem S. (1997) Optimal linear combinations of Neural Networks. Neural Networks 10(4) 599–614.

[40] Breiman L. (1999) Combining predictors, in A.J.C. Sharkey (eds) Combining artificial neural nets: ensemble and modular multinet systems. Springer, Berlin Heidelberg New York, 31–50.

[41] Kuncheva L.I. (2004) Combining Pattern Classifiers: Methods and Algorithms. John Wiley & Sons.

[42] Wolpert D.H. (1992) Stacked generalization. Neural Networks 5(2), 241–259.

11 Local Learning Regularization Networks for localized regression

Local learning algorithms use a neighbourhood of training data close to a given testing query point in order to learn the local parameters and create on-the-fly a local model specifically designed for this query point. The local approach delivers breakthrough performance in many application domains. This chapter considers local learning versions of Regularization Networks (RN) and investigates several options for improving their online prediction performance, both in accuracy and speed.

First we exploit the interplay between locally optimized and globally optimized hyper-parameters (regularization parameter and kernel width) each new predictor needs to optimize online. There is a substantial reduction of the operation cost in the case we use two globally optimized hyper-parameters that are common to all local models. We also demonstrate that this global optimization of the two hyper-parameters produces more accurate models than the other cases that locally optimize online either the regularization parameter, or the kernel width, or both.

Then by comparing Eigenvalue decomposition (EVD) with Cholesky decomposition specifically for the local learning training and testing phases, we also reveal that the Cholesky based implementations are faster that their EVD counterparts for all the training cases. While EVD is suitable for validating cost–effectively several regularization parameters, Cholesky should be preferred when validating several neighbourhood sizes (the number of k-nearest neighbours) as well as when the local network operates online.

Then we exploit parallelism in a multi-core system for these local computations demonstrating that the execution times are further reduced. Finally, although the use of pre-computed stored local models instead of the online learning local models is even faster, this option deteriorates the performance. Apparently, there is a substantial gain in waiting for a testing point to arrive before building a local model, and hence the online local learning RNs are more accurate than their pre-computed stored local models. To support all these findings we also present extensive experimental results and comparisons on several benchmark datasets.

11.1 Introduction

Owning to their prediction capabilities, neural networks have been proven powerful tools for regression tasks and forecasting. Traditional supervised learning with neural networks [1] has a training phase which uses the whole training dataset to create the one global learning model that is the most capable for predicting all the unseen data. If the input space has many different regions of variable complexity the construction of a single global model suitable for all of them, typically can pose some difficulties. One of the potential alternatives is to localize the algorithm. In contrast to global learning algorithms, a local learning algorithm [6] predicts a testing point based only on a local training data neighbourhood close to the testing point. We study localized learning versions of regularization networks [1-4].

Regularization is one common strategy that is performed during training, in order to render the global neural model to generalize well on new examples. Regularization Networks [2][3][4] are supervised learning neural networks [1] with one hidden layer and one output layer, designed for reconstructing input-output mappings. They have very good theoretical background, simple training and belong to kernel methods. The global learning algorithm of the Regularization Network uses all the training data points to form the kernel functions, and creates one global model that serves as a predictor. Using the real points in the kernels is valuable when data features have discrete values, like in cases of image processing, pattern recognition, computer vision and data mining, a fact that elevates such types of kernel methods [5] to state-of-the-art in modern machine learning.

Local learning algorithms emerge recently as an alternative to the global learning strategy, and quickly adopted for localized predictions. In a seminal paper, Bottou and Vapnik [6] proposed the local learning procedure which uses neighbourhoods to learn the model parameters online and create the model at runtime after the testing point is known. Hence, for each new testing point, a local learning algorithm builds at runtime a different new local learning model, by using only a local list of the k nearest training points, or the training points lying inside a user defined region, of which the centre is the current testing point. This approach delivers very effective performance in many application domains and inspires a variety of algorithms [7-26] (see also section 2.1) that are based on local neighbourhoods.

Still, a local learning algorithm as formulated in [6] is a lazy learner that relies on locality principles and waits until an unknown example appears. The on-the-fly model creation needs to create from scratch a new predictor for each new example which brings an additional overhead and sometimes the computational cost of operation is significant. The optimization of a specific model involves trying multiple networks with different parameters in order to achieve acceptable model accuracy. The execution time per example can vary from several seconds to minutes. This renders many algorithms like those that depend on regularized kernel methods expensive to apply them as local learners.

In this work we investigate four options for improving the online performance (accuracy and speed) of the local learning regularization networks. By exploiting the interplay between locally optimized and globally optimized parameters and thus reducing the hyper-parameters each new predictor needs to optimize online, by using efficient matrix decompositions, by using parallelism in the local computations, and by pre-compute and store the local models we demonstrate viable ways for the network

operation. Some preliminary results of the first two options have been also reported in [26]. In this work we present a thorough investigation of all the four options and describe their detailed algorithms for all the different optimization cases of the local learning regularization networks.

Each local regularization network model has two hyper-parameters which are the regularization parameter λ and the Gaussian width σ of the kernel functions. Local learning algorithms assume that the network weights differ among different local regions of the training set which defined for the different query points. Still even if the weights differ for each region, the local hyper-parameters of the models can be locally optimized for each region on-the-fly, or globally optimized over the entire dataset. By comparing the possible cases for the hyper-parameters $\{\sigma, \lambda\}$ we find that the case in which these local hyper-parameters are globally optimized (a unique set of them is used during network operation), then the local learning regularization network outperforms every other locally optimized case, and it is also much faster.

The chapter is organized as follows. Section 2 surveys related work local learning algorithms and presents the basic concepts of regularization networks. Section 3 introduces the local regularization network training and operation, describes the virtual leave-one-out cross validation strategy for model selection of the hyper-parameters, presents the methods for reducing the operation cost of the local models, describes the proposed training and testing algorithms used in the comparisons and outlines the basic principles of the multi-core implementations. Section 4 provides experimental results for the accuracy and timing of the four different cases of training and testing the local learning regularization networks and gives additional timing results. Section 5 summarizes our conclusions.

In the following, a matrix will be symbolised with a bold capital letter, a vector with a bold lowercase letter and a scalar variable with an italic letter. Fig. 11.1 illustrates the meaning of common symbols and notations used in this chapter.

$\mathbf{x}_n$ – a n^{th} training example
y_n – the target value corresponding to $\mathbf{x}_n$ training example
$\{\mathbf{x}_n, y_n\}_{n=1}^{N}$ – the whole training set
N – the number of training examples
k – number of nearest neighbours
λ – regularization parameter
σ – kernel function width
$\mathbf{K}$ – kernel matrix
$\mathbf{w}$ – weights vector
$f_k(\mathbf{q})$ – local model for each query testing point $\mathbf{q}$
k-NN($\mathbf{q}$) – list of k-nearest neighbour training points $\mathbf{x}_n$ closer to $\mathbf{q}$

Figure 11.1. Common symbols and notations

11.2 Relevant Material

11.2.1 Related work in Local Learning algorithms

After the local learning idea introduced in [6] it has been adapted for various tasks. Local algorithms for dependencies estimation and pattern recognition have been studied in [7] through the view of local structural risk minimization. A local learning neural network studied in [8] explores the concept of sequential learning and the effectiveness of global and local neural network learning algorithms on a sequential learning task. The Locally adaptive subspace regression method proposed in [9] localizes the linear regressions to approximates nonlinear functions by means of piecewise linear models and thus works successfully in high dimensional spaces. Hence, these local projections, that employ local dimensionality reduction by principal component regression, can be used to accomplish local function approximation in the neighborhood of a given query point. In this spirit the Locally Weighted Projection Regression proposed in [10] is an algorithm that achieves nonlinear function approximation in high dimensional spaces with redundant and irrelevant input dimensions.

For classification tasks the label of each point can be predicted by the data points in its neighborhood. The Transductive Classification [11] formulation, which learns from both labeled and unlabeled data, has been studied via Local Learning Regularization in [12]. The Dimensionality reduction via Local Learning Projections which was studied in [13] reveals that the projection of a point can be well estimated based on its neighbors in the same class. Unlike principal component analysis that minimizes a global estimation error the Local Learning Projection minimizes a local estimation error that additionally uses the class labels.

Local Support Vector Machine (SVM) classifiers have also been considered for localized classification tasks. Two such local SVM-KNN classifiers where independently proposed in [14] for visual recognition tasks and in [15] for remote sensing images [16] in which a local bound on the generalization error is also derived. Another local SVM classifier approach is detailed in [17] for noise reduction applications. In a variant of SVM-KNN presented in [18] the computational complexity is reduced by pre-computed local SVM, that are not calculated by using the *k*-nearest neighbors of the testing point, but on the k-nearest neighbors of the certain training point presumably closest to the testing point. The main idea of local SVM-KNN classifier is to build an example-specific maximal marginal hyperplane based on the set of k-neighbours. An analogous approach of adaptive Local hyperplane classification is applied for face recognition [19]. A different localized SVM is presented in [20] that instead of the *k*-nearest neighbors it uses the fixed area (radii) of the neighborhood data around the testing point.

A theoretical study of consistency and localizability presented in [21] for fixed areas of neighbourhoods, reveal that every consistent learning algorithm is asymptotically acting like a local one. A theoretical proof for the universal consistency of localized versions of regularized kernel methods that use *k*-nearest neighbors was described in [22] which shows how a large class of regularized kernel methods can be localized in order to get a universally consistent learning algorithm.

From the view of Bayesian confidence the work in [23] has shown graphically how a local learning Probabilistic Neural Network classifier that uses few *k*-nearest neighbour

neurons can maintain the confidence ratio of the correctly classified samples and by intrinsically optimizing the number of *k*-nearest neighbours can reduce substantially the operation cost for the larger datasets. For image classification tasks the work in [24] considers local learning linear SVM and further proposes to use large-scale multi-label classification based on Bayesian compressed sensing to predict the set of good neighbourhood training data close to the particular testing point, by estimating the composition and the size of the local training subset that is likely to yield an accurate local model on-the-fly. For image retrieval, a local Extreme Learning Machine classification algorithm to extract the contour-based shape features of objects is presented in [25]. This model finds nearest neighbours of the testing set from the original training samples and by reconstructing the new training set it trains a local classification model. The extracted shape features of the images have rotation, scaling and translation invariance and can distinguish the class of each object without influenced by noise, or shape distortion.

11.2.2 Regularization Network basics

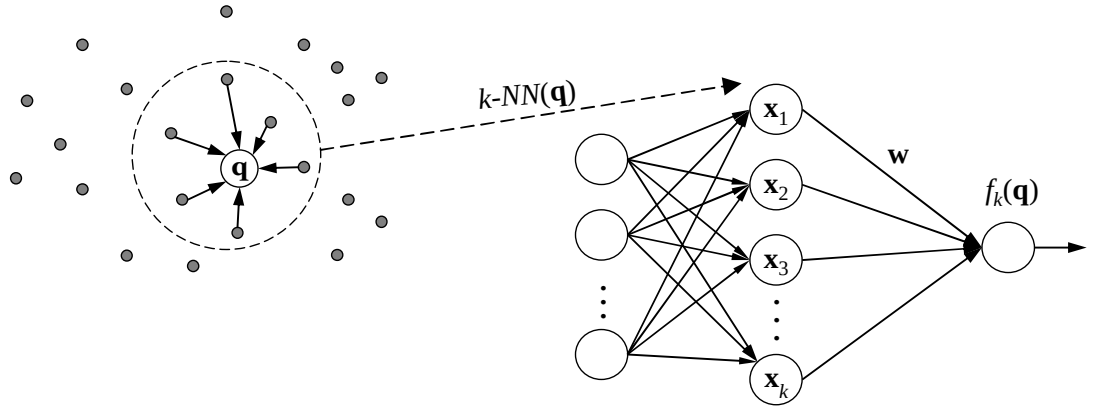
The Regularization Network algorithm was proposed in [2] where Poggio and Girosi have shown that for the problem of reconstructing a real function of several variables from a finite number of measurements the regularized solution is a neural network, called Regularization Network (RN). For typical kernel machines [3][4] [27], the basic Tikhonov regularization idea is to stabilize the, generally ill-posed solution, by means of some auxiliary nonnegative function that embeds prior information about the solution. The most common form of prior information is the assumption that the mapping function is smooth. This means that two similar inputs correspond to two similar outputs and that the function does not oscillate much. The algorithm becomes stable with respect to small changes of the data which implies that if one perturbs the data a little or a training point is removed, the algorithm must not be influenced at all. The regularized solution is found by minimizing on a Reproducing kernel Hilbert space (RKHS), associated with the kernel function, a regularized functional that contains the usual square loss data term and the regularization term which is the stabilizer representing the a-priori knowledge that penalizes unlikely non-smooth solutions.

A typical Regularization Network [1-4] has an input layer, a hidden layer, and an output layer. All the points of the training set $\{\mathbf{x}_n, y_n\}_{n=1}^{N}$ are loaded into the hidden layer to form the kernel functions. The global model $f(\mathbf{x}) = \sum_n^N w_n k(\mathbf{x}, \mathbf{x}_n)$ is the linear weighted combination of these kernel functions [2][3][4]. The Regularization Network training phase computes the kernel matrix $\mathbf{K}$, and finds the weights $\mathbf{w}$ by direct solving the system $(\mathbf{K}+\lambda\mathbf{I})\mathbf{w} = \mathbf{y}$. The process of how to optimize the regularization parameter λ is an issue. The most efficient and commonly used strategy is to perform an Eigenvalue decomposition on the matrix $\mathbf{K} = \mathbf{Q}\mathbf{D}\mathbf{Q}^{\mathrm{T}}$. Then, several candidate λ values can be tested at a minimal cost since $(\mathbf{K} + \lambda\mathbf{I})^{-1} = \mathbf{Q}\,(\mathbf{D} + \lambda\mathbf{I})^{-1}\,\mathbf{Q}^{\mathrm{T}}$ where only the trivial inversion of the diagonal is required. In this way one can avoid several matrix inversions and use instead the reusable matrices of eigenvalues $\mathbf{Q}$ and eigenvectors $\mathbf{D}$.

11.3 Local Learning Regularization Networks

11.3.1 Local learning regularization network

Given a testing query point **q** a local learning algorithm [6] builds on the fly a local model $f_k(\mathbf{q})$ which is trained with the k-NN(**q**) list of nearest training points closest to **q**. Fig. 11.2 illustrates a local learning regularization network topology and operation.



Operation: For a query point **q**
I) Select a local region k-NN(**q**) of k closest to **q** training examples $\mathbf{x}_i$
II) Train a regularization network $f_k(\mathbf{q})$ with only these examples in the k-NN(**q**) list
III) Apply the resulting regularization network for predicting the output $f_k(\mathbf{q})$

Figure 11.2. A local learning regularization network uses the k-nearest neighbours of the query point **q** as centers for the kernel functions.

Hence, a different local learning RN model is created for every different **q** by using its k-NN(**q**) list which defines the local training set $\{\mathbf{x}_i, y_i\}_{i=1}^{k}$. All $\mathbf{x}_i$ points in this list will become the centers of the kernel functions. As kernel functions, many strictly positive radially symmetric functions can be employed for which the Gaussian kernel is commonly used and their linear weighted sum will give the output of the local RN model $f_k(\mathbf{q})$ as:

$$f_k(\mathbf{q}) = \sum_{i=1}^{k} w_i \exp(-\|\mathbf{q} - \mathbf{x}_i\|^2/\sigma^2) \tag{11.1}$$

where k is the number of the nearest neighbours of **q**, and w_i are the weights of the kernel functions. Given the k-NN(**q**) list, of nearest to **q** training data, the weights are found by solving, in Reproducing Kernel Hilbert Space H_K, the minimization problem (eq. 11.2) for the regularized functional:

$$\arg\min_{f \in H_k} \left\{ \frac{1}{k} \sum_{i=1}^{k} (y_i - f_k(\mathbf{x}_i))^2 + \frac{\lambda}{k} \|f_k\|^2 \right\} \tag{11.2}$$

Minimizing eq. 11.2 the weight vector **w** is given by:

$$\mathbf{w} = (\mathbf{K} + \lambda \mathbf{I})^{-1} \mathbf{y} \tag{11.3}$$

where **K** is the local kernel matrix with entries $\mathbf{K}_{i,j} = \exp(-\|\mathbf{x}_i - \mathbf{x}_j\|^2/\sigma^2)$, **I** is the identity matrix, λ is the regularization parameter and $\mathbf{y} = [y_1, y_2, \ldots, y_k]^T$ is the target vector. The local kernel matrix **K** has size $k \times k$ and it is computed for each different **q** from the training points in the k-NN(**q**) list.

11.3.2 Selecting the hyper-parameters

Each RN model $f_k(\mathbf{q})$ has two hyper-parameters which are the regularization parameter λ and the Gaussian kernel width σ. In general, a very small width σ may drive the solution much closer to the training data and may loose information. This data closeness usually makes the Regularization Network to loose its generalization ability, of efficiently recognizing new unseen examples. The regularization parameter λ makes the solution more robust and improves the generalization performance by determining to what extend the complexity of a predictor is penalized (in order to avoid over-fitting). However a very large λ may considerably disturb the information and structure of the regularized kernel matrix, while a very small λ may not be as effective in improving the generalization performance. Therefore, the two hyper-parameters $\{\sigma, \lambda\}$ control the trade-off between data closeness and generalization. These parameters must be optimized during the model selection phase via cross-validation, which validates several candidate pairs $\{\sigma, \lambda\}$ and selects the best pair $\{\sigma_{best}, \lambda_{best}\}$ that has the minimum cross-validation error on the local training examples.

In leave one-out cross-validation each partition consists of a single pattern. Leave one-out cross-validation gives an almost unbiased estimate for the error. Suppose we drop the i^{th} example $\mathbf{x}_i$ from the training set and then retrain the learning model $f_{(-i)}()$. With the model $f_{(-i)}()$ constructed with the i^{th} example $\mathbf{x}_i$ omitted we predict the output $f_{(-i)}(\mathbf{x}_i)$. Then for each $\mathbf{x}_i$ the leave-one-out residual error is $e_{(-i)}(\mathbf{x}_i) = (y_n - f_{(-i)}(\mathbf{x}_i))$. This procedure is repeated for each one of the training examples, which obviously is very time-consuming.

The virtual leave-one-out cross-validation squared error E_{loo} is used instead, which is the most efficient way for computing the cross-validated error when a kernel matrix $\mathbf{K}$ is involved in the training. Although each model is different, they are similar since their datasets only differ by a single point. Hence, it has been shown [27] [28] [29] [30] that actually the residual $e_{(-i)}(\mathbf{x}_i)$ can be given by the ordinary residual $e(\mathbf{x}_i)$, which is the one that is found when the model is trained on all examples including $\mathbf{x}_i$, divided by 1 minus the i-th diagonal element of the hat matrix $\mathbf{K}(\mathbf{K}+\lambda\mathbf{I})^{-1}$ (for the hat matrix see details in [27] [28] [29] [30]), which after some simplifications [30] produces the virtual leave-one-out cross-validation squared error as:

$$E_{loo} = \frac{1}{k}\sum_{i=1}^{k} e_{(-i)}^2 = \frac{1}{k}\sum_{i=1}^{k} (w_i / g_{ii})^2 \tag{11.4}$$

where for each $\mathbf{x}_i$ in the training set $\{\mathbf{x}_i, y_i\}_{i=1}^{k}$ the residual $e_{(-i)}$ is given in terms of the weight w_i divided by g_{jj} which is the diagonal element of the matrix $\mathbf{G} = (\mathbf{K} + \lambda\mathbf{I})^{-1}$. For model selection in regularized kernel based methods the virtual leave-one-out squared error estimator in eq. 11.4 is the fastest known strategy for cross-validation and has been used in several works (see also [28][29] [30]).

11.3.3 Reducing the operation cost of the local models

While local learning can reduce the training cost, the operation cost can become larger. Learning on-the-fly the best local RN model for each testing point $\mathbf{q}$ requires finding the best pair of hyper-parameters $\{\sigma, \lambda\}$ that minimizes the local error E_{loo}. This strategy has a large cost, since by validating say 20 values for λ and 20 values for σ, the local learning testing phase must build on-the-fly 400 different local models for each testing point. The best number k of neighbours is also another global parameter that must be

determined during training. One could improve the online performance (accuracy and speed) by:

- exploiting the interplay between locally optimized and globally optimized parameters, in order to reduce the optimization time of the local hyper-parameters each new predictor needs during the online testing phase,
- computing the virtual leave-one-out error by using efficient matrix decompositions (Eigen decomposition or Cholesky) for the off-line training and on-line testing phases,
- using an approximation via pre-computed and stored representative local models from each training point, and then using the closest to **q** model during operation,
- using parallel computations for the local models.

We study all these options, and compare their accuracy and speed.

11.3.4 Interplay between locally optimized and globally optimized parameters

A reduction of the hyper-parameters each new predictor needs to optimize locally online can improve the online operation time.

An interesting interplay exists between locally optimized parameters and globally optimized parameters. While a different locally optimized parameter must be found during operation when a new testing point arrives, a globally optimized parameter is optimized only during the training phase, by using all the training data points, and then it is used during operation for all the testing points. There is a substantial reduction of the operation cost when using globally optimized parameters, either σ or λ, or both.

Table 11.1 shows the possible combinations of the hyper-parameters needed to be optimized from each local regularization network (RN) case. While Case 1 optimizes locally the pair $\{\sigma, \lambda\}$ there are other cases to consider also. Case 2 optimizes locally only the regularization parameter λ and uses a single globally optimized width σ. Then, case 3 optimizes a local σ for each model and uses a single global λ for all models, and case 4 exclusively uses a single global λ and a single global σ.

Table 11.1. The hyper-parameters needed to be optimized for each algorithm

Algorithm	Parameters to be optimized during training (off-line) for all training points	Parameters to be optimized during testing (on-line) for each testing point
Local RN case 1	global k	local σ, local λ
Local RN case 2	global k, global σ	local λ
Local RN case 3	global k , global λ	local σ
Local RN case 4	global k , global λ, global σ	

We explore all these cases in the experimental section, and we surprisingly found that case 4 is the fastest one and this case is also the most accurate.

Therefore, during on-line operation and assuming that the model selection needs to validate Λ candidate values for λ and Σ candidate values for σ, then an optimized local RN predictor needs to build on-the-fly $\Sigma \cdot \Lambda$ candidate local models when it must locally optimize both $\{\sigma, \lambda\}$ in case 1, Λ candidate local models when it must locally optimize λ, in case 2, Σ candidate local models when it locally optimizes σ, in case 3 and only one model in case 4.

11.3.5 Computing the virtual leave-one-out squared error

Fast computations are essential for the virtual leave-one-out cross validation squared error, whose solution can be computed in closed form during solving for the regularization network weights at practically minimal additional cost. To this end, one can resort to matrix decompositions like Eigenvalue Decomposition (EVD) or Cholesky Decomposition which have been studied in kernel methods [27] [28] [29] [30].

The re-usable matrices those decompositions produce is the key. In essence we will see in the next paragraphs that for the local regularization network EVD is more suitable when validating several candidate λ values with a fixed k, while Cholesky is preferred during validation of several k candidate values with a fixed λ. Thus, although for a conventional single RN training algorithm with the full dataset the Eigenvalue decomposition was the fastest known method of choice so far, we cannot just transfer this method 'as is' into the local learning models, for which after a thorough investigation we have surprisingly found that Cholesky is the fastest.

11.3.6 Using EigenValue Decomposition (EVD)

We have already seen in section 2.2 that Eigenvalue decomposition [31] [32] is the preferred method for finding the weights of a single regularization network by using the full dataset. The local learning models use k neighbor training points. Then, given the symmetric squared kernel matrix $\mathbf{K}^{k\times k}$ (with k rows and k columns) the solution is provided without actually calculating the inverse, by means of the Eigenvalue Decomposition of $\mathbf{K} = \mathbf{Q}\mathbf{D}\mathbf{Q}^T$, where $\mathbf{Q}^{k\times k}$ is the column orthonormal matrix containing the eigenvectors in its columns, and $\mathbf{D}$ is the diagonal matrix that contains the Eigenvalues. The code for EVD can be found in [32]. Since $\mathbf{K}$ is symmetric (i.e. $\mathbf{K}=\mathbf{K}^T$) it holds that $\mathbf{Q}^{-1}=\mathbf{Q}^T$, $\mathbf{K}^{-1}=\mathbf{Q}\mathbf{D}^{-1}\mathbf{Q}^T$ and $\mathbf{Q}^T\mathbf{Q}=\mathbf{I}=\mathbf{Q}\mathbf{Q}^T$. Hence we can validate cost-effectively several values of λ by using $(\mathbf{K} + \lambda\mathbf{I}) = \mathbf{Q}\mathbf{D}\mathbf{Q}^T + \lambda\mathbf{I} = \mathbf{Q}(\mathbf{D}+\lambda\mathbf{I})\mathbf{Q}^T$.

The inverse $\mathbf{G} = (\mathbf{K} + \lambda\mathbf{I})^{-1}$ is defined in terms of the reusable matrices $\mathbf{Q}$ and $\mathbf{D}$ by:

$$\mathbf{G} \equiv (\mathbf{K} + \lambda\mathbf{I})^{-1} = (\mathbf{Q}^T)^{-1}\,(\mathbf{D} + \lambda\mathbf{I})^{-1}\,\mathbf{Q}^{-1} = \mathbf{Q}\,(\mathbf{D} + \lambda\mathbf{I})^{-1}\,\mathbf{Q}^T \tag{11.5}$$

Then several candidate λ values can be tested at a minimal cost.

The weight vector $\mathbf{w}$ can be obtained as:

$$\mathbf{w} = (\mathbf{K} + \lambda\mathbf{I})^{-1}\mathbf{y} = \mathbf{Q}(\mathbf{D}+\lambda\mathbf{I})^{-1}\mathbf{Q}^T\,\mathbf{y} \tag{11.6}$$

Note that a i^{th} element of the weight vector $\mathbf{w}$ is given by:

$$w_i = \sum_j^k \; q_{ij}\,[\mathbf{Q}^T\mathbf{y}]_j \,/\, (d_{jj}+\lambda) \tag{11.7}$$

where q_{ij} is an element in the i^{th} row and j^{th} column of the matrix $\mathbf{Q}$, d_{jj} denotes the j^{th} diagonal element of the Eigenvalue matrix $\mathbf{D}$, and $[\mathbf{Q}^T\mathbf{y}]_j$ is the j^{th} element of the vector $[\mathbf{Q}^T\mathbf{y}]$. This shows that the cost for the weights $\mathbf{w}$ needs only k^2 operations.

In the same spirit, every i-th diagonal element $[\mathbf{G}]_{ii}$ is easily computed in terms of the reusable matrices $\mathbf{Q}$ and $\mathbf{D}$ as (see [30]):

$$g_{ii} = [\mathbf{G}]_{ii} = [(\mathbf{K} + \lambda\mathbf{I})^{-1}]_{ii} = \sum_j^k \; q_{ij}^2 \; / \; (d_{jj}+\lambda) \tag{11.8}$$

and the notation $[]_{ii}$ designates the i-th diagonal element of the matrix in the brackets.

11.3.7 Using incremental Cholesky decomposition

Since every local regularized square matrix $\mathbf{K}+\lambda\mathbf{I}$ is positive definite (a tiny positive value to the diagonal ensures positive definiteness), the system can be solved directly using the Cholesky decomposition [31] [32] which decomposes the regularized matrix as $(\mathbf{K} + \lambda\mathbf{I}) = \mathbf{R}^T\mathbf{R}$ where $\mathbf{R}$ is the upper triangular Cholesky factor. The code for Cholesky can be found in [32] The linear system can be solve by $(\mathbf{K} + \lambda\mathbf{I})\,\mathbf{w} = \mathbf{y}$ => $\mathbf{R}^T\mathbf{R}\,\mathbf{w} = \mathbf{y}$ => $\mathbf{R}^T\mathbf{a} = \mathbf{y}$.

Then the inverse is $\mathbf{G} \equiv (\mathbf{K}+\lambda\mathbf{I})^{-1} = \mathbf{R}^{-1}(\mathbf{R}^{-1})^T = \mathbf{S}\mathbf{S}^T$ where $\mathbf{S} = \mathbf{R}^{-1}$ represents the inverse of the upper triangular (in practice one first computes the inverse of the lower triangular $\mathbf{R}^T$ which is simpler to compute and then transposes it). With the Cholesky factors the weights $\mathbf{w}$ are solved directly by using (the process of solving a system with a triangular matrix) forward-substitutions and back-substitutions.

The optimization for the best k neighborhood searches the grid $\{\delta L, 2\delta L, \ldots, L_{\max}\}$ with step δL. Each time it adds δL rows and columns into the local kernel matrix $\mathbf{K}$.

Cholesky decomposition is incremental, regarding additional rows and columns (see fig. 11.3). Thus, if we first compute once the upper triangular $\mathbf{R}^{L\max \times L\max}$ for the matrix $(\mathbf{K}^{L\max \times L\max}+\lambda\mathbf{I})$, then any smaller sub-matrix $(\mathbf{K}^{k\times k}+\lambda\mathbf{I})$, that starts from the first row and column of $(\mathbf{K}^{L\max \times L\max}+\lambda\mathbf{I})$, has decomposition $\mathbf{R}^{k\times k}$ which is also known.

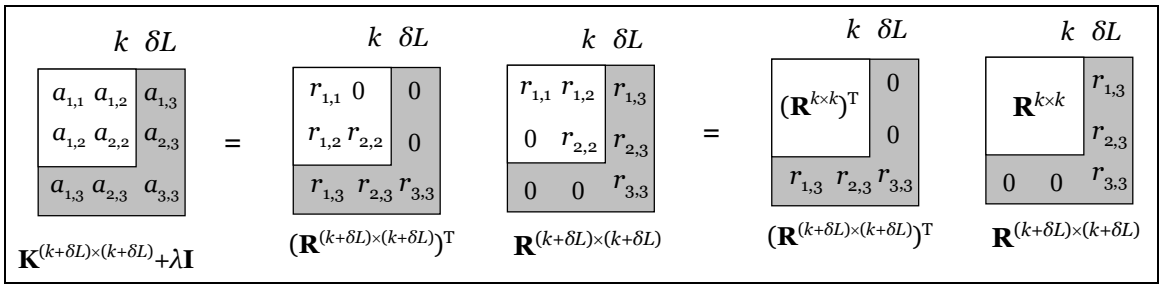


Figure 11.3. Incremental computations regarding the addition of new columns and rows δL, indicated in grey scale, to the kernel matrix $\mathbf{K}$ that is augmented by δL. The corresponding $(\mathbf{K}^{(k+\delta L)\times(k+\delta L)}+\lambda\mathbf{I})$, its upper triangular Cholesky factor $\mathbf{R}^{(k+\delta L)\times(k+\delta L)}$, as well as the inverse $\mathbf{S}=\mathbf{R}^{-1}$ of the upper triangular Cholesky factor, all retain their previous values, designated in white.

The computation of the matrix $\mathbf{G}^{(k+\delta L)\times(k+\delta L)} = (\mathbf{S}^{(k+\delta L)\times(k+\delta L)})(\mathbf{S}^{(k+\delta L)\times(k+\delta L)})^T$ in terms of $\mathbf{S}$ is illustrated in fig. 11.4. The computation steps is a series of rank one updates based on the previous $\mathbf{G}^{k\times k}$ values.

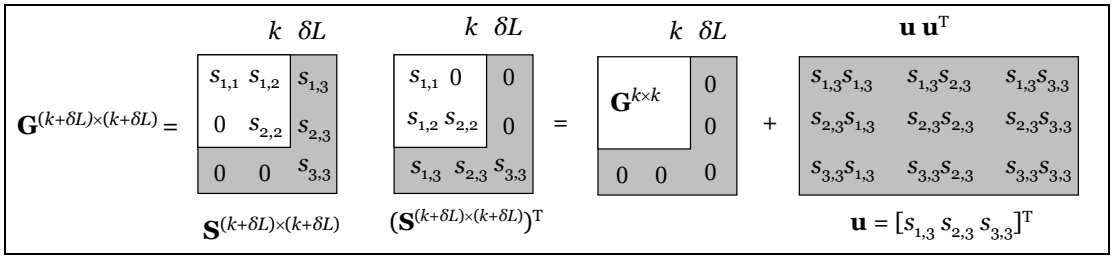


Figure 11.4. Rank one updating for the inverse **G** regarding the addition of new δL column and rows which are indicated in grey scale. When the local kernel matrix **K** is augmented symmetrically by δL columns and δL rows the corresponding inverse of the Gram matrix ($\mathbf{G}^{k+\delta L}$) is given in terms of $\mathbf{S}^{(k+\delta L)\times(k+\delta L)}$ that denotes the inverse of the upper triangular Cholesky factor.

Although the full inverse $\mathbf{G} \equiv (\mathbf{K}+\lambda\mathbf{I})^{-1}$ is given by $\mathbf{SS}^{\mathrm{T}}$, for the virtual leave-one-out cross-validation we only need the diagonal elements g_{ii} given by:

$$g_{ii} = [(\mathbf{K}+\lambda\mathbf{I})^{-1}]_{ii} = \sum_{j=i}^{k} s_{ij}^2 \tag{11.9}$$

where g_{ii} is an element of the matrix **G** and s_{ij} is an element of the matrix $\mathbf{S} = \mathbf{R}^{-1}$ that denotes the inverse of the upper triangular Cholesky factor.

When matrix **S** of size $L_{\max}\times L_{\max}$ is computed, then the diagonal elements g_{ii} can be computed recursively, during validating several candidate k values, as follows:

For $i = 0$ to $i = L_{\max}$ set $g_{ii} = 0$
set $k_{\text{prev}} = 0$
For each candidate k value ranging from δL to $L_{\max}$ with step δL
  For $i = 0$ to $i = k_{\text{prev}}$ set $g_{ii} = g_{ii} + \sum_{j=kprev}^{k} s_{ij}^2$ //recursive rank one updates
  For $i = k_{\text{prev}}$ to $i = k$ set $g_{ii} = g_{ii} + \sum_{j=i}^{k} s_{ij}^2$
  $k_{\text{prev}} = k_{\text{prev}} + \delta L$

Hence, Cholesky decomposition can be used to validate several candidate k values of neighbours at low cost.

All algorithms are given in the appendix. Algorithm 11.1 presents the training phase of the local learning regularization network when implemented with EVD, while algorithm 11.2 shows the testing phase with EVD. For the Cholesky implementations algorithm 11.3 presents the training phase, while algorithm 11.4 shows the testing phase.

11.3.8 Pre-computed stored local models

Pre-computing and storing representative local models $f_{k,n}(\mathbf{x}_n)$, one for each training example $\mathbf{x}_n$ could also improve the operation time. Using pre-computed stored local models avoids creating a new model (at runtime) for each new query point **q** based on the k-NN(**q**) list. Instead, the pre-computed model $f_{k,n}(\mathbf{x}_n)$ that is closest to **q** is used as predictor, where the closeness is defined by the distance $|\mathbf{x}_n - \mathbf{q}|$. In this model the k neighbours of $\mathbf{x}_n$ are used as neighbours of **q**. We examine four algorithms, one for each case previously mentioned, that follow the same interplay between local and global optimized parameters. The training and testing phases of the stored models are illustrated in algorithm 11.5 given in the appendix.

11.3.9 Multi-core Implementations

Although local learning RN uses localized calculations, still it has a substantial cost. A direct way to satisfy the computational demands of such local learning models is to use parallel computations [33]. The principal idea of such parallel learning is to divide a large problem into a number of smaller problems that can be solved concurrently, on either distributed memory or shared memory multiprocessing environments. To this end we examine a shared memory multi-core CPU platform in which the computational load can be divided into many threads, lowering thus substantially the processing time. With additional cores added on chip, individual CPU threads can be assigned and processed by their own units in hardware. Thus, a single problem is decomposed and solved by several threads without over-utilizing a single core. The parallelisation of applications on such multi-core platforms is based on the thread programming model which can use Pthreads, OpenMP, Intel Cilk++ or Intel TBB. We utilize Open Message Passing (OpenMP), which is an API based on fork-join operations when the program enters into a parallel region.

Each thread can exhibit both data and task-level parallelism as it can independently execute code within a same parallel region. For data-level parallelism, the common pool of main memory is allowed to be decomposed and cached on a per-core basis for efficient reuse. During the RN training phase we assign a different thread for a different training example. In task-level parallelism each core can asynchronously execute separate threads on separate data regions. We assign every thread to execute a different candidate local model. The model with the lower leave-one-out error is selected as the most suitable to compute $f_k(\mathbf{q})$.

11.4 Experimental Simulations

The first set of experimental simulations compare the four cases of the local learning Regularization Network. We investigate and compare for each one of the four cases the testing error rates of the online local learning models against the pre-computed stored local models. We also measure and compare the training time and testing time for each case when using either EVD or Choleksy implementation. The last set of simulations study the reduction of execution time versus the number of cores and the parallel speedup in the multi-core system.

11.4.1 Benchmark datasets

The benchmark datasets we use in the regression experiments are listed in Table 11.2. California Housing, White Wine quality, Red Wine quality, Boston Housing and Madelon can be found in the UCI repository (http://www.ics.uci.edu/~mlearn). Kinematics, Computer activity and Puma dynamic were downloaded from the Delve Repository (http://www.cs.toronto.edu/delve). The rest datasets like Friedman, 2Dplanes, Census, Elevators, Ailerons, Delta Elevators, and Delta Ailerons can be found in the Torgo's site (http://www.dcc.fc.up.pt/~ltorgo/Regression).

Table 11.2. Benchmark datasets for regression

Dataset	Instances	Features
Friedman	40768	10
2D planes	40768	10
Census (8L)	22784	8
California Housing	20640	8
Elevators	16599	18
Ailerons	13750	39
Delta Elevators	9517	6
Computer activity	8192	12
Kinematics (8NM)	8192	8
Delta Ailerons	7129	5
Puma dynamic (8NH)	8193	8
White wine quality	4898	11
Red wine quality	1599	11
Boston Housing	506	13
Madelon	2600	500

Every dataset in table 11.2 is randomly splitted into 50% training set and 50% test set. The input features and output targets have been scaled into the range [0,1]. The prediction error is measured on the test set using the Root Mean Squared Error (RMSE). This RMSE for each case and each dataset is averaged over 20 experimental trials.

11.4.2 Parameter settings

Parameter settings are crucial for all the four cases of the local learning regularization networks. The optimum number k of nearest neighbours in all four cases is found by searching the grid {10, 20, …, 100} with step 10. The optimum width parameter σ (local or global) is found by grid searching through a set {0.00625, 0.0125, 0.025, 0.05, 0.1, 0.3, 0.5, 0.7, 1.0, 1.2, 1.5, 1.7, 2.0, 2.5, 3.0, 4.0, 5.0, 6.0, 7.0, 8.0} of twenty candidate values with varying step. The optimum regularization parameter λ (local or global) is found by grid searching first through a set $\{2^{-15}, 2^{-14}\ldots, 2^{-2}, 2^{-1}\}$ of fifteen small values and then through another set of ten large values {1,…, 10} with step 1.

11.4.3 Comparison of RMSE in regression results

Table 11.3 shows the averaged RMSE for each dataset from the comparisons of the four local RN cases {local σ, local λ}, {global σ, local λ}, {local σ, global λ} and {global σ, global λ}. For each case the pre-computed stored local models are compared against the online local learning models. For each regression dataset the lowest RMSE is highlighted in boldface.

Table 11.3. Root Mean Squared Errors (RMSE) of the test set, from the four cases of the local learning Regularization Networks, averaged over 20 runs

	Pre-computed stored local models				Online local learning models			
Dataset	Case 1 local σ local λ	Case 2 global σ local λ	Case 3 local σ global λ	Case 4 global σ global λ	Case 1 local σ local λ	Case 2 global σ local λ	Case 3 local σ global λ	Case 4 global σ global λ
Friedman	0.04142	0.04092	0.04099	0.04090	0.04030	0.03922	0.03893	**0.03857**
2D planes	0.04192	0.04158	0.04165	0.04150	0.04184	0.04136	0.04143	**0.04126**
Census	0.06778	0.06575	0.06645	0.06503	0.06455	0.06293	0.06398	**0.06259**
California Housing	0.12442	0.12158	0.12138	0.11975	0.11648	0.11511	0.11572	**0.11460**
Elevators	0.03638	0.03564	0.03603	0.03541	0.03368	0.03316	0.03335	**0.03295**
Ailerons	0.04818	0.04750	0.04748	0.04658	0.04735	0.04613	0.04630	**0.04566**
Delta Elevators	0.05425	0.05365	0.05385	0.05330	0.05388	0.05329	0.05336	**0.05302**
Computer activity	0.03201	0.03108	0.03211	0.03064	0.02955	0.02922	0.02939	**0.02897**
Kinematics	0.06981	0.07078	0.06913	0.06853	0.06614	0.06686	0.06499	**0.06309**
Delta Ailerons	0.03948	0.03889	0.03917	0.03861	0.03887	0.03856	0.03858	**0.03835**
Puma dynamic	0.15496	0.15399	0.15447	0.15312	0.15372	0.15288	0.15269	**0.15002**
White wine quality	0.13057	0.12676	0.12972	0.12811	0.12173	0.11921	0.11828	**0.11779**
Red wine quality	0.13917	0.13611	0.13761	0.13310	0.13225	0.13055	0.13103	**0.12862**
Boston Housing	0.09202	0.08908	0.08882	0.08884	0.07989	0.07855	0.07867	**0.07738**
Madelon	0.50091	0.50186	0.50218	0.50003	0.48455	0.48430	0.48332	**0.48321**
Average Rankings	7.73	6.20	6.67	4.73	4.53	2.40	2.73	**1.00**
Wilcoxon p-values	<0.0001	<0.0001	<0.001	<0.0001	<0.0001	<0.0001	<0.0001	

The last column of table 11.3 has the lowest error of all. Thus, case 4 with online local learning outperforms all others.

The statistical significance of the results in table 11.3 was investigated via the Freidman test and the Wilcoxon test. We first perform the Friedman test (a detailed description can be found in [34]) which is a rank-based non-parametric test. Friedman uses the rankings of different learning algorithms (the columns of table 11.3) on multiple datasets (the rows of table 11.3). The null hypothesis states that all the methods perform equivalently and thus their ranks should be equivalent. The average rankings in the bottom of table 11.3 show that the proposed method in the last column (online local learning case 4) is ranked in the first place, according to the Friedman test. The Friedman chi-square statistic was found to be $\chi_F^2 = 94.69$. Assuming a significance level of 0.05, the p-value of the Friedman test was found to be p-value $< 10^{-8}$ which implies that the null hypothesis has to be rejected at a high level of confidence. Having rejected the null hypothesis, we can proceed with a paired comparison of methods. Therefore, a Wilcoxon signed-rank test is performed next, in order to investigate the statistical

significance of each pair of the different learning algorithms (columns in table 11.3). Each algorithm is paired with the last column which was ranked first. The Wilcoxon signed-rank test ranks the differences in performance of the two methods for each dataset, ignoring the signs, and compares the ranks for the positive and the negative differences. Assuming a significance level of 0.05 the p-values of the Wilcoxon signed-rank test on these pairs (the last column with the others) are illustrated in the bottom of table 11.3. The test returns true (h=1) in succeeding to reject the null hypothesis. From the Wilcoxon *p*-values it can be seen that all these paired comparisons are statistically significant, and hence we can confidently conclude that the online local learning case 4 yields better results than all the other cases.

From table 11.3 other conclusions can also be drawn. First there is a clear advantage of online local learning models versus the pre-computed stored models and second there is another extra advantage of global optimization of the local hyper-parameters. A discussion on these two trends is presented on the next subsections.

11.4.4 Advantage of Online local learning vs pre-computed stored models

Clearly the online local learning case 4 is better than the pre-computed stored case 4, and the same holds for the other three cases. All the online local learning cases are more accurate than their pre-computed stored counterparts. Apparently, there is a substantial gain in first waiting for the testing point $\mathbf{q}$ to arrive and then find its k-neighbours to perform online training for the local model $f_k(\mathbf{q})$, instead of using the pre-computed stored local model $f_k(\mathbf{x}_q)$ of the closest to $\mathbf{q}$ training point $\mathbf{x}_q$. Pre-computed stored models assume that the k-neighbours of $\mathbf{x}_q$ are similar to those of the testing point $\mathbf{q}$. However, the point $\mathbf{q}$ and its closest $\mathbf{x}_q$ might not share exactly the same neighbours. The locally optimized hyper-parameters of each stored model are based on the local errors of the $\mathbf{x}_q$ neighbourhood. This is an approximation since the $\mathbf{q}$ neighbourhood could be sometimes different. Hence, the online local learning cases perform better.

11.4.5 Advantage of global optimization of local hyper-parameters

Case 4 outperforms the other three cases both within the pre-computed stored models as well as within the online local learning models. Case 4 uses global optimization of all the hyper-parameters while the cases 1, 2 and 3 locally optimize either σ or λ or both.

Local optimization of the hyper-parameters monitors locally the virtual leave-one-out cross-validation errors E_{loo} of each neighbourhood. Each local learning model $f_k(\mathbf{x})$ is optimized for predicting all its training examples in the k-NN($\mathbf{x}$) list. That is why cases 1, 2 and 3 first locally optimize the hyper-parameters of the model $f_k(\mathbf{x})$, in order to achieve the best performance. Thus the weights of the best local models $f_k(\mathbf{x})$ are first optimized for best predicting all the neighbourhood points around $\mathbf{x}$, (in the hope that they would be suitable for $\mathbf{x}$ also) and then the best of those models are used for predicting $\mathbf{x}$.

Global optimization of all the hyper-parameters, like case 4 does, monitors globally the true errors $(y–f_k(\mathbf{x}))$ of all training examples and in local learning methods it is a natural leave-one-out cross-validation for the globally optimized local hyper-parameters. Each

regional model $f_k(\mathbf{x})$ is globally optimized for predicting each $\mathbf{x}$ specifically, by leave out this $\mathbf{x}$ from the training of the model $f_k(\mathbf{x})$.

Case 4 uses globally the true errors $(y–f_k(\mathbf{x}))$. For each training example $\mathbf{x}$ it finds a k-NN($\mathbf{x}$) list, which defines a dataset around $\mathbf{x}$ without the $\mathbf{x}$ itself (natural leave-one-out). Then it creates the same number of local models $f_k(\mathbf{x})$ as the other cases do, but without local optimization of the $\{k, \sigma, \lambda\}$ parameters. Rather it globally optimizes those parameters by monitoring only the true errors $(y–f_k(\mathbf{x}))$ for all models, which means that for each candidate local model it stores only the true error $e(k, \sigma, \lambda) = (y–f_k(\mathbf{x}))$, focusing thus globally on those parameters $\{k, \sigma, \lambda\}$ that best predict $\mathbf{x}$ only. The local virtual leave-one-out cross-validation training error E_{loo} of the neighbourhood points around $\mathbf{x}$ is not taken into account during the process.

It is worth mentioning that for a given dataset and a given case both Cholesky and EVD produce exactly the same outcome, the same local weights and the same root mean squared errors. This is not surprising since they are very solid matrix decompositions. In addition, all these algorithms are robust solutions since they remove ill-conditioning via regularization.

11.4.6 Execution Times

The execution times are important. Tables 11.4 and 11.5 illustrate the corresponding measurements of the training time and the testing time respectively for the four local RN cases when performed with either EVD or Cholesky.

11.4.6.1 Training times

Table 11.4 is very informative and reveals which local algorithm has the fastest training phase. We expect from cases 1, 2 and 3 to have similar execution times while case 4 must be faster because it avoids computing the diagonals g_{ii} and storing the local weight vectors $\mathbf{w}(k, \sigma_m, \lambda_l)$ for each candidate value $\{k, \sigma_m, \lambda_l\}$. In table 11.4 the lowest training time for each dataset is highlighted in boldface.

Table 11.4. Training time (in secs) for each case, by using EVD or Cholesky implementation,

Dataset	Case 1 Local σ Local λ		Case 2 Global σ Local λ		Case 3 Local σ Global λ		Case 4 Global σ Global λ	
	EVD	*Chol*	*EVD*	*Chol*	*EVD*	*Chol*	*EVD*	*Chol*
Friedman	19644	18524	19815	18550	19913	18261	17045	**9438**
2D planes	19092	18960	19090	18944	19082	18242	16295	**9505**
Census	11800	10711	11787	10697	11657	10718	10146	**5741**
California Housing	10823	9593	10895	9582	10937	9589	9485	**5087**
Elevators	8885	7837	8881	7827	8887	7832	7827	**4159**
Ailerons	7967	6490	7177	6471	7191	6486	6350	**3382**
Delta Elevators	4795	4488	4662	4496	4659	4389	4036	**2301**
Computer activity	4556	3859	4438	3837	4437	3768	3927	**2027**
Kinematics (8NM)	4068	3782	4059	3773	4042	3774	3490	**2012**

Delta Ailerons	3584	3334	3489	3325	3494	3325	3048	**1765**
Puma dynamic	2506	1937	3648	1932	2984	1933	2612	**1125**
White wine quality	2934	2340	3726	2335	3532	2335	2733	**1236**
Red wine quality	857	753	863	752	967	753	708	**401**
Boston Housing	302	238	274	239	329	240	263	**126**
Madelon	1230	1145	1230	1143	1231	1143	1077	**612**

From table 11.4 one can observe that the training times for cases 1, 2 and 3 are comparable since they all need to compute the virtual leave-one-out error and store the weights (see the three extra lines in the corresponding algorithms), while case 4 is faster since it don't. When comparing Cholesky and EVD for each case separately it is clear that the training via Cholesky is a bit faster than its EVD counterparts in cases 1, 2, 3 and 4.

Within the Cholesky implementations the cases 1, 2 and 3 are similar in speed as expected, while case 4 is much faster than them (with almost half training time), and the reason is that it avoids computing the diagonals g_{ii} and storing the local weights for each candidate value $\{k, \sigma_m, \lambda_l\}$. The fastest of all the training algorithms is case 4 when implemented via Cholesky.

Notice also that there is an interesting trade-off. If one uses more candidate width values σ_m then the cost will increase evenly for both Cholesky and EVD implementations. If less candidate k values are used then the situation will be reversed and EVD will become faster than Cholesky. However if more candidate k values are used then Cholesky will become faster than EVD. In addition, if less candidate regularization values λ_l are used then the Cholesky implementations will again become even faster than their EVD counterparts.

11.4.6.2 Testing times

During the testing phase whose timings are illustrated in table 11.5, the local RN cases that need less local parameters to optimize are favoured more. Therefore, case 4 that uses only globally optimized parameters, does not need to optimize any parameter on-the-fly and thus will be the fastest. For each dataset in table 11.5 the lowest testing time is highlighted in boldface.

Table 11.5. Testing time for each case, by using either EVD or Cholesky implementation, in secs

Dataset	Case 1 Local σ Local λ		Case 2 Global σ Local λ		Case 3 Local σ Global λ		Case 4 Global σ Global λ	
	EVD	*Chol*	*EVD*	*Chol*	*EVD*	*Chol*	*EVD*	*Chol*
Friedman	4367	16115	245	607	3433	522	184	**19**
2D planes	4402	17360	213	601	3185	568	176	**19**
Census	2717	9615	135	350	2182	427	93	**11**
California Housing	2542	7027	121	318	2006	290	89	**10**

Elevators	2045	6901	106	260	1617	240	82	**9**
Ailerons	1699	5947	92	213	1321	196	78	**10**
Delta Elevators	1087	4093	51	146	823	131	45	**5**
Computer activity	1028	3580	58	128	802	112	44	**4**
Kinematics (8NM)	940	3527	51	122	732	103	40	**4**
Delta Ailerons	802	2063	40	109	620	98	26	**3**
Puma dynamic	788	1840	43	74	434	67	22	**2**
White wine quality	570	1748	30	65	410	60	23	**2**
Red wine quality	187	682	10	24	145	22	8	**0.7**
Boston Housing	66	75	3	7	51	7	2	**0.3**
Madelon	291	82	32	51	229	46	29	**14**

First before commented the results in table 11.5 it has to be noted that theoretically the EVD amortized time for a dense local kernel matrix of size $k \times k$ is analogous to $O(3k^3)$ while the equivalent time for Cholesky is analogous to $O(k^3/3)$) (see [31][32]). An inspection of the results between the last two columns of table 11.5, which solve the same problem in case 4, reveals that what we expect theoretically is also what we get experimentally.

From table 11.5 it is evident that the actual online operation of Local RN (the testing phase) via EVD is faster in cases 1 and 2, while in cases 3 and 4 the online operation with Cholesky is faster. The fastest of all online operation algorithms is case 4 with Cholesky.

The testing phase of cases 2 and 3 is faster than case 1, since they optimize on-the-fly and locally only one of the two hyper-parameters. Case 4 testing time is impressively much lower. Case 4 uses for every testing example the same best pair $\{\sigma, \lambda\}$ of hyper-parameters found during training and thus it is the fastest of all. In a nutshell, case 4 outperforms all other cases both in accuracy as well in speed.

To give a scalability paradigm, for the California dataset the case 4 during training computes and validates more than 10000 local models per second, and during testing it predicts 1000 testing points per second.

11.4.7 Comparison of Standard Deviations

Table 11.6 presents the comparison of the Standard Deviations of the testing errors when averaged over 20 runs. The results are exactly the same for both Cholesky and EVD.

Table 11.6. Comparison of the Standard Deviations of the testing errors, averaged over 20 runs, for the four cases of the local learning regularization networks

	Stored closest local models				Online local learning models			
Dataset	Case 1 local σ local λ	Case 2 global σ local λ	Case 3 local σ global λ	Case 4 global σ global λ	Case 1 local σ local λ	Case 2 global σ local λ	Case 3 local σ global λ	Case 4 global σ global λ

Friedman	0.00026	0.00025	0.00039	0.00029	0.00035	0.00031	0.00031	0.00030
2D planes	0.00019	0.00018	0.00017	0.00017	0.00018	0.00016	0.00017	0.00016
Census	0.00147	0.00137	0.00125	0.00112	0.00115	0.00099	0.00102	0.00096
California Housing	0.00396	0.00285	0.00338	0.00131	0.00165	0.00163	0.00140	0.00128
Elevators	0.00051	0.00046	0.00045	0.00044	0.00031	0.00032	0.00030	0.00032
Ailerons	0.00045	0.00045	0.00048	0.00047	0.00051	0.00047	0.00043	0.00043
Delta Elevators	0.00065	0.00063	0.00065	0.00058	0.00060	0.00057	0.00056	0.00058
Computer activity	0.00098	0.00132	0.00129	0.00102	0.00062	0.00055	0.00072	0.00043
Kinematics	0.00091	0.00092	0.00084	0.00098	0.00078	0.00075	0.00088	0.00080
Delta Ailerons	0.00072	0.00051	0.00061	0.00051	0.00079	0.00051	0.00071	0.00052
Puma dynamic	0.00207	0.00186	0.00225	0.00201	0.00186	0.00173	0.00185	0.00167
White wine quality	0.00467	0.00288	0.00318	0.00262	0.00299	0.00187	0.00195	0.00160
Red wine quality	0.00385	0.00285	0.00429	0.00331	0.00333	0.00328	0.00360	0.00311
Boston Housing	0.00850	0.00819	0.00831	0.00741	0.00811	0.00670	0.00801	0.00737
Madelon	0.00508	0.00581	0.00617	0.00495	0.00265	0.00239	0.00215	0.00168

As observed from table 11.6 the standard deviations of the errors are small and similar through out the four cases of the local learning regularization networks.

11.4.8 Best global parameters found during training

Table 11.7 illustrates for each dataset and each local RN case the best global parameters that were found during the training phase, as averaged over the 20 runs.

Table 11.7. Best k number of neighbours and best global parameters σ and λ.

Dataset	Case 1 Local σ Local λ	Case 2 Global σ Local λ		Case 3 Local σ Global λ		Case 4 Global σ Global λ		
	k	k	σ	k	λ	k	σ	λ
Friedman	97	98	5.100	96	0.0001	97	5.000	0.0001
2D planes	94	96	6.100	95	0.0031	96	6.500	0.0062
Census	95	97	5.885	96	0.0142	98	5.545	0.0005
California Housing	90	77	4.950	94	0.0073	79	4.260	0.0006
Elevators	94	95	6.650	96	0.0001	95	5.450	0.0001
Ailerons	96	99	6.275	98	0.0031	99	6.250	0.0054
Delta Elevators	95	97	5.850	98	0.0021	95	4.875	0.0095
Computer activity	84	82	5.510	85	0.0004	78	6.275	0.0001
Kinematics	96	96	2.230	94	0.0060	94	1.180	0.0037
Delta Ailerons	89	88	5.275	87	0.0016	89	3.085	0.0031
Puma dynamic	92	95	2.250	94	0.0162	96	2.055	0.0106
White wine quality	83	39	1.575	31	0.1940	35	0.935	0.0893
Red wine quality	94	95	1.880	91	0.2823	91	1.515	0.1450
Boston Housing	75	74	3.180	78	0.0023	76	3.960	0.0025
Madelon	65	65	4.125	54	0.3525	58	6.550	0.4375

Case 1 finds a global k for all examples during training and for each example it optimizes a different pair $\{\sigma, \lambda\}$ of hyper-parameters. Case 2 finds a global k and a global σ for all training examples. Case 3 finds a global k and a global λ for all training examples. Case 4 finds a global k, a global σ and a global λ for all training examples and uses them in the testing examples. The best number k is similar in the four cases.

Here the best k number of neighbors and best global parameters σ and λ are found different from our preliminary experiments reported in [26] because we extend the grid search for all the parameters. Here the maximum k parameter is now 100, while grid searching for many smaller λ values is also included. Note that there is a trade-off between σ and λ. The best global regularization λ values in Table 11.7 are not as extremely small as they seem; since they are per example. You must multiply them with k=100 to get the global λ values per model. Also note that larger values for global σ are usually preferred. Parameter σ defines the Reproducing kernel Hilbert space (RKHS). It is the new field $\varphi(\mathbf{x})$ that maps $\mathbf{x}$ into this space. So in kernel machines it is not limited to small values when data features are scaled in [0, 1]. On the other hand the best regularization parameter λ is the one that is related to the noise level. In a full training dataset there are many clusters and not all points are relevant to each other (there exist a high noise level per point). In local learning models the k-NN neighbors form a dense cluster around each query point $\mathbf{q}$ when creating a local model. Regularization makes the mapping function of a local model smooth and robust to oscillations. The dense clusters are smooth enough. Inside the cluster almost all neighbor points are relevant to the query point, and to each other. There are no outliers or irrelevant points. The noise level and its interference for such dense clusters must be low (as the estimated best λ values are). The σ values must be large enough so as not to loose information about the cluster (as the estimated best σ values are). At the same time for a global parameter σ and a global parameter λ, there is only one Hilbert space. In a nutshell, each local model produced by a dense cluster of k-NN neighbors prefers a large σ and a small λ. Those tendencies coincide with the results in table 11.7 which are produced by the virtual leave-one-out cross-validation training process.

11.4.9 Parallel speedup measurements

For the parallel speedup measurements of the local learning regularization network on multi-cores the experimental simulations were conducted on a system consisted of two CPU's, AMD opteron (tm) Processor 6128 HE, with eight cores each, 800MHz clock speed and 16GB RAM, under Ubuntu Linux 10.04 operating system.

The OpenMP version 3.0 was used. We simulate the training phase of the local RN and measure the total parallel execution time versus the number of cores and the speedup ratio Ts / Tp, where Ts the sequential run time in a single core and Tp the time that simulates the network in parallel. The timing measurements fix the size of the problem and increase the number of cores.

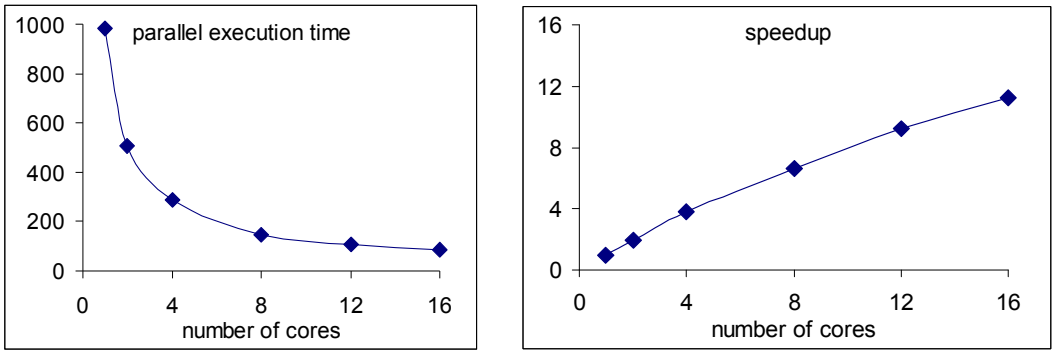


Figure 11.5. (a) Reduction of execution time versus the number of threads, (b) Speedup

Fig. 11.5(a) illustrates the time reduction on increasing the number of cores. Fig. 11.5(b) presents the speedup ratio that assists in the scalability analysis. Using 16 cores the parallel execution time was reduced by 92%.

11.5 Summary

This chapter investigates local learning approaches of regularization networks that create on-the-fly a different local model specifically designed for predicting a different particular testing point by using only the k-nearest neighbour training data to this testing point. In local learning methods the lazy use of the training dataset increases the online operational cost. For improving the online prediction performance of the local RN, both in accuracy and speed, many options are explored.

The first option is to exploit the interplay between locally optimized and globally optimized hyper-parameters, in order to tackle with the extra computational overhead during the network operation that must optimize locally the width parameter σ and the regularization parameter λ. To this end, we investigate which of the hyper-parameters can be optimized globally and off-line. The resulting four cases use {local σ, local λ}, {global σ, local λ}, {local σ, global λ} and {global σ, global λ} and reveal that the last case is the fastest during off-line training as well during on-line operation. Beyond the substantial reduction of the computational cost the last case (case 4), which takes advantage of the global optimization of the two local hyper-parameters, produces more accurate models than the other three cases. In a nutshell, case 4 outperforms all other parameter optimization cases both in accuracy as well in speed.

The second option is to study efficient matrix decompositions (Eigen decomposition or Cholesky) for the off-line training and on-line testing phases, and take advantage of the re-usable matrices those decompositions produce. We demonstrate that while EVD is more suitable for validating several candidate λ values with a fixed k size of the kernel matrix at low cost, Cholesky should be preferred for validation of several k candidate values with a fixed λ. We show that for the local RN training phase the Cholesky implementations are faster that their EVD counterparts for each parameter optimization case, and case 4 with Cholesky has the fastest training time. During the testing phase EVD is faster in cases 1 and 2, while in cases 3 and 4 the online operation with Cholesky is much faster. However the fastest of all during the testing phase is case 4 when implemented with Cholesky. Hence the Cholesky based implementations should be the preferred methods for both training as well as operation of the local learning RN.

A third option is to use an approximation for improve further the online operation time by pre-compute and store an optimized representative local model for each training example, and call the closest to the query testing point such model during the online operation. We find that this option deteriorates the accuracy performance, since a testing point and its closest training point might not share exactly the same neighbours. Apparently, there is a substantial gain and advantage in waiting for a testing point to arrive before building a local learning model, and hence the online local learning RNs are more accurate than their pre-computed stored local models.

Last but not least option is that of using parallelism in multi-cores for the local computations in order to further reduce the execution times. For speeding up the local predictions the simulations show that the local regularization network parallelizes well in multi-core systems.

An interesting future work, given that Cholesky is suitable for parallel implementation using Graphic processing units (GPUs), could be that of GPU accelerated online local learning RN for classification. The regularization network solution is very closely related to other kernel-based algorithms that combine Tikhonov regularization, like kernel ridge regression, regularized least-squares classification, kernel least mean squares, gaussian process regression and least-squares support vector machines. These methods are based on a kernel matrix and they use a kernel width parameter and a regularization parameter. Hence, the possibility of straightforwardly extending the present study to those applications could be worthwhile exploring in the future.

Appendix

For all the algorithms we use a lot of caching to speed up the process. The training phase of each case must also find the best number of neighbors, denoted as k_{best}. We search for the best k number in the grid $\{\delta L, 2\delta L, \ldots, L_{max}\}$ where L_{max} is the maximum candidate number of neighbors. A local distance matrix maintains the cached distances between the neighbour points. Based on this matrix a cached local kernel matrix is created once for every candidate σ_m value. Thus for each L_{max}, σ_m and λ_l value only one Cholesky is computed for the kernel matrix. Then, progressively the Cholesky back substitution solves for the local weights of all the k candidate values. All four local RN cases use the minimum global training errors to find the best global parameters.

In the training phase there are 3 loops one inside another. One loop iterates through the candidate width values σ_m, another loop iterates through the candidate k-neighbor values and another iterates through the candidate regularization values λ_l. The ordering of the loops is important. For the best and fastest ordering in the EVD implementations the loop that tests the widths σ_m must be first, followed by a second loop that iterates through the candidate k-neighbor values which inside has the last loop that validates the candidate regularization values λ_l. In the Cholesky implementations the fastest ordering of computations again has first the loop for the widths σ_m, but now the second loop must iterate through the candidate regularization values λ_l, and the loop for the candidate k-neighbor values must be the third.

Algorithm 11.1. Training phase of a local learning regularization network with EVD.

Algorithm 11.1: local RN training with EVD

1 For each training point $\mathbf{x}_n$ (n = 1,…,N)

1.1 find its L_{max}-nearest training points and set k-NN($\mathbf{x}_n$) = $\{\mathbf{x}_j, y_j\}_{j=1}^{L\max}$

1.2 compute cached distance vector $\mathbf{v}_n$ with elements $v_j = \|\mathbf{x}_n - \mathbf{x}_j\|^2$, $\mathbf{x}_j \in k$-NN($\mathbf{x}_n$)

1.3 compute cached distance matrix $\mathbf{\Delta}^{L\max \times L\max}$ with entries $\delta_{ij} = \|\mathbf{x}_i - \mathbf{x}_j\|^2$, $\mathbf{x}_i, \mathbf{x}_j \in k$-NN($\mathbf{x}_n$)

1.4 For each candidate width value σ_m

1.4.1 compute cached kernel matrix $\mathbf{C}^{L\max \times L\max}$ with entries $c_{ij} = \exp(-\delta_{ij} / \sigma_m^2)$

1.4.2 set $\mathbf{h} = [h_1,\ldots,h_{L_{\max}}]$ with $h_j = \exp(-v_j / \sigma_m^2)$, ($j$ =1,…, $L_{\max}$)

1.4.3 For each candidate k value ranging from δL to $L_{\max}$ with step δL

1.4.3.1 set $\mathbf{K} = \mathbf{C}^{k\times k}$ (equal to the first k rows and columns of $\mathbf{C}$)

1.4.3.2 perform EVD via $\mathbf{K} = \mathbf{QDQ}^{\mathrm{T}}$ and store the vector $[\mathbf{Q}^{\mathrm{T}}\mathbf{y}]$

1.4.3.3 For each candidate regularization value λ_l

1.4.3.3.1 solve the local weights $\mathbf{w}$ with $w_i = \sum_{j=1}^{k} q_{ij}[\mathbf{Q}^{\mathrm{T}}\mathbf{y}]_j/(d_{jj}+\lambda_l)$

1.4.3.3.2 find all diagonals $g_{ii} = \sum_{j=1}^{k} q_{ij}^2/(d_{jj}+\lambda_l)$, (i,j=1,..,k)

1.4.3.3.3 store the local error $E_{Loo}(k, \sigma_m, \lambda_l) = (1/k)\sum_{i=1}^{k} (w_i / g_{ii})^2$

1.4.3.3.4 store the local weight vector $\mathbf{w}(k, \sigma_m, \lambda_l) = \mathbf{w}$

} lines for cases 1,2,3

——————— Case 4 ———————

1.4.3.3.5 predict $f_k(\mathbf{x}_n) = \mathbf{h}\,\mathbf{w}$ and update $e_4(k, \sigma_m, \lambda_l) = e_4(k, \sigma_m, \lambda_l) + (y_n - f_k(\mathbf{x}_n))^2$

——————— Case 1 ———————

1.5 For each candidate k value ranging from δL to $L_{\max}$ with step δL

1.5.1 find local $\{\sigma_{best}, \lambda_{best}\}$ and weights $\mathbf{w}(k, \sigma_{best}, \lambda_{best})$ with the lowest $E_{Loo}(k, \sigma_m, \lambda_l)$

1.5.2 set $\mathbf{h} = [h_1,\ldots,h_k]$ with $h_j = \exp(-v_j / \sigma_{best}^2)$, ($j$ =1,…, k)

1.5.3 predict $f_k(\mathbf{x}_n) = \mathbf{h}\,\mathbf{w}(k, \sigma_{best}, \lambda_{best})$ and update $e_1(k) = e_1(k) + (y_n - f_k(\mathbf{x}_n))^2$

——————— Case 2 ———————

1.5 For each candidate k value ranging from δL to $L_{\max}$ with step δL

1.5.1 For each candidate width value σ_m

1.5.1.1 find local λ_{best} and weights $\mathbf{w}(k, \sigma_m, \lambda_{best})$ with the lowest $E_{Loo}(k, \sigma_m, \lambda_l)$

1.5.1.2 set $\mathbf{h} = [h_1,\ldots,h_k]$ with $h_j = \exp(-v_j / \sigma_m^2)$, ($j$ =1,…, k)

1.5.1.3 predict $f_k(\mathbf{x}_n) = \mathbf{h}\,\mathbf{w}(k, \sigma_m, \lambda_{best})$ and update $e_2(k,\sigma_m) = e_2(k,\sigma_m) + (y_n - f_k(\mathbf{x}_n))^2$

——————— Case 3 ———————

1.5 For each candidate k value ranging from δL to $L_{\max}$ with step δL

1.5.1 For each candidate regularization value λ_l

1.5.1.1 find local σ_{best} and weights $\mathbf{w}(k, \sigma_{best}, \lambda_l)$ with the lowest $E_{Loo}(k, \sigma_m, \lambda_l)$

1.5.1.2 set $\mathbf{h} = [h_1,\ldots,h_k]$ with $h_j = \exp(-v_j / \sigma_{best}^2)$, ($j$ =1,…, k)

1.5.1.3 predict $f_k(\mathbf{x}_n) = \mathbf{h}\,\mathbf{w}(k, \sigma_{best}, \lambda_l)$ and update $e_3(k,\lambda_l) = e_3(k,\lambda_l) + (y_n - f_k(\mathbf{x}_n))^2$

Case 1: find global k_{best} with the minimum global training error $e_1(k)$

Case 2: find global $\{k_{best}, \sigma_{best}\}$ with the minimum global training error $e_2(k,\sigma_m)$

Case 3: find global $\{k_{best}, \lambda_{best}\}$ with the minimum global training error $e_3(k, \lambda_l)$

Case 4: find global $\{k_{best}, \sigma_{best}, \lambda_{best}\}$ with the minimum global training error $e_4(k, \sigma_m, \lambda_l)$

Algorithm 11.2. Testing phase of a local learning regularization network with EVD.

Algorithm 11.2: local RN testing with EVD

1 For each testing point $\mathbf{x}_n$ ($n = 1,\ldots,N$)

1.1 set $k = k_{best}$ and find its k-nearest training points $k\text{-NN}(\mathbf{x}_n) = \{\mathbf{x}_j, y_j\}_{j=1}^{kbest}$

1.2 compute cached distance matrix $\mathbf{\Delta}^{k\times k}$ with $\delta_{ij} = \|\mathbf{x}_i - \mathbf{x}_j\|^2$, $\mathbf{x}_i,\mathbf{x}_j \in k\text{-NN}(\mathbf{x}_n)$

——————— Case 1 (with global k_{best}) ———————

1.3 For each candidate width value σ_m

1.3.1 compute local kernel matrix $\mathbf{K}$ with entries $k_{ij} = \exp(-\delta_{ij} / \sigma_m^2)$

1.3.2 perform EVD via $\mathbf{K} = \mathbf{QDQ}^{\mathrm{T}}$ and store the vector $[\mathbf{Q}^{\mathrm{T}}\mathbf{y}]$

1.3.3 For each candidate regularization value λ_l

1.3.3.1 store the local weight vector $\mathbf{w}(\sigma_m, \lambda_l)$ with $w_i = \sum_{j=1}^{k} q_{ij}[\mathbf{Q}^{\mathrm{T}}\mathbf{y}]_j/(d_{jj}+ \lambda_l)$

1.3.3.2 find local $E_{Loo}(\sigma_m, \lambda_l) = (1/k)\sum_{i=1}^{k} (w_i / g_{ii})^2$ where $g_{ii} = \sum_{j=1}^{k} q_{ij}^2/(d_{jj}+ \lambda_l)$

1.4 find local $\{\sigma_{best}, \lambda_{best}\}$ and weights $\mathbf{w}(\sigma_{best}, \lambda_{best})$ with the lowest $E_{Loo}(\sigma_m, \lambda_l)$

1.5 set $\mathbf{h} = [h_1,\ldots,h_{k_{best}}]$ with $h_j = \exp(-\|\mathbf{x}_n - \mathbf{x}_j\|^2/ \sigma_{best}^2)$

1.6 predict $f_k(\mathbf{x}_n) = \mathbf{h}\,\mathbf{w}(\sigma_{best}, \lambda_{best})$ and update $e_1 = e_1 + (y_n - f_k(\mathbf{x}_n))^2$

——————— Case 2 (with global k_{best}, σ_{best}) ———————

1.3 compute local kernel matrix $\mathbf{K}$ with entries $k_{ij} = \exp(-\delta_{ij} / \sigma_{best}^2)$

1.4 perform EVD via $\mathbf{K} = \mathbf{QDQ}^{\mathrm{T}}$ and store the vector $[\mathbf{Q}^{\mathrm{T}}\mathbf{y}]$

1.5 For each candidate regularization value λ_l

1.5.1 solve the local weight vector $\mathbf{w}$ with $w_i = \sum_{j}^{k} q_{ij}[\mathbf{Q}^{\mathrm{T}}\mathbf{y}]_j/(d_{jj}+ \lambda_l)$

1.5.2 find local $E_{Loo}(\lambda_l) = (1/k)\sum_{i=1}^{k} (w_i / g_{ii})^2$ where $g_{ii} = \sum_{j=1}^{k} q_{ij}^2/(d_{jj}+ \lambda_l)$

1.5.3 store the local weight vector $\mathbf{w}(\lambda_l) = \mathbf{w}$

1.6 find local λ_{best} and weights $\mathbf{w}(\lambda_{best})$ with the lowest $E_{Loo}(\lambda_l)$

1.7 set $\mathbf{h} = [h_1,\ldots,h_{k_{best}}]$ with $h_j = \exp(-\|\mathbf{x}_n - \mathbf{x}_j\|^2/ \sigma_{best}^2)$

1.8 predict $f_k(\mathbf{x}_n) = \mathbf{h}\,\mathbf{w}(\lambda_{best})$ and update $e_2 = e_2 + (y_n - f_k(\mathbf{x}_n))^2$

——————— Case 3 (with global k_{best}, λ_{best}) ———————

1.3 For each candidate width value σ_m

1.3.1 compute local kernel matrix $\mathbf{K}$ with entries $k_{ij} = \exp(-\delta_{ij} / \sigma_m^2)$

1.3.2 perform EVD via $\mathbf{K} = \mathbf{QDQ}^{\mathrm{T}}$ and store the vector $[\mathbf{Q}^{\mathrm{T}}\mathbf{y}]$

1.3.3 store the local weight vector $\mathbf{w}(\sigma_m)$ with $w_i = \sum_{j}^{k} q_{ij}[\mathbf{Q}^{\mathrm{T}}\mathbf{y}]_j/(d_{jj}+ \lambda_{best})$

1.3.4 find local $E_{Loo}(\sigma_m) = (1/k)\sum_{i=1}^{k} (w_i / g_{ii})^2$ where $g_{ii} = \sum_{j=1}^{k} q_{ij}^2/(d_{jj}+ \lambda_{best})$

1.4 find local σ_{best} and best local weights $\mathbf{w}(\sigma_{best})$ with the lowest $E_{Loo}(\sigma_m)$

1.5 set $\mathbf{h} = [h_1,\ldots,h_{k_{best}}]$ with $h_j = \exp(-\|\mathbf{x}_n - \mathbf{x}_j\|^2/ \sigma_{best}^2)$

1.6 predict $f_k(\mathbf{x}_n) = \mathbf{h}\,\mathbf{w}(\sigma_{best})$ and update $e_3 = e_3 + (y_n - f_k(\mathbf{x}_n))^2$

——————— Case 4 (with global k_{best}, σ_{best}, λ_{best}) ———————

1.3 compute local kernel matrix $\mathbf{K}$ with entries $k_{ij} = \exp(-\delta_{ij} / \sigma_{best}^2)$

1.4 perform EVD via $\mathbf{K} = \mathbf{QDQ}^{\mathrm{T}}$ and store the vector $[\mathbf{Q}^{\mathrm{T}}\mathbf{y}]$

1.5 find the local weights $\mathbf{w}$ with $w_i = \sum_{j=1}^{k} q_{ij}[\mathbf{Q}^{\mathrm{T}}\mathbf{y}]_j/(d_{jj}+ \lambda_{best})$

1.5 set $\mathbf{h} = [h_1,\ldots,h_{k_{best}}]$ with $h_j = \exp(-\|\mathbf{x}_n - \mathbf{x}_j\|^2/ \sigma_{best}^2)$, ($j = 1,\ldots, k_{best}$)

1.6 predict $f_k(\mathbf{x}_n) = \mathbf{h}\,\mathbf{w}$ and update $e_4 = e_4 + (y_n - f_k(\mathbf{x}_n))^2$

Algorithm 11.3. Training phase of a local learning regularization network with Cholesky

Algorithm 11.3: local RN training with Cholesky

1 For each training point $\mathbf{x}_n$ ($n = 1,\ldots,N$)

1.1 find its $L_{\max}$-nearest training points and set k-NN($\mathbf{x}_n$) = $\{\mathbf{x}_j, y_j\}_{j=1}^{L\max}$

1.2 compute cached distance vector $\mathbf{v}_n$ with elements $v_j = \|\mathbf{x}_n - \mathbf{x}_j\|^2$, $\mathbf{x}_j \in k$-NN($\mathbf{x}_n$)

1.3 compute cached distance matrix $\mathbf{\Delta}^{L\max \times L\max}$ with entries $\delta_{ij} = \|\mathbf{x}_i - \mathbf{x}_j\|^2$, $\mathbf{x}_i, \mathbf{x}_j \in k$-NN($\mathbf{x}_n$)

1.4 For each candidate width value σ_m

1.4.1 compute cached kernel matrix $\mathbf{K}^{L\max \times L\max}$ with entries $c_{ij} = \exp(-\delta_{ij} / \sigma_m^2)$

1.4.2 set $\mathbf{h} = [h_1,\ldots,h_{L_{\max}}]$ with $h_j = \exp(-v_j / \sigma_m^2)$, ($j$ =1,…, $L_{\max}$)

1.4.3 For each candidate regularization value λ_l

1.4.3.1 find Cholesky of $(\mathbf{K} + \lambda_l) = \mathbf{R}^{\mathrm{T}}\mathbf{R}$ and store the inverse $\mathbf{S} = \mathbf{R}^{-1}$

1.4.3.2 For each candidate k value ranging from δL to $L_{\max}$ with step δL

1.4.3.2.1 set $\mathbf{R}_k = \mathbf{R}^{k\times k}$, $\mathbf{S}_k = \mathbf{S}^{k\times k}$ (equal to the first k rows and columns of $\mathbf{R}$ and $\mathbf{S}$)

1.4.3.2.2 via back substitutions solve $\mathbf{R}_k^{\mathrm{T}}\,\mathbf{R}_k\,\mathbf{w} = \mathbf{y}$ to find the weights $\mathbf{w}$

1.4.3.2.3 via rank one updates find diagonals $g_{ii} = \sum_{j=i}^{k} s_{ij}^2$, (i=1,..,k)

1.4.3.2.4 store the local error $E_{Loo}(k, \sigma_m, \lambda_l) = (1/k) \sum_{i=1}^{k} (w_i / g_{ii})^2$

1.4.3.2.5 store the local weight vector $\mathbf{w}(k, \sigma_m, \lambda_l) = \mathbf{w}$

} lines for cases 1,2,3

——————— Case 4 ———————

1.4.3.2.6 predict $f_k(\mathbf{x}_n) = \mathbf{h}\,\mathbf{w}$ and update $e_4(k, \sigma_m, \lambda_l) = e_4(k, \sigma_m, \lambda_l) + (y_n - f_k(\mathbf{x}_n))^2$

——————— Case 1 ———————

1.5 For each candidate k value ranging from δL to $L_{\max}$ with step δL

1.5.1 find local $\{\sigma_{best}, \lambda_{best}\}$ and weights $\mathbf{w}(k, \sigma_{best}, \lambda_{best})$ with the lowest $E_{Loo}(k, \sigma_m, \lambda_l)$

1.5.2 set $\mathbf{h} = [h_1,\ldots,h_k]$ with $h_j = \exp(-v_j / \sigma_{best}^2)$, ($j$ =1,…, k)

1.5.3 predict $f_k(\mathbf{x}_n) = \mathbf{h}\,\mathbf{w}(k, \sigma_{best}, \lambda_{best})$ and update $e_1(k) = e_1(k) + (y_n - f_k(\mathbf{x}_n))^2$

——————— Case 2 ———————

1.5 For each candidate k value ranging from δL to $L_{\max}$ with step δL

1.5.1 For each candidate width value σ_m

1.5.1.1 find local λ_{best} and weights $\mathbf{w}(k, \sigma_m, \lambda_{best})$ with the lowest $E_{Loo}(k, \sigma_m, \lambda_l)$

1.5.1.2 set $\mathbf{h} = [h_1,\ldots,h_k]$ with $h_j = \exp(-v_j / \sigma_m^2)$, ($j$ =1,…, k)

1.5.1.3 predict $f_k(\mathbf{x}_n) = \mathbf{h}\,\mathbf{w}(k, \sigma_m, \lambda_{best})$ and update $e_2(k,\sigma_m) = e_2(k,\sigma_m) + (y_n - f_k(\mathbf{x}_n))^2$

——————— Case 3 ———————

1.5 For each candidate k value ranging from δL to $L_{\max}$ with step δL

1.5.1 For each candidate regularization value λ_l

1.5.1.1 find local σ_{best} and weights $\mathbf{w}(k, \sigma_{best}, \lambda_l)$ with the lowest $E_{Loo}(k, \sigma_m, \lambda_l)$

1.5.1.2 set $\mathbf{h} = [h_1,\ldots,h_k]$ with $h_j = \exp(-v_j / \sigma_{best}^2)$, ($j$ =1,…, k)

1.5.1.3 predict $f_k(\mathbf{x}_n) = \mathbf{h}\,\mathbf{w}(k, \sigma_{best}, \lambda_l)$ and update $e_3(k,\lambda_l) = e_3(k,\lambda_l) + (y_n - f_k(\mathbf{x}_n))^2$

Case 1: find global k_{best} with the minimum global training error $e_1(k)$

Case 2: find global $\{k_{best}, \sigma_{best}\}$ with the minimum global training error $e_2(k,\sigma_m)$

Case 3: find global $\{k_{best}, \lambda_{best}\}$ with the minimum global training error $e_3(k, \lambda_l)$

Case 4: find global $\{k_{best}, \sigma_{best}, \lambda_{best}\}$ with the minimum global training error $e_4(k, \sigma_m, \lambda_l)$

Algorithm 11.4. Testing phase of a local learning regularization network with Cholesky.

Algorithm 11.4: local RN testing with Cholesky

1 For each testing point $\mathbf{x}_n$ ($n = 1,\ldots,N$)

1.1 set $k = k_{best}$ and find its k-nearest training points $k\text{-NN}(\mathbf{x}_n) = \{\mathbf{x}_j, y_j\}_{j=1}^{kbest}$

1.2 compute cached distance matrix $\mathbf{\Delta}^{k\times k}$ with $\delta_{ij} = ||\mathbf{x}_i - \mathbf{x}_j||^2$, $\mathbf{x}_i,\mathbf{x}_j \in k\text{-NN}(\mathbf{x}_n)$

——————— Case 1 (with global k_{best}) ———————

1.3 For each candidate width value σ_m

1.3.1 compute cached local kernel matrix $\mathbf{K}$ with entries $k_{ij} = \exp(-\delta_{ij} / \sigma_m^2)$

1.3.3 For each candidate regularization value λ_l

1.3.3.1 perform Cholesky on $(\mathbf{K}+\lambda_l) = \mathbf{R}^T\mathbf{R}$ and set $\mathbf{S} = \mathbf{R}^{-1}$

1.3.3.2 solve $\mathbf{R}^T\mathbf{R}\ \mathbf{w} = \mathbf{y}$ for the local weight vector $\mathbf{w}$

1.3.3.2 find local $E_{Loo}(\sigma_m, \lambda_l) = (1/k)\sum_{i=1}^{k} (w_i / g_{ii})^2$ where $g_{ii} = \sum_{j=i}^{k} s_{ij}^2$

1.3.3.2 store the local weight vector $\mathbf{w}(\sigma_m, \lambda_l) = \mathbf{w}$

1.4 find local $\{\sigma_{best}, \lambda_{best}\}$ and weights $\mathbf{w}(\sigma_{best}, \lambda_{best})$ with the lowest $E_{Loo}(\sigma_m, \lambda_l)$

1.5 set $\mathbf{h} = [h_1,\ldots,h_{k_{best}}]$ with $h_j = \exp(-||\mathbf{x}_n - \mathbf{x}_j||^2 / \sigma_{best}^2)$

1.6 predict $f_k(\mathbf{x}_n) = \mathbf{h}\ \mathbf{w}(\sigma_{best}, \lambda_{best})$ and update $e_1 = e_1 + (y_n - f_k(\mathbf{x}_n))^2$

——————— Case 2 (with global k_{best}, σ_{best}) ———————

1.3 compute cached local kernel matrix $\mathbf{K}$ with entries $k_{ij} = \exp(-\delta_{ij} / \sigma_{best}^2)$

1.4 For each candidate regularization value λ_l

1.4.1 perform Cholesky on $(\mathbf{K}+\lambda_l) = \mathbf{R}^T\mathbf{R}$ and set $\mathbf{S} = \mathbf{R}^{-1}$

1.4.2 solve $\mathbf{R}^T\mathbf{R}\ \mathbf{w} = \mathbf{y}$ for the local weight vector $\mathbf{w}$

1.4.3 find local $E_{Loo}(\lambda_l) = (1/k)\sum_{i=1}^{k} (w_i / g_{ii})^2$ where $g_{ii} = \sum_{j=i}^{k} s_{ij}^2$

1.4.4 store the local weight vector $\mathbf{w}(\lambda_l) = \mathbf{w}$

1.5 find local λ_{best} and weights $\mathbf{w}(\lambda_{best})$ with the lowest $E_{Loo}(\lambda_l)$

1.6 set $\mathbf{h} = [h_1,\ldots,h_{k_{best}}]$ with $h_j = \exp(-||\mathbf{x}_n - \mathbf{x}_j||^2 / \sigma_{best}^2)$

1.7 predict $f_k(\mathbf{x}_n) = \mathbf{h}\ \mathbf{w}(\lambda_{best})$ and update $e_2 = e_2 + (y_n - f_k(\mathbf{x}_n))^2$

——————— Case 3 (with global k_{best}, λ_{best}) ———————

1.3 For each candidate width value σ_m

1.3.1 compute cached local kernel matrix $\mathbf{K}$ with entries $k_{ij} = \exp(-\delta_{ij} / \sigma_m^2)$

1.3.2 perform Cholesky on $(\mathbf{K}+\lambda_{best}) = \mathbf{R}^T\mathbf{R}$ and set $\mathbf{S} = \mathbf{R}^{-1}$

1.3.3 solve $\mathbf{R}^T\mathbf{R}\ \mathbf{w} = \mathbf{y}$ for the local weight vector $\mathbf{w}$

1.3.4 find local $E_{Loo}(\sigma_m) = (1/k)\sum_{i=1}^{k} (w_i / g_{ii})^2$ where $g_{ii} = \sum_{j=i}^{k} s_{ij}^2$

1.3.5 store the local weight vector $\mathbf{w}(\sigma_m) = \mathbf{w}$

1.4 find local σ_{best} and best local weights $\mathbf{w}(\sigma_{best})$ with the lowest $E_{Loo}(\sigma_m)$

1.5 set $\mathbf{h} = [h_1,\ldots,h_{k_{best}}]$ with $h_j = \exp(-||\mathbf{x}_n - \mathbf{x}_j||^2 / \sigma_{best}^2)$

1.6 predict $f_k(\mathbf{x}_n) = \mathbf{h}\ \mathbf{w}(\sigma_{best})$ and update $e_3 = e_3 + (y_n - f_k(\mathbf{x}_n))^2$

——————— Case 4 (with global k_{best}, σ_{best}, λ_{best}) ———————

1.3 compute local kernel matrix $\mathbf{K}$ with entries $k_{ij} = \exp(-\delta_{ij} / \sigma_{best}^2)$

1.4 perform Cholesky on $(\mathbf{K}+\lambda_{best}) = \mathbf{R}^T\mathbf{R}$

1.5 solve $\mathbf{R}^T\mathbf{R}\ \mathbf{w} = \mathbf{y}$ for the local weight vector $\mathbf{w}$

1.6 set $\mathbf{h} = [h_1,\ldots,h_{k_{best}}]$ with $h_j = \exp(-||\mathbf{x}_n - \mathbf{x}_j||^2 / \sigma_{best}^2)$, ($j = 1,\ldots, k_{best}$)

1.7 predict $f_k(\mathbf{x}_n) = \mathbf{h}\ \mathbf{w}$ and update $e_4 = e_4 + (y_n - f_k(\mathbf{x}_n))^2$

Algorithm 11.5: Pre-computed stored local learning models

Training: For each training point $\mathbf{x}_n$ pre-compute and store the best local model $f_{k,n}(\mathbf{x}_n)$ using the k-NN($\mathbf{x}_n$) list as a local training set in order to optimize the local parameters, either case 1 {local σ, local λ}, or case 2 {local λ}, or case 3 {local σ}, or case 4.

Testing: For each testing query $\mathbf{q}$ find the closest $\mathbf{x}_n$ and the corresponding stored local model $f_{k,n}(\mathbf{x}_n)$ and then use this model to predict the output $f_{k,n}(\mathbf{q})$.

References for chapter 11

[1] Bishop CM (1995) Neural networks for pattern recognition. Oxford University Press, Oxford, UK

[2] Poggio T, Girosi F (1990) Regularization algorithms for learning that are equivalent to multilayer networks. Science 247: 978–982

[3] Girosi F, Jones M, Poggio T (1995) Regularization theory and neural networks architectures. Neural Computation 7: 219–269

[4] Evgeniou T, Pontil M, Poggio T (2000) Regularization Networks and Support Vector Machines. Advances in Computational Mathematics 13: 1–50

[5] Kashima H, Ide T, Kato T, Sugiyama M (2009) Recent Advances and Trends in Large-scale Kernel Methods. IEICE Transactions on Information and systems E92-D, (7): 1338–1353

[6] Bottou L, Vapnik V (1992) Local learning algorithms. Neural Computation 4(6): 888–900

[7] Vapnik V, Bottou L (1993) Local Algorithms for Pattern Recognition and Dependencies Estimation. Neural Computation 5(6): 893–909

[8] Robins A, Frean M (1998) Local Learning Algorithms for Sequential Tasks in Neural Networks. Journal of Advanced Computational Intelligence and Intelligent Informatics, 2(6): 221–227

[9] Vijayakumar S, Schaal S (1998) Local Adaptive Subspace Regression. Neural Processing Letters, 7(3): 139–149

[10] Vijayakumar S, Schaal S (2000) Locally Weighted Projection Regression: An O(n) Algorithm for Incremental Real Time Learning in High Dimensional Space. In: ACM proceedings of the 17th International conference on Machine Learning (ICML2000), pp 1079–1086

[11] Zhou D, Bousquet O, Lal TN, Weston J, Schölkopf B (2004) Learning with local and global consistency. In: Advances in Neural Information Processing Systems 16, MIT Press, pp 321–328

[12] Wu M, Schölkopf B (2007) Transductive Classification via Local Learning Regularization. In: Proceedings of the Eleventh International Conference on Artificial Intelligence and Statistics

[13] Wu M, Yu K, Yu S, Schölkopf B (2007) Local Learning Projections. In: ACM Proceedings of the 24th International Conference on Machine Learning (ICML2007), pp 1039–1046

[14] Zhang H, Berg AC, Maire M, Malik J (2006) SVM-KNN: Discriminative nearest neighbor classification for visual category recognition. In: proceedings of the IEEE Computer Society Conference on Computer Vision and Pattern Recognition, 2, pp 2126–2136

[15] Blanzieri E, Melgani F (2006) An adaptive SVM nearest neighbor classifier for remotely sensed imagery. In: proceedings of the IEEE International Conference on Geoscience and Remote Sensing Symposium (IGARSS 06), pp 3931–3934

[16] Blanzieri E, Melgani F (2008) Nearest neighbor classification of remote sensing images with the maximal margin principle. IEEE Transactions on Geoscience and Remote Sensing, 46(6): 1804–1811

[17] Segata N, Blanzieri E, Delany SJ, Cunningham P (2009) Noise reduction for instance-based learning with a local maximal margin approach. Journal of Intelligent Information Systems 35(2): 301–331

[18] Segata N, Blanzieri E (2010) Fast and scalable local kernel machines. Journal of Machine Learning Research, 11: 1883–1926

[19] Yang T, Kecman V (2010) Face recognition with adaptive local hyperplane algorithm. Pattern Analysis & Applications, 13(1): 79–83

[20] Cheng H, Tan P-N, Jin R (2010) Efficient algorithm for localized support vector machine. IEEE Transactions on Knowledge and Data Engineering, 22(4): 537–549

[21] Zakai A, Ritov Y (2009) Consistency and localizability. Journal of Machine Learning Research, 10:827–856

[22] Hable R (2013) Universal Consistency of Localized Versions of Regularized Kernel Methods. Journal of Machine Learning Research 14: 153–186

[23] Kokkinos Y, Margaritis KG (2013) Parallel and local learning for fast Probabilistic Neural Networks in scalable data mining. In: ACM proceedings of the 6th Balkan Conference in Informatics (BCI 2013), pp 47–52

[24] Yu A, Grauman K (2014) Predicting Useful Neighborhoods for Lazy Local Learning. In: Neural Information Processing Systems (NIPS 2014) pp 1916–1924

[25] Zhang J, Feng L, Wu B (2016) Local extreme learning machine: local classification model for shape feature extraction. Neural Computing and Applications, DOI 10.1007/s00521-015-2008-7

[26] Kokkinos Y, Margaritis KG (2015) Multithreaded Local Learning Regularization Neural Networks for regression tasks. In: proceedings of the 16th International Conference on Engineering Applications of Neural Networks (EANN 2015) pp 129–138

[27] Schölkopf B, Smola AJ (2002) Learning with kernels: Support vector machines, regularization, optimization, and beyond. MIT Press, Cambridge, MA

[28] Rifkin R, Yeo G, Poggio T (2003) Regularized least-squares classification, Nato Science Series Sub Series III Computer and Systems Sciences 190: 131–154

[29] Cawley GC, Talbot NLC (2007) Preventing Over-Fitting during Model Selection via Bayesian Regularisation of the Hyper-Parameters. Journal of Machine Learning Research 8: 841–861

[30] Rifkin RM, Lippert RA (2007) Notes on regularized least squares. Technical report, MIT Computer Science and Artificial Intelligence Laboratory

[31] Golub GH, van Loan CF (1996) Matrix Computations, 3rd ed., John Hopkins University Press, Baltimore, MD

[32] Press WH, Teukolsky SA, Vetterling WT, Flannery BP (2002) Numerical Recipes in C++: The Art of Scientific Computing. 2nd ed., Cambridge University Press

[33] Buyya R (1999) High Performance Cluster Computing: Programming and Applications, 2. Prentice Hall, New Jersey

[34] García S, Fernández A, Luengo J, Herrera F, (2010) Advanced nonparametric tests for multiple comparisons in the design of experiments in computational intelligence and data mining: Experimental analysis of power, Information Science 180: 2044–2064

12 Conclusions and Future Work

The thesis focuses on Parallel and Distributed Neural Networks and Machine Learning Schemes.

12.1 Conclusions

In chapter 3 we analyzed scalable algorithms for Extreme Learning Machine (ELM) model selection via Cholesky, SVD, QR and Eigen decompositions in the computational steps inside either k-fold cross-validation or leave-one-out cross-validation. It was found that while the type of matrix decomposition is important for faster executions, equally important is the type of cross-validation. We compared virtual leave-one-out Cross Validation, efficient 10-fold Cross-Validation, and scalable 10-fold Cross-Validation version. We found that the last version can save a huge amount of computational time. The combinations for ELM cross-validation are presented in 7 representative algorithms. By examining all the required matrix-matrix multiplications and matrix-vector multiplications we identify the most expensive term that is dominant in the computational steps of all the algorithms except the last one, where this term is absent, and this is the main reason why it is the fastest. Hence we can conclude that the most promising combination in terms of scalability is the scalable 10-fold Cross-Validation with Eigen Value Decomposition.

In chapter 4 we studied the parallelised Incremental ELMs which add neurons one-by-one and proceed via vector-vector computations and not matrix-matrix. They use one vector at a time and they do not require a full matrix in memory. We proposed the Enhanced Convex Incremental ELM (ECI-ELM) algorithm which is a combination of the existing Enhanced Incremental ELM (EI-ELM) and Convex Incremental ELM (CI-ELM). We show that the proposed ECI-ELM is also highly parallelized and furthermore outperforms the existing I-ELM, CI-ELM and EI-ELM for regression tasks. ECI-ELM has the lowest error rate curve and converges much faster than the other algorithms. From the simulations we conclude that ECI-ELM is fully scalable when operates in a parallel system of distributed memory workstations.

In chapter 5 we presented a ring pipeline parallel parsimonious Probabilistic Neural Network (PNN) architecture where each processor holds a portion of data and a portion of neurons. The new PNN training is effective and by combining the proposed leave-one-out kernel averaged gradient descent with Subtractive Clustering and Expectation Maximization creates the PNN model automatically, without the need of any user-defined parameter. We show how this new scheme can be efficiently operate in the ring pipeline, which minimizes the communication delays. Several experimental simulations reveal that the larger datasets have parallel speedup curves very close to linear and efficiencies very close to one.

In chapter 6 we demonstrated that a ring pipeline parallel Radial Basis Function Neural Network for regression tasks can be designed by first mapping the recently proposed leave-one-out Kernel Averaged Gradient descent Subtractive Clustering (KG-SC) for automatically selecting appropriate RBF centers with their number and then mapping

a mini-batch gradient descent for refining all the RBF network parameters. Here again the RBFNN model is created automatically since no user-defined parameters are required. The proposed scheme is scalable and delivers parallel speedups close to the ideal case.

In chapter 7 we described a completely asynchronous, scalable and distributed privacy-preserving Regularization Network (RN) committee machine of isolated Regularization Network classifiers that avoids the gathering of all data to a single location. The validation set for one classifier becomes the training set of the other and vice versa. A coarse-grained asymmetric mutual validation matrix is formed to map all the committee machine members. The proposed method manages to compute the weights for the proposed RN committee, while preserving the privacy and outperforming majority voting, simple weighted average and stacked generalization in most of the cases.

In chapter 8 we focused on ensemble selection of neural network classifiers for distributed data mining. A new method called Confidence ratio Affinity Propagation was presented. It selects the best neural network classifiers of the ensemble in an asynchronous distributed and privacy-preserving computing cycle. We concluded that the proposed scheme produced promising results as compared with other pruning methods, such as reduce-error pruning, Kappa pruning, orientation pruning and JAM's diversity pruning. The method does not need to predefine the number of classifiers or to monitor the ensemble performance on a separate validation set.

In chapter 9 we dealt with a probabilistic neural network (PNN) committee machine composed of class-specialized regularization network classifiers locally trained on each Peer. The committee training takes into account the asynchronous distributed and privacy-preserving requirements that can be met in P2P systems. An asynchronous distributed computing P2P cycle is executed to construct a mutual validation matrix. From this matrix a weight based ensemble selection of best regularization network members for every class can be performed, based on regularized non-negative weighting of the class-specialized members. This improves significantly the accuracy and speed of the PNN committee.

In chapter 10 we presented the Hierarchical Markovian Radial Basis Function Neural Network (HiMarkovRBFNN) classifier model that enables recursive operations. The Hierarchical Learning structure is a nested system that allows data access in a recursive fashion. It was derived from a classical divide-and-conquer parallel programming strategy that can use tree-based message passing (recursive) to create one hierarchical model. All hidden neurons in the hierarchy levels are composed of truly RBF Neural Networks with two weight matrices, in contrast to the simple summation neurons with only linear weighted combinations which are usually encountered in ensemble models and cascading networks. Thus the operation is exactly the same at all levels. Comparisons with classical meta-learning architectures, namely Committee Machines and Cascaded Machines, revealed that the proposed method produces better results and compares well as a combiner that can merge many HiMarkovRBFNN child nodes into one higher level parent HiMarkovRBFNN.

In chapter 11 we considered Local learning versions of Regularization Networks (RN) and proposed several options for improving their online prediction performance, both in accuracy and speed. By exploiting the interplay between locally optimized and globally optimized parameters, it reduces the optimization time of the hyper-

parameters that are needed at runtime (online). While Eigen decomposition is suitable for validating cost–effectively several regularization parameters, Cholesky decomposition should be preferred when validating several neighbourhood sizes (the number of k-nearest neighbours) as well as when the local network operates online. Parallelism in a multi-core system for these local computations demonstrates that their execution times can be further reduced. It was also demonstrated that there is a substantial gain in waiting for a testing point to arrive before building a local model, and hence the online local learning RNs were more accurate than their pre-computed stored local models.

12.2 Future work

Some ideas for future research can be derived. From chapter 3 the Scalable cross-validation versions deserve further consideration and there is future merit in this observation, since other algorithms like incomplete principal Cholesky or non-negative matrix factorizations or regularized orthogonal least squares can be potentially used for guiding another important aspect that is the network pruning. Future experiments could explore such a possibility. From chapter 5 which uses the proposed Kernel Gradient Subtractive Clustering (KG-SC) algorithm for PNN training, an interesting future work could explore the application of KG-SC in the Regularization Network for creating a low rank approximation of the kernel matrix. From chapter 6 we can explore in the future the employment of the mini-batch parallel KG-SC with the mini-batch parallel gradient descent for training in the same way a multi-output RBFNN classifier. From chapter 7 we could investigate an extension of the RN committee machine in the case of multi-task learning. From chapter 8 the proposed confidence ratio affinity propagation can possibly be extended as an on-line algorithm for dynamic ensemble selection. From chapter 9 the weight base ensemble selection by using non-negative regularized least squares could be examined in the context of other models, like multi-label classification, that similarly require pruning. From chapter 10 a hierarchical neural network which adapts into a given structure of nested clusters could be of assistance in hierarchical rule extraction. Beyond the scalability advantages and the straightforward parallel operation, the presented method could be proven useful in classifying highly irregular datasets. From chapter 11 an interesting future work, given that Cholesky is suitable for parallel implementation using Graphic processing units (GPUs), could be that of GPU accelerated online local learning RN for classification.