Studies in Big Data 7

Noel Lopes
Bernardete Ribeiro

Machine Learning for Adaptive Many-Core Machines – A Practical Approach

Springer

Studies in Big Data

Volume 7

Series editor

Janusz Kacprzyk, Polish Academy of Sciences, Warsaw, Poland
e-mail: kacprzyk@ibspan.waw.pl

For further volumes:
http://www.springer.com/series/11970

About this Series

The series "Studies in Big Data" (SBD) publishes new developments and advances in the various areas of Big Data- quickly and with a high quality. The intent is to cover the theory, research, development, and applications of Big Data, as embedded in the fields of engineering, computer science, physics, economics and life sciences. The books of the series refer to the analysis and understanding of large, complex, and/or distributed data sets generated from recent digital sources coming from sensors or other physical instruments as well as simulations, crowd sourcing, social networks or other internet transactions, such as emails or video click streams and other. The series contains monographs, lecture notes and edited volumes in Big Data spanning the areas of computational intelligence incl. neural networks, evolutionary computation, soft computing, fuzzy systems, as well as artificial intelligence, data mining, modern statistics and Operations research, as well as self-organizing systems. Of particular value to both the contributors and the readership are the short publication timeframe and the world-wide distribution, which enable both wide and rapid dissemination of research output.

Noel Lopes · Bernardete Ribeiro

Machine Learning for Adaptive Many-Core Machines – A Practical Approach

Springer

Noel Lopes
Polytechnic Institute of Guarda
Guarda
Portugal

Bernardete Ribeiro
Department of Informatics Engineering
Faculty of Sciences and Technology
University of Coimbra, Polo II
Coimbra
Portugal

ISSN 2197-6503 ISSN 2197-6511 (electronic)
ISBN 978-3-319-06937-1 ISBN 978-3-319-06938-8 (eBook)
DOI 10.1007/978-3-319-06938-8
Springer Cham Heidelberg New York Dordrecht London

Library of Congress Control Number: 2014939947

Printed on acid-free paper

Springer is part of Springer Science+Business Media (www.springer.com)

To Sara and Pedro
To my family
Noel Lopes

To Miguel and Alexander
To my family
Bernardete Ribeiro

Preface

Motivation and Scope

Today the increasing complexity, performance requirements and cost of current (and future) applications in society is transversal to a wide range of activities, from science to business and industry. In particular, this is a fundamental issue in the Machine Learning (ML) area, which is becoming increasingly relevant in a wide diversity of domains. The scale of the data from Web growth and advances in sensor data collection technology have been rapidly increasing the magnitude and complexity of tasks that ML algorithms have to solve.

Much of the data that we are generating and capturing will be available "indefinitely" since it is considered a strategic asset from which useful and valuable information can be extracted. In this context, Machine Learning (ML) algorithms play a vital role in providing new insights from the abundant streams and increasingly large repositories of data. However, it is well-known that the computational complexity of ML methodologies, often directly related with the amount of data, is a limiting factor that can render the application of many algorithms to real-world problems impractical. Thus, the challenge consists of processing such large quantities of data in a realistic (useful) time frame, which drives the need to extend the applicability of existing ML algorithms and to devise parallel algorithms that scale well with the volume of data or, in other words, can handle "Big Data".

This volume takes a practical approach for addressing this problematic, by presenting ways to extend the applicability of well-known ML algorithms with the help of high-scalable Graphics Processing Unit (GPU) parallel implementations. Modern GPUs are highly parallel devices that can perform general-purpose computations, yielding significant speedups for many problems in a wide range of areas. Consequently, the GPU, with its many cores, represents a novel and compelling solution to tackle the aforementioned problem, by providing the means to analyze and study larger datasets.

Rationally, we can not view the GPU implementations of ML algorithms as a universal solution for the "Big Data" challenges, but rather as part of the answer, which may require the use of different strategies coupled together. In this perspective, this volume addresses other strategies, such as using instance-based selection methods to choose a representative subset of the original training data, which can in turn be used to build models in a fraction of the time needed to derive a model from the complete dataset. Nevertheless, large scale datasets and data streams may require learning algorithms that scale roughly linearly with the total amount of data. Hence, traditional batch algorithms may not be up to the challenge and therefore the book also addresses incremental learning algorithms that continuously adjust their models with upcoming new data. These embody the potential to handle the gradual concept drifts inherent to data streams and non-stationary dynamic databases.

Finally, in practical scenarios, the awareness of handling large quantities of data is often exacerbated by the presence of incomplete data, which is an unavoidable problem for most real-world databases. Therefore, this volume also presents a novel strategy for dealing with this ubiquitous problem that does not affect significantly either the algorithms performance or the preprocessing burden.

The book is not intended to be a comprehensive survey of the state-of-the-art of the broad field of Machine Learning. Its purpose is less ambitious and more practical: to explain and illustrate some of the more important methods brought to a practical view of GPU-based implementation in part to respond to the new challenges of the Big Data.

Plan and Organization

The book comprehends nine chapters and one appendix. The chapters are organized into four parts: the first part relating to fundamental topics in Machine Learning and Graphics Processing Units encloses the first two chapters; the second part includes four chapters and gives the main supervised learning algorithms, including methods to handle missing data and approaches for instance-based learning; the third part with two chapters concerns unsupervised and semi-supervised learning approaches; in the fourth part we conclude the book with a summary of many-core algorithms approaches and techniques developed across this volume and give new trends to scale up algorithms to many-core processors. The self-contained chapters provide an enlightened view of the interplay between ML and GPU approaches.

Chapter 1 details the Machine Learning challenges on Big Data, gives an overview of the topics included in the book, and contains background material on ML formulating the problem setting and the main learning paradigms.

Chapter 2 presents a new open-source GPU ML library (GPU Machine Learning Library – GPUMLib) that aims at providing the building blocks for the development of efficient GPU ML software. In this context, we analyze the potential of the GPU in the ML area, covering its evolution. Moreover, an overview of the existing ML

GPU parallel implementations is presented and we argue for the need of a GPU ML library. We then present the CUDA (Compute Unified Device Architecture) programming model and architecture, which was used to develop GPU Machine Learning Library (GPUMLib) and we detail its architecture.

Chapter 3 reviews the fundamentals of Neural Networks, in particular, the multi-layered approaches and investigates techniques for reducing the amount of time necessary to build NN models. Specifically, it focuses on details of a GPU parallel implementation of the Back-Propagation (BP) and Multiple Back-Propagation (MBP) algorithms. An Autonomous Training System (ATS) that reduces significantly the effort necessary for building NN models is also discussed. A practical approach to support the effectiveness of the proposed systems on both benchmark and real-world problems is presented.

Chapter 4 analyses the treatment of missing data and alternatives to deal with this ubiquitous problem generated by numerous causes. It reviews missing data mechanisms as well as methods for handling Missing Values (MVs) in Machine Learning. Unlike pre-processing techniques, such as imputation, a novel approach Neural Selective Input Model (NSIM) is introduced. Its application on several datasets with both different distributions and proportion of MVs shows that the NSIM approach is very robust and yields good to excellent results. With the scalability in mind a GPU paralell implementation of Neural Selective Input Model (NSIM) to cope with Big Data is described.

Chapter 5 considers a class of learning mechanisms known as the Support Vector Machines (SVMs). It provides a general view of the machine learning framework and describes formally the SVMs as large margin classifiers. It explores the Sequential Minimal Optimization (SMO) algorithm as an optimization methodology to solve an SVM. The rest of the chapter is dedicated to the aspects related to its implementation in multi-thread CPU and GPU platforms. We also present a comprehensive comparison of the evaluation methods on benchmark datasets and on real-world case studies. We intend to give a clear understanding of specific aspects related to the implementation of basic SVM machines in a many-core perspective. Further deployment of other SVM variants are essential for Big Data analytics applications.

Chapter 6 addresses incremental learning algorithms where the models incorporate new information on a sample-by-sample basis. It introduces a novel algorithm the Incremental Hypersphere Classifier Incremental Hypersphere Classifier (IHC) which presents good properties in terms of multi-class support, complexity, scalability and interpretability. The IHC is tested in well-known benchmarks yielding good classification performance results. Additionally, it can be used as an instance selection method since it preserves class boundary samples. Details of its application to a real case study in the field of bioinformatics are provided.

Chapter 7 deals with unsupervised and semi-supervised learning algorithms. It presents the Non-Negative Matrix Factorization (NMF) algorithm as well as a new semi-supervised method, designated by Semi-Supervised NMF (SSNMF). In addition, this Chapter also covers a hybrid NMF-based face recognition approach.

Chapter 8 motivates for the deep learning architectures. It starts by introducing the Restricted Boltzmann Machines (RBMs) and the Deep Belief Networks (DBNs) models. Being unsupervised learning approaches their importance is shown in multiple facets specifically by the feature generation through many layers, contrasting with shallow architectures. We address their GPU parallel implementations giving a detailed explanation of the kernels involved. It includes an extensive experiment, involving the MNIST database of hand-written digits and the HHreco multi-stroke symbol database in order to gain a better understanding of the DBNs.

In the final **Chapter 9** we give an extended summary of the contributions of the book. In addition we present research trends with special focus on the big data and stream computing. Finally, to meet future challenges on real-time big data analysis from thousands of sources new platforms should be exploited to accelerate many-core software research.

Audience

The book is designed for practitioners and researchers in the areas of Machine Learning (ML) and GPU computing (CUDA) and is suitable for postgraduate students in computer science, engineering, information technology and other related disciplines. Previous background in the areas of ML or GPU computing (CUDA) will be beneficial, although we attempt to cover the basics of these topics.

Acknowledgments

We would like to acknowledge and thank all those who have contributed to bringing this book to publication for their help, support and input.

We thank many stimulating user's requirements to include new perspectives in the GPUMLib due to many downloads of the software. It turn out possible to improve and extend many aspects of the library.

We also wish to thank the support of the Polytechnic Institute of Guarda and of the Centre of Informatics and Systems of the Informatics Engineering Department, Faculty of Science and Technologies, University of Coimbra, for the means provided during the research.

Our thanks to Samuel Walter Best who reviewed the syntactic aspects of the book.

Our special thanks and appreciation to our editor, Professor Janusz Kacprzyk, of Studies in Big Data, Springer, for his essential encouragement.

Lastly, to our families and friends for their love and support.

Coimbra, Portugal — Noel Lopes
February 2014 — Bernardete Ribeiro

Contents

Acronyms

API	Application Programming Interface
APU	Accelerated Processing Unit
ATS	Autonomous Training System
BP	Back-Propagation
CBCL	Center for Biological and Computational Learning
CD	Contrastive Divergence
CMU	Carnegie Mellon University
CPU	Central Processing Unit
CUDA	Compute Unified Device Architecture
DBN	Deep Belief Network
DCT	Discrete Cosine Transform
DOS	Denial Of Service
ECG	Electrocardiograph
EM	Expectation-Maximization
ERM	Empirical Risk Minimization
FF	Feed-Forward
FPGA	Field-Programmable Gate Array
FPU	Floating-Point Unit
FRCM	Face Recognition Committee Machine
GPGPU	General-Purpose computing on Graphics Processing Units
GPU	Graphics Processing Unit
GPUMLib	GPU Machine Learning Library
HPC	High-Performance Computing
IB3	Instance Based learning
ICA	Independent Component Analysis
IHC	Incremental Hypersphere Classifier
I/O	Input/Output
KDD	Knowledge Discovery and Data mining
KKT	Karush-Kuhn-Tucker
LDA	Linear Discriminant Analysis
LIBSVM	Library for Support Vector Machines

MAR	Missing At Random
MB	Megabyte(s)
MBP	Multiple Back-Propagation
MCAR	Missing Completely At Random
MDF	Modified Direction Feature
MCMC	Markov Chain Monte Carlo
ME	Mixture of Experts
MFF	Multiple Feed-Forward
MIT	Massachusetts Institute of Technology
ML	Machine Learning
MLP	Multi-Layer Perceptron
MPI	Message Passing Interface
MV	Missing Value
MVP	Missing Values Problem
NMAR	Not Missing At Random
NMF	Non-Negative Matrix Factorization
k-nn	*k*-nearest neighbor
NN	Neural Network
NORM	Multiple imputation of incomplete multivariate data under a normal model
NSIM	Neural Selective Input Model
NSW	New South Wales
OpenCL	Open Computing Language
OpenMP	Open Multi-Processing
PCA	Principal Component Analysis
PVC	Premature Ventricular Contraction
QP	Quadratic Programming
R2L	unauthorized access from a remote machine
RBF	Radial Basis Function
RBM	Restricted Boltzmann Machine
RMSE	Root Mean Square Error
SCOP	Structural Classification Of Proteins
SFU	Special Function Unit
SIMT	Single-Instruction Multiple-Thread
SM	Streaming Multiprocessor
SMO	Sequential Minimal Optimization
SP	Scalar Processor
SRM	Structural Risk Minimization
SSNMF	Semi-Supervised NMF
SV	Support Vector
SVM	Support Vector Machine
U2R	unauthorized access to local superuser privileges
UCI	University of California, Irvine
UKF	Universal Kernel Function
UMA	Unified Memory Access

VA	Ventricular Arrhythmia
VC	Vapnik-Chervonenkis
WVTool	Word Vector Tool

Notation

a_j Activation of the neuron j.
a_i Accuracy of sample i.
$\mathbf{b}$ Bias of the hidden units.
Be Bernoulli distribution.
$\mathbf{c}$ Bias of the visible units.
C Number of classes.
C Penalty parameter of the error term (soft margin).
d Adaptive step size decrement factor.
D Number of features (input dimensionality).
E Error.
f Mapping function.
fn False negatives.
fp False positives.
g Gravity.
$\mathbf{h}$ Hidden units (outputs of a Restricted Boltzmann Machine).
$\mathbf{H}$ Extracted features matrix.
I Number of visible units.
J Number of hidden units.
$\mathbf{K}$ Response indicator matrix.
l Number of layers.
L Lagrangian function.
m Importance factor.
n Number of samples stored in the memory.
N Number of samples.
N' Number of test samples.
p Probability.
P Number of model parameters.
r Number of reduced features (rank).
r Robustness (reducing) factor.
s Number of shared parameters (between models).

$\mathbf{t}$ Targets (desired values).
$\top$ Transpose.
tn True negatives.
tp True positives.
u Adaptive step size increment factor.
$\mathbf{v}$ Visible units (inputs of a Restricted Boltzmann Machine).
$\mathbf{V}$ Input matrix with non-negative coefficients.
$\mathbf{W}$ Weights matrix.
$\mathbf{x}$ Input vector.
$\tilde{x}_i$ Result of the input transformation, performed to the original input x_i.
$\mathbf{X}$ Input matrix.
$\mathbf{y}$ Outputs.
Z Energy partition function (of a Restricted Boltzmann Machine).
α Momentum term.
α_i Lagrange multiplier.
γ Width of the Gaussian RBF kernel.
δ Local gradient.
Δ Change of a model parameter (e.g. ΔW_{ij} is the weight change).
η Learning rate.
θ Model parameter.
κ Response indicator vector.
ξ Missing data mechanism parameter.
ξ_i Slack variables.
ρ Margin.
ρ_i Radius of sample i.
σ Sigmoid function.
ϕ Neuron activation function.
$\mathbb{R}$ Set of real numbers.

Part I
Introduction

Chapter 1
Motivation and Preliminaries

Abstract. In this Chapter the motivation for the setting of adaptive many-core machines able to deal with big machine learning challenges is emphasized. A framework for inference in Big Data from real-time sources is presented as well as the reasons for developing high-throughput Machine Learning (ML) implementations. The chapter gives an overview of the research covered in the book spanning the topics of advanced ML methodologies, the GPU framework and a practical application perspective. The chapter describes the main Machine Learning (ML) paradigms, and formalizes the supervised and unsupervised ML problems along with the notation used throughout the book. Great relevance has been rightfully given to the learning problem setting bringing to solutions that need to be consistent, well-posed and robust. In the final of the chapter an approach to combine supervised and unsupervised models is given which can impart in better adaptive models in many applications.

1.1 Machine Learning Challenges: Big Data

Big Data is here to stay, posing inevitable challenges is many areas and in particular in the ML field. By the beginning of this decade there were already 5 billion mobile phones producing data everyday. Moreover, millions of networked sensors are being routinely integrated into ordinary objects, such as cars, televisions or even refrigerators, which will become an active part in the Internet of Things [146]. Additionally, the deployment (already envisioned) of worldwide distributed ubiquitous sensor arrays for long-term monitoring, will allow mankind to collect previously inaccessible information in real-time, especially in remote and potentially dangerous areas such as the ocean floor or the mountains' top, bringing the dream of creating a "sensors everywhere" infrastructure a step closer to reality. In turn this data will feed computer models which will generate even more data [85].

In the early years of the previous decade the global data produced grew approximately 30% per year [144]. Today, a decade later, the projected growth is already of 40% [146] and this trend is likely to endure, fueled by new technological

N. Lopes and B. Ribeiro, *Machine Learning for Adaptive Many-Core Machines – A Practical Approach,* Studies in Big Data 7,
DOI: 10.1007/978-3-319-06938-8_1,

advances in communication, storage and sensor device technologies. Despite this exponential growth, much of the accumulated data that we are generating and capturing will be made permanently available for the purposes of continued analysis [85]. In this context, data is an asset *per se*, from which useful and valuable information can be extracted. Currently, ML algorithms and in particular supervised learning approaches play the central role in this process [155].

Figure 1.1 illustrates in part how ML algorithms are an important component of this knowledge extraction process. The block diagram gives a schematic view of the intreplay between the different phase involved.

1. The phenomenal growth of the Internet and the availability of devices (laptops, mobile phones, etc.) and low-cost sensors and devices capable of capturing, storing and sharing information anytime and anywhere, have led to an abundant wealth of data sources.
2. In the scientific domain, this "real" data can be used to build sophisticated computer simulation models, which in turn generate additional (artificial) data.
3. Eventually, some of the important data, within those stream sources, will be stored in persistent repositories.
4. Extracting useful information from these large repositories of data using ML algorithms is becoming increasingly important.
5. The resulting ML models will be a source of relevant information in several areas, which help to solve many problems.

The need for gaining understanding of the information contained in large and complex datasets is common to virtually all fields, ranging from business and industry to science and engineering. In particular, in the business world, the corporate and customer data are already recognized as a strategic resource from which invaluable competitive knowledge can be obtained [47]. Moreover, science is gradually moving towards being computational and data centric [85].

However, using computers in order to gain understanding from the continuous streams and the increasingly large repositories of data is a daunting task that may likely take decades, as we are at an early stage of a new "data-intensive" science paradigm. If we are to achieve major breakthroughs, in science and other fields, we need to embrace a new data-intensive paradigm where "data scientists" will work side-by-side with disciplinary experts, inventing new techniques and algorithms for analyzing and extracting information from the huge amassed volumes of digital data [85].

Over the last few decades, ML algorithms have steadily been the source of many innovative and successful applications in a wide range of areas (e.g. science, engineering, business and medicine), encompassing the potential to enhance every aspect of lives [6, 153]. Indeed, in many situations, it is not possible to rely exclusively on human perception to cope with the high data acquisition rates and the large volumes of data inherent to many activities (e.g. scientific observations, business transactions) [153].

As a result, we are increasingly relying on Machine Learning (ML) algorithms to extract relevant and context useful information from data. Therefore, our

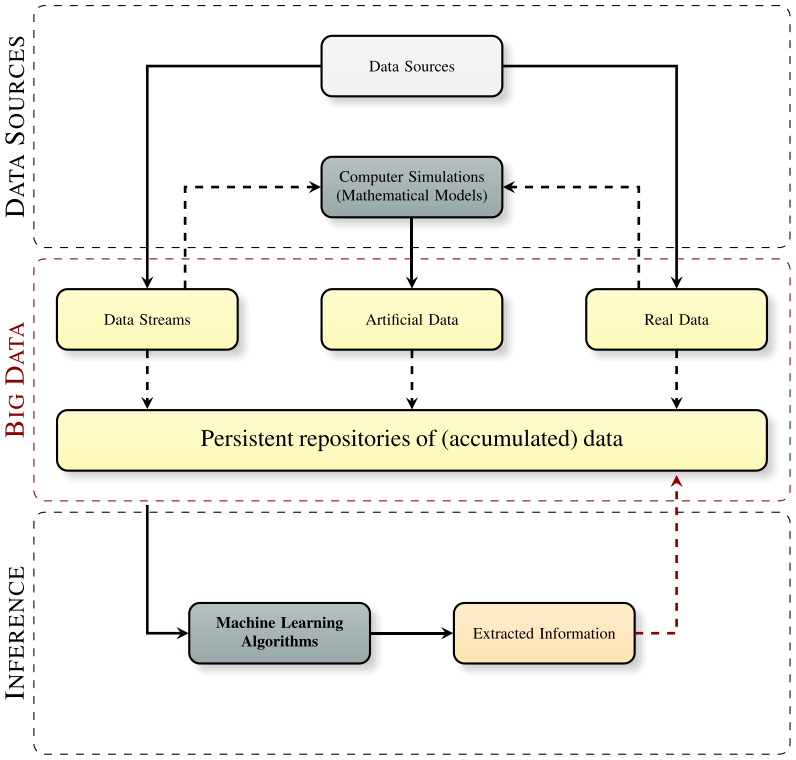


Fig. 1.1 Using Machine Learning (ML) algorithms to extract information from data

unprecedented capacity to generate, capture and share vast amounts of high-dimensional data increases substantially the magnitude and complexity of ML tasks. However, it is well known that the computational complexity of ML methodologies, often directly related with the amount of the training data, is a limiting factor that can render the application of many algorithms to real-world problems, involving large datasets, impractical [22, 69]. Thus, the challenge consists of processing large quantities of data in a realistic time frame, which subsequently drives the need to extend the applicability of existing algorithms to larger datasets, often encompassing complex and hard to discover relationships, and to devise parallel algorithms that scale well enough with the volume of data.

Manyika et al. attempted to present a subjective definition for the Big Data problem – Big Data refers to datasets whose size is beyond the ability of typical tools to process – that is particularly pertinent in the ML field [146]. Hence, several factors might influence the applicability of ML methods [13]. These are depicted in Figure 1.2 which schematically structures the main reasons for the development of high-throughput implementations.

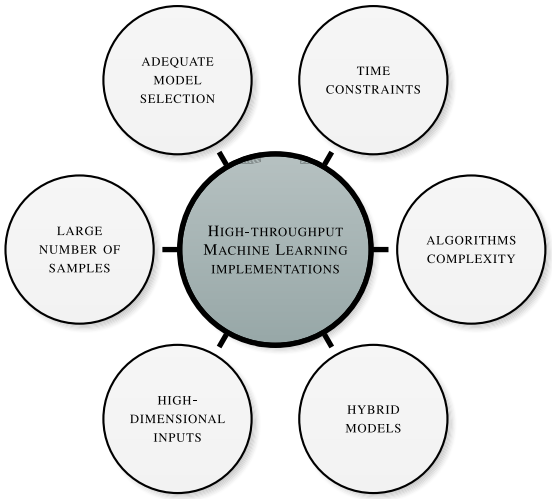


Fig. 1.2 Reasons for developing high-throughput Machine Learning (ML) implementations

Naturally, the primary reasons pertain the computational complexity of ML algorithms and the need to explore big datasets encompassing a large number of samples and/or features. However, there are other factors which demand for high-throughput algorithms. For example, in practical scenarios, obtaining first-class models requires building (training) and testing several distinct models using different architectures and parameter configurations. Often cross-validation and grid-search methods are used to determine proper model architectures and favorable parameter configurations. However, these methods can be very slow even for relatively small datasets, since the training process must be repeated several times according to the number of different architecture and parameter combinations. Incidentally, the increasing complexity of ML problems often result in multi-step hybrid systems encompassing different algorithms. The rationale consists of dividing the original problem into simpler and more manageable subproblems. However, in this case, the cumulative time of creating each individual model must be considered. Moreover, the end result of aggregating several individual models does not always meet the expectations, in which case we may need to restart the process, possibly using different approaches. Finally, another reason has to do with the existence of time constraints, either for building the model and/or for obtaining the inference results. Regardless of the reasons for scaling up ML algorithms, building high-throughput implementations will ultimately lead to improved ML models and to the solution of otherwise impractical problems.

Although new technologies, such as GPU parallel computing, may not provide a complete solution for this problem, its effective application may account for significant advances in dealing with problems that would otherwise be impractical to solve [85]. Modern GPUs are highly parallel devices that can perform general-

purpose computations, providing significant speedups for many problems in a wide range of areas. Consequently, the GPU, with its many cores, represents a novel and compelling solution to tackle the aforementioned problem, by providing the means to analyze and study larger datasets [171, 197]. Notwithstanding, parallel computer programs are by far more difficult to design, write, debug and fine-tune than their sequential counterparts [85]. Moreover, the GPU programming model is significantly different from the traditional models [71, 171]. As a result, few ML algorithms have been implemented on the GPU and most of them are not openly shared, posing difficulties for those aiming to take advantage of this architecture. Thus, the development of an open-source GPU ML library could mitigate this problem and promote cooperation within the area. The objective is two-fold: (*i*) to reduce the effort of implementing new GPU ML software and algorithms, therefore contributing to the development of innovative applications; (*ii*) to provide functional GPU implementations of well-known ML algorithms that can be used to reduce considerably the time needed to create useful models and subsequently explore larger datasets.

Rationally, we can not view the GPU implementations of ML algorithms as a universal solution for the Big Data challenges, but rather as part of the answer, which may require the use of different strategies coupled together. For instance, the careful design of semi-supervised algorithms may result not only in faster methods but also in models with improved performance. Another strategy consists of using instance selection methods to choose a representative subset of the original training data, which can in turn be used to build models in a fraction of the time needed to derive a model from the complete dataset. Nevertheless, large scale datasets and data streams may require learning algorithms that scale roughly linearly with the total amount of data [22]. Hence, traditional batch algorithms may not be up to the challenge and instead we must rely on incremental learning algorithms [96] that continuously adjust their models with upcoming new data. These embody the potential to handle the gradual concept drifts inherent to data streams and non-stationary dynamic databases.

Finally, in practical scenarios, the problem of handling large quantities of data is often exacerbated by the presence of incomplete data, which is an unavoidable problem for most real-world databases [105, 102]. Therefore, it is important to devise strategies to deal with this ubiquitous problem that does not affect significantly either the algorithms performance or the preprocessing burden.

This book, which is based on the PhD thesis of the first author, tackles the aforementioned problems, by making use of two complementary components: a body of novel ML algorithms and a set of high-performance ML parallel implementations for adaptive many-core machines. Specifically, it takes a practical approach, presenting ways to extend the applicability of well-known ML algorithms with the help of high-scalable GPU parallel implementations. Moreover, it covers new algorithms that scale well in the presence of large amounts of data. In addition, it tackles the missing data problem, which often occurs in large databases. Finally, a computational framework GPUMLib for implementing these algorithms is present.

1.2 Topics Overview

The contents of this book, predominantly focus on techniques for scaling up supervised, unsupervised and semi-supervised learning algorithms using the GPU parallel computing architecture. However, other topics such as incremental learning or handling missing data, related to the goal of extending the applicability of ML algorithms to larger datasets are also addressed. The following gives an overview of the main topics covered throughout the book:

- **Advanced Machine Learning (ML) Topics**
  - A new **adaptive step size technique for RBMs** that improves considerably their training convergence, thereby significantly reducing the time necessary to achieve a good reconstruction error. The proposed technique effectively decreases the training time of RBMs and consequently of **Deep Belief Networks (DBNs)**. Additionally, at each iteration the technique seeks to find the near-optimal step sizes, solving the problem of finding an adequate and suitable learning rate for training the networks.
  - A new **Semi-Supervised Non-Negative Matrix Factorization (SSNMF)** algorithm that reduces the computational cost of the original Non-Negative Matrix Factorization (NMF) method while improving the accuracy of the resulting models. The proposed approach aims at extracting the most unique and discriminating characteristics of each class, increasing the models classification performance. Identifying the particular characteristics of each individual class is manifestly important when dealing with unbalanced datasets where the distinct characteristics of minority classes may be considered noise by traditional NMF approaches. Moreover, SSNMF creates sparser matrices, which potentially results in reduced storage requirements and improved interpretation of their factors.
  - A novel instance-based **Incremental Hypersphere Classifier (IHC)** learning algorithm, which presents advantageous properties in terms of multi-class support, scalability and interpretability, while providing good classification results. The IHC is highly-scalable, since it can accommodate memory and computational restrictions, creating the best possible model according to the amount of resources given. A key feature of this algorithm lies in its ability to update models and classify new data in real-time. Moreover, IHC is prepared to deal with concept-drift scenarios and can be used as an instance selection method, since it tries to preserve the class boundary samples while removing inaccurate/noisy samples.
  - A novel **Neural Selective Input Model (NSIM)** which provides a novel strategy for directly handling Missing Values (MVs) in Neural Networks (NNs). The proposed technique accounts for the creation of different transparent and bound conceptual NN models instead of relying on tedious data preprocessing techniques, which may inadvertently inject outliers into the data. The projected solution presents several advantages as compared to traditional methods for handling MVs, making this a first-class method

for dealing with this crucial problem. Moreover, evidence suggests that the NSIM performs better than the state-of-the-art imputation techniques when considering datasets either with a high prevalence of MVs in a large number of features or with a significant proportion of MVs, while delivering competitive performance in the remaining cases. The proposed method, positions NNs, traditionally considered to be highly sensitive to MVs, among the restricted group of learning algorithms that are capable of handling MVs directly, widening their scope of application. Additionally, the NSIM is prepared to deal with faulty sensors, increasing the attractiveness of this architecture.

- **GPU Computational Framework**
  - An open-source **GPU Machine Learning Library (GPUMLib)** that aims at providing the building blocks for the development of high-performance ML software. GPUMLib contributes for improving and widening the base of GPU ML source code that is available for the scientific community and thus reduce the time and effort devoted to the development of innovative ML applications.
  - A **GPU parallel implementation of the Back-Propagation (BP) and MBP algorithms**, which reduces considerably the long training times of these types of NNs.
  - A **GPU parallel implementation of the NSIM**, which reduces greatly the time spent in the learning phase, making the NSIM an excellent choice for dealing with the Missing Values Problem (MVP).
  - An **Autonomous Training System (ATS)** that tries to mimic our heuristics for model selection. The resulting system, built on top of the BP and MBP GPU parallel implementations, actively searches for better model solutions, by gradually adjusting the topology of the NNs. In addition, it is capable of finding high-quality solutions without human intervention, privileging topologies that are adequate for the specific problems.
  - A total of four different **GPU parallel implementations of the NMF algorithm**, featuring both the multiplicative and the additive update rules and using either the Euclidean distance or the Kullback-Leibler divergence metrics. The performance results of the GPU implementations excel by far those of the Central Processing Unit (CPU), yielding extremely high speedups.
  - A **GPU parallel implementation of the RBMs and DBNs**, which accelerates significantly the (time consuming and computationally expensive) training process of these network architectures The RBM implementation incorporates a proposed adaptive step size procedure for tuning the learning parameters.
- **Practical Application Perspective**
  - A new **learning framework (IHC-SVM) for the protein membership prediction**. This is a particularly relevant real-world problem, because proteins play a prominent role in understanding many biological systems and the fast-growing databases in this area demand new scalable approaches. The resulting two-step system uses the IHC for selecting a reduced subset of the

original data, which is subsequently used to build an SVM model. Given the appropriate memory settings, the proposed approach is able to improve the accuracy performance over the baseline SVM model.
- A new approach for the **prediction of bankruptcy of French companies** (healthy and distressed). This is an actual and pertinent real-world problem, because in recent years, due to the financial crisis, the rate of insolvency has been globally aggravated. The resulting **NSIM-based** systems yielded improved performance over previous approaches, which relied on preprocessing techniques.
- A new model for the **detection of VAs**, in which the GPU parallel implementations were crucial. This is a particularly important real-world problem, because the prevalence of VAs may result in cardiac arrest problems and ultimately lead to sudden death.
- A **hybrid face recognition approach** that combines the NMF-based methods with supervised learning algorithms. The NMF-based methods are used to extract a set of parts-based characteristics, thereby reducing the dimensionality of the data while preserving the information of the most relevant image features. Subsequently, a supervised method, such as the MBP or the SVM is used to build a classifier. The proposed approach is tested on the Yale and AT&T (ORL) facial images databases, demonstrating its potential and usefulness, as well as evidencing robustness to different lighting conditions.
- An **extensive study** for analyzing the factors that affect the quality of **DBNs**, which was made possible thanks to the algorithms' GPU parallel implementations. The study involved training hundreds of DBNs with different configurations on two distinct handwritten character recognition databases (MNIST and HHreco) and contributes for a better understanding of this deep learning system.

1.3 Machine Learning Preliminaries

Learning in the context of ML corresponds to the task of adjusting the parameters, θ, of an adaptive model, using the information contained in a so-called training dataset. Typically, the goal of such models consists of extracting useful information directly from the data or predicting some concept of interest. Depending on the learning approach, ML algorithms can be classified into three different paradigms (supervised, unsupervised and reinforcement learning) [18], as depicted in Figure 1.3. However, the work presented here does not cover the reinforcement learning paradigm. Instead, it is primarily focused on supervised and unsupervised learning, which are traditionally considered to be the two fundamental types of tasks in the ML area [41]. Nevertheless, we also present a semi-supervised learning algorithm. Semi-supervised algorithms offer an in-between approach to unsupervised and supervised algorithms. Essentially, in addition to the unlabeled input data, the algorithm also receives some supervision knowledge, which may

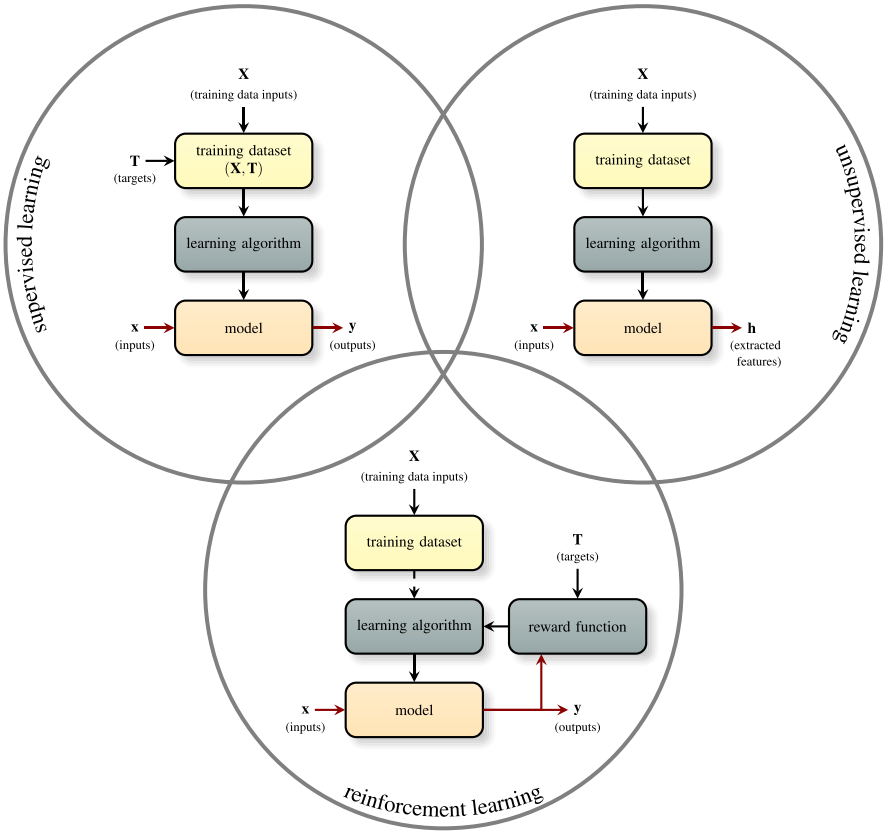


Fig. 1.3 Machine Learning paradigms

include a subset of the targets or some constraint mechanism that guides the learning process [41].

In this book framework, we shall assume that the training dataset is comprised by a set of N samples (instances). Each sample is composed by an input vector, $\mathbf{x} = [x_1, x_2, \ldots, x_D]$, containing the values of the D features that are considered to be relevant for the specific problem being tackled and in the case of the supervised learning paradigm by the corresponding targets (desired values), $\mathbf{t}$. Additionally, we shall assume that all the features are represented by real numbers, i.e. $\mathbf{x} \in \mathbb{R}^D$. Moreover, we are predominantly interested in classification problems in which the model aims to distinguish between the objects of C different classes, based on its inputs. Hence, unless explicitly specified otherwise, we shall consider that $\mathbf{t} = [t_1, t_2, \ldots, t_C]$ where $t_i \in \{0, 1\}$.

Accordingly, the goal of supervised learning algorithms consists of creating a dependency model that associates a specific output vector, $\mathbf{y} \in \mathbb{R}^C$, to each input vector, $\mathbf{x} \in \mathbb{R}^D$. Typically, algorithms relying on the Empirical

Risk Minimization (ERM) principle, e.g. BP, adjust the model parameters, such that the resulting mapping function, $f : \mathbb{R}^D \longrightarrow \mathbb{R}^C$, fits the training data. On the other hand, the Structural Risk Minimization (SRM), e.g. SVMs, attempts to find the models with low Vapnik-Chervonenkis (VC) dimension [169]. This is a core concept, which relates to the interplay between how complex the model is and the capacity of generalization it can achieve. Either way, the objective consists of exploiting the observed data to build models that can make predictions about the output values of unseen input vectors [18].

Let us assume that the training dataset input vectors, $\{\mathbf{x_1}, \mathbf{x_2}, \ldots, \mathbf{x_N}\}$, form an input matrix, $\mathbf{X} \in \mathbb{R}^{N \times D}$, where each row contains an input vector $\mathbf{x_i} \in \mathbb{R}^D$ and similarly, the target vectors, $\{\mathbf{t_1}, \mathbf{t_2}, \ldots, \mathbf{t_N}\}$, form a target matrix, $\mathbf{T} \in \mathbb{R}^{N \times C}$, where each row contains a target vector $\mathbf{t_i} \in \mathbb{R}^C$. Solutions of learning problems by ERM need to be consistent, so that they may be predictive. They also need to be well-posed in the sense of being stable, so that they might be used robustly. Within the empirical risk algorithms we minimize an error function $E(\mathbf{Y}, \mathbf{T}, \theta)$ that measures the discrepancy between the actual model outputs, $\mathbf{Y}$, and the targets, $\mathbf{T}$, so that the model fits the training data. As before, we assume that the model output vectors, $\{\mathbf{y_1}, \mathbf{y_2}, \ldots, \mathbf{y_N}\}$ form an output matrix, $\mathbf{Y} \in \mathbb{R}^{N \times C}$, such that each row contains an output vector $\mathbf{y_i} \in \mathbb{R}^C$. Note that when referring to a generic output vector, we use $\mathbf{y} = [y_1, y_2, \ldots, y_C] \in \mathbb{R}^C$. Although the targets, t_i, are binary $\{0, 1\}$, the actual model outputs, y_i, are usual in the real domain $\mathbb{R}$. Notwithstanding, their values lie in the interval $[0, 1]$, such that (for some algorithms) they can be viewed as a probability

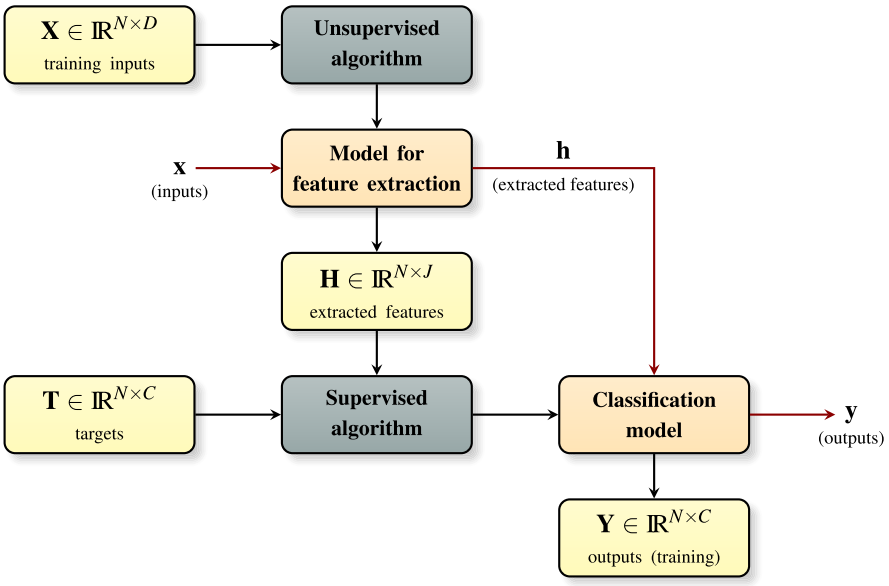


Fig. 1.4 Combining supervised and unsupervised models

(e.g. in a neural network model this value resorts to the odds that the sample belongs to class i).

In the case of unsupervised learning, typically the goal of the algorithms consists of producing a set of J informative features, $\mathbf{h} = [h_1, h_2, \ldots, h_J] \in \mathbb{R}^J$, for each input vector, $\mathbf{x} \in \mathbb{R}^D$. By analogy, the extracted features' vectors, $\{\mathbf{h_1}, \mathbf{h_2}, \ldots, \mathbf{h_N}\}$, form a feature matrix, $\mathbf{H} \in \mathbb{R}^{N \times J}$, where each row contains a feature vector $\mathbf{h_i} \in \mathbb{R}^J$. Eventually, the extracted features can compose a basis for creating better supervised models. This process is illustrated in Figure 1.4.

1.4 Conclusion

In this chapter we intrinsically give the motivation for the development of adaptive many core-machines able to extract knowledge domain in Big Data. The large amount of data is generated from huge multiple sources and real-time data streams in real applications, for which current Machine Learning methods and tools are unable to cope with. Therefore, the need to extend their applicability to such deluge of information by making use of software research platforms with easy accessibility. The chapter intents to give an overview of research topics that will be approached through the book from multiple points of view: theory, development, and application. Regarding the latter issue, we have described in a practical perspective methods and tools using GPU platforms that are able in part to respond to such challenges. Moreover, we cover the preliminaries that are used across the book for a clear understanding of the methods and approaches carried out in both theoretical and experimental parts.

Chapter 2
GPU Machine Learning Library (GPUMLib)

Abstract. The previous chapter accentuated the need for the understanding of large, complex, and distributed data sets generated from digital sources coming from sensors or other physical instruments as well as simulations, crowd sourcing, social networks or other internet transactions. The focus was on the difficulties posed to ML algorithms to extract knowledge with prohibitive computational requirements. In this chapter we introduce the GPU, which represents a novel and compelling solution for this problem, due to its inherent high-parallelism. Seldom ML algorithms have been implemented on the GPU and most are not openly shared. To mitigate this problem, this Chapter describes a new open-source library (GPUMLib), that aims to provide the building blocks for the development of efficient GPU ML software. In the first part of the chapter we cast arguments for the need of an open-source GPU ML library. Next, it presents an overview of the open-source and proprietary ML algorithms implemented on the GPU, prior to the development of GPUMLib. In addition we focus on the evolution of the GPU from a fixed-function device, designed to accelerate specific tasks, into a general-purpose computing device. The last part of the chapter details the CUDA programming model and architecture, which was used to develop GPUMLib. Finally, the general GPUMLib architecture is described.

2.1 Introduction

The rate at which new information is produced has been and continues to grow with an unprecedented magnitude. New devices and sensors allow humans and machines to readily gather, store and share vast amounts of information worldwide. Projects such as the Australian Square Kilometre Array of radio telescopes, the CERN's Large Hadron Collider and astronomy's Pan-STARRS array of celestial telescopes can generate several petabytes of data per day on their own [85]. However, availability does not necessarily imply usefulness and humans facing the innumerable requests, imposed by modern life, need help to cope and take advantage of the high-volume of data generated and accumulated by our society [129].

N. Lopes and B. Ribeiro, *Machine Learning for Adaptive Many-Core Machines – A Practical Approach,* Studies in Big Data 7,
DOI: 10.1007/978-3-319-06938-8_2,

Usually obtaining the information represents only a fraction of the time and effort needed to analyze it [85]. This brings the need for intelligent systems that can extract relevant and useful information from today's large repositories of data, and subsequently the issues posed by more challenging and demanding ML algorithms, often computationally expensive [139].

Although at present there are plentiful excellent toolkits which provide support for developing ML software in several environments (e.g. Python, R, Lua, Matlab) [104], these fail to meet the expectations in terms of computational performance, when dealing with many of today's real-world problems. Typically, ML algorithms are computationally expensive and their complexity is often directly related with the amount of data being processed. Rationally, as the volume of data increases, the trend is to have more challenging and computationally demanding problems that can become intractable for traditional CPU architectures. Therefore, the pressure to shift development toward parallel architectures with high-throughput has been accentuated. In this context, the GPU represents a compelling solution to address the increasing needs of computational performance, in particular in the ML field [129].

Over the last decade the performance and capabilities of the GPUs have been significantly augmented and today's GPUs, included in mainstream computing systems, are powerful, highly parallel and programmable devices that can be used for general-purpose computing applications [171]. Since GPUs are designed for high-performance rendering where repeated operations are common, they are much more effective in utilizing parallelism and pipelining than CPUs [97]. Hence, they can provide remarkable performance gains for computationally-intensive applications involving data-parallelizable tasks.

Current GPUs offer an unprecedented peak performance that is over one order of magnitude larger than those of modern CPUs and this gap is likely to increase in the future. This aspect is depicted in Figure 2.1, updated from Owens et al. [170], which shows that the GPU peak performance is growing at a much faster pace than the corresponding CPU performance. Typically, the GPU performance is doubled every 12 months while the CPU performance doubles every 18 months [262].

It is not uncommon for GPU implementations to achieve significant time reductions, as compared with CPU counterparts (e.g. weeks of processing on the CPU may be transformed into hours on the GPU [123]). Such characteristics trigger the interest of the scientific community who successfully mapped a broad range of computationally demanding problems to the GPU [171]. As a result, the GPU represents a credible alternative to traditional microprocessors in the high-performance computer systems of the future [171].

To successfully take advantage of the GPU, applications and algorithms should present a high-degree of parallelism, large computational requirements and favor data throughput in detriment of the latency of individual operations [171]. Since most ML algorithms and techniques fall under these guidelines, GPUs represent a hardware framework that provides the means for the realization of high-performance implementations of ML algorithms. Hence, they are an attractive alternative to the use of dedicated hardware, such as Field-Programmable Gate Arrays (FPGAs). In

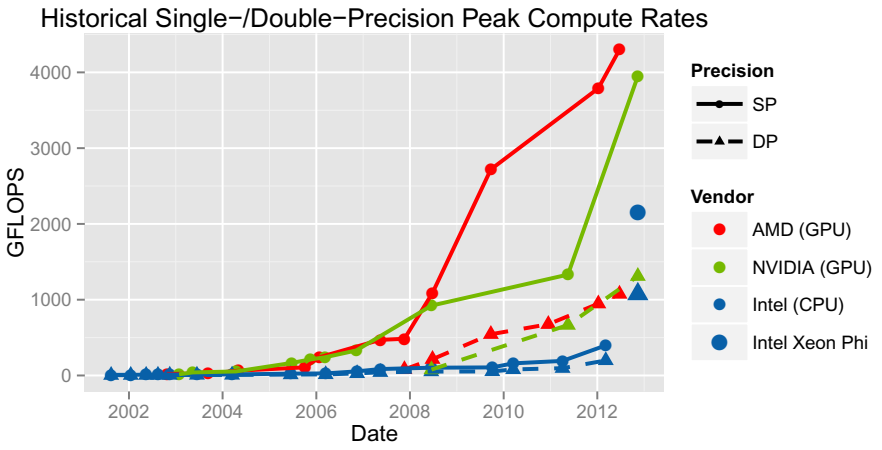


Fig. 2.1 Disparity between the CPU and the GPU peak floating point performance, over the years, in billions (10^9) of floating-point operations per second (GFLOPS)[1]

our view, the GPU represents the most compelling option, concerning these two types of accelerators, since dedicated hardware usually fails to meet expectations, as it is typically expensive, unreliable, poorly documented, with reduced flexibility, and obsolete within a few years [217, 25]. Although FPGAs are highly customizable hardware devices, they are much harder to program. Typically, adapting and changing algorithms requires hardware modifications, while the same process can be accomplished on the GPU simply by rewriting and recompiling the code [43]. Moreover, although FPGAs can potentially yield the best performance results [43], recently several studies have concluded that GPUs are not only easier to program, but they also tend to outperform FPGAs in scientific computation tasks [259]. In addition, the flexibility of the GPU allows software to run on a wide range of devices without any changes, while the software developed for FPGAs is highly dependent on the specific type of chip for which it was conceived and therefore has a very limited portability [1]. Furthermore, the resulting implementations cannot be shared and validated by others, who probably do not have access to the hardware. GPUs on the other hand are used in the ubiquitous gaming industry, and thus mass produced and regularly replaced by a new generation with increasing computational power and additional levels of programmability. Consequently, unlike many of the earlier throughput-oriented architectures, they are widely available and relatively inexpensive [71, 217, 35].

Naturally, the programming model used to develop applications for the GPU plays a fundamental role in its success as a general-purpose computing device. In this context, the Compute Unified Device Architecture (CUDA) represented a major step toward the simplification of the GPU programming model by providing support

[1] Figure 2.1 is a courtesy of Professor John Owens, from the University of California, Davis, USA.

for accessible programming interfaces and industry-standard languages, such as C and C++. CUDA was released by NVIDIA in the end of 2006 and since then numerous GPU implementations, spanning a wide range of applications, have been developed using this technology. While there are alternative options, such as the Open Computing Language (OpenCL), the Microsoft Directcompute or the AMD Stream, so far CUDA is the only technology that has achieved wide adoption and usage [216].

Using GPUs for general-purpose scientific computing allowed a wide range of challenging problems to be solved more rapidly, providing the mechanisms to study larger datasets [197]. GPUs are responsible for impressive speedups for many problems in a wide range of areas. Thus it is not surprising that they have become the platform of choice in the scientific computing community [197].

The scientific breakthroughs of the future will undoubtedly be powered by advanced computing capabilities that will allow to manipulate and explore massive datasets [85]. However, cooperation among researchers also plays a fundamental role and the speed at which a given scientific field advances will depend on how well they collaborate with one another [85].

Overtime, a large body of powerful algorithms, suitable for a wide range of applications, has been developed in the field of ML. Unfortunately, the true potential of these methods has not been fully capitalized on, since existing implementations are not openly shared, resulting in software with low usability and weak interoperability [215].

Moreover, the lack of openly available implementations is a serious obstacle to algorithm replication and application to new tasks and therefore poses a barrier to the progress of the ML field. Sonnenburg et al. argue that these problems could be significantly amended by giving incentives to the publication of software under an open source model [215]. This model presents many advantages that ultimately lead to: better reproducibility of experimental results and fair comparison of algorithms; quicker detection of errors; faster adoption of algorithms; innovative applications and easier combination of advances, by fomenting cooperation: it is possible to build on top of existing resources (rather than re-implementing them); faster adoption of ML methods in other disciplines and in industry [215].

Recognizing the importance of publishing ML software under the open source model, Sonnenburg et al. even propose a method for formal publication of ML software, similar to those that the ACM Transactions on Mathematical Software provide for Numerical Analysis. They also argue that supporting software and data should be distributed under a suitable open source license along with scientific papers, pointing out that this is a common practice in some bio-medical research, where protocols and biological samples are frequently made publicly available [215].

2.2 A Review of GPU Parallel Implementations of ML Algorithms

We conducted an in-depth analysis of several papers dealing with GPU ML implementations. To illustrate the overwhelming throughput of current research, we represent in Figure 2.2 the chronology of ML software GPU implementations, until late 2010, based on the data scrutiny from several papers [20, 166, 30, 143, 217, 244, 252, 262, 16, 27, 44, 250, 81, 150, 25, 35, 55, 67, 77, 97, 107, 109, 204, 205, 226, 232, 78, 124, 159, 180, 191, 248, 128].

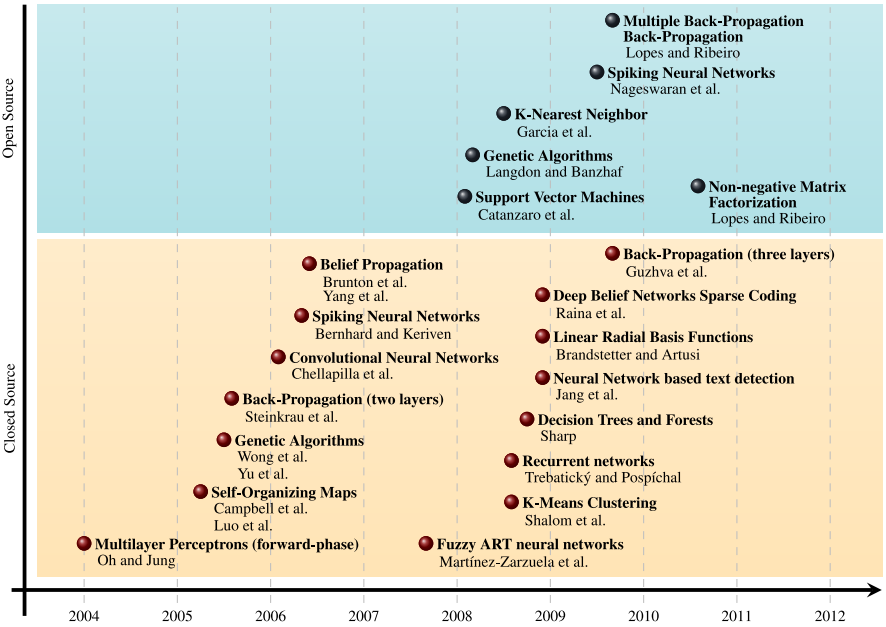


Fig. 2.2 Chronology of ML software GPU implementations

The number of GPU implementations of ML algorithms has increased substantially over the last few years. However, within the period analyzed, only a few of those were released under open source. Aside from our own implementations, we were able to find only four more open source GPU implementations of ML algorithms. This is an obstacle to the progress of the ML field, as it may force those facing problems where the computational requirements are prohibitive, to build from scratch GPU ML algorithms that were not yet released under open source. Moreover, being an excellent ML researcher does not necessary imply being an excellent programmer [215]. Additionally, the GPU programming model is significantly different from the traditional models [71, 171] and to fully take advantage of this architecture one must first become versed on the specificities of

this new programming paradigm. Thus, many researchers may not have the skills or the time required to implement algorithms from scratch. To alleviate this problem and promote cooperation, we have developed a new GPU ML library, designated GPUMLib, as part of this Thesis framework. GPUMLib aims at reducing the effort of implementing new ML algorithms for the GPU and contribute to the development of innovative applications in the area. The library, described in more detail in Section 2.5, is developed mainly in C++, using the CUDA architecture.

Recently, other GPU implementations have been released for SVMs [114, 72, 108, 83], genetic algorithms [36, 37, 31, 48], belief propagation [246], *k*-means clustering and *k*-nearest neighbor (*k*-nn) [99], particle swarm optimization [95], ant colony optimization [38], random forest classifiers [59] and sparse Principal Component Analysis (PCA) [190]. However, only a few have their source code publicly available.

2.3 GPU Computing

All of today's commodity GPUs structure their computation in a graphics pipeline, designed to maintain high computation rates through parallel execution [170]. The graphics pipeline typically receives as input a representation of a three-dimensional (3D) scene and produces a two-dimensional (2D) raster image as output. The pipeline is divided into several stages, as illustrated in Figure 2.3 [62]. Originally, it was simply a fixed-function pipeline, with a limited number of predefined operations (in each stage) hard-wired for specific tasks. Even though these hard-wired graphics algorithms could be configured in a variety of ways, applications could not reprogram the hardware to do tasks unanticipated by its designers [170]. Fortunately, this situation has changed over the last decade. The fixed-function pipeline has gradually been transformed into a more flexible and increasingly programmable one. The vital step for enabling General-Purpose computing on Graphics Processing Units (GPGPU) was given with the introduction of fully programmable hardware and an assembly language for specifying programs to run on each vertex or fragment [170].

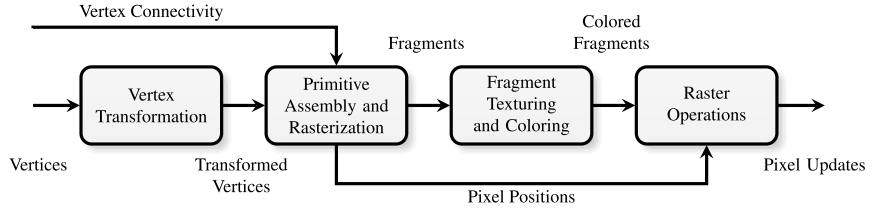


Fig. 2.3 Graphics hardware pipeline

Recognizing the potential of GPUs for general-purpose computing, vendors added driver and hardware support to use the highly parallel hardware of the GPU without the need for computation to proceed through the entire graphics pipeline and without the need to use 3D Application Programming Interfaces (APIs) at all [42]. NVIDIA CUDA general-purpose parallel computing architecture is an example of the efforts made in order to embrace the promising new market of GPGPU. Instead of using graphics APIs, we can use the industry-standard C and C++ languages together with CUDA extensions to target a general-purpose, massively parallel processor (GPU). To differentiate this new model of programming for the GPU, and clearly separate it from traditional GPGPU, the term GPU Computing was coined [162]. Another example of commitment of the hardware industry consists of the emergence of GPUs, such as the Tesla, whose sole purpose is to allow high-performance general-purpose computing. This boosted the deployment of economic personal desktop supercomputers, which can achieve a performance far superior to standard personal computers.

Owens et al. provided a very exhaustive survey on GPGPU, identifying many of the algorithms, techniques and applications implemented on graphics hardware [170].

2.4 Compute Unified Device Architecture (CUDA)

The CUDA architecture exposes the GPU as a massive-parallel device that operates as a co-processor to the host (CPU). The GPU can significantly reduce the computation time for data parallel workloads, where analogous operations are executed in large quantities of data. Once data parallel workloads are identified, portions of the application can be retargeted to take advantage of the GPU parallel characteristics. To this end, programs must be able to break down the original workload tasks into independent processing blocks [133].

2.4.1 CUDA Programming Model

The CUDA programming model extends the C and C++ languages, allowing us to explicitly denote data parallel computations by defining special functions, designated by kernels. Kernel functions are executed in parallel by different threads, on a physically separate device (GPU) that operates as a co-processor to the host (CPU) running the program. These functions define the sequence of work to be carried out individually by each thread mapped over a domain (the set of threads to be invoked) [42]. Threads must be organized/grouped into blocks, which in turn form a grid. In recent GPUs, grids may have up to three dimensions, while on older devices the limit is two dimensions. This information is contained in Table 2.1 which presents the main technical specifications according to the CUDA device compute

capability. A complete list of the specifications can be found in the NVIDA CUDA C programming guide [164]. Moreover, a list of the devices supporting each compute capability can be found at `http://developer.nvidia.com/cuda-gpus`.

Table 2.1 Principal technical specifications according to the CUDA device compute capability

	Compute Capability					
Technical Specifications	**1.0**	**1.1**	**1.2**	**1.3**	**2.x**	**3.0**
Maximum grid dimensionality	2 (x, y)				3 (x, y, z)	
Maximum x-dimension of a grid	65535					$2^{31}-1$
Maximum y or z-dimension of a grid	65535					
Maximum block dimensionality	3 (x, y, z)					
Maximum x or y-dimension of a block	512				1024	
Maximum z-dimension of a block	64					
Maximum number of threads per block	512				1024	
Warp size (see Section 2.4.2, page 26)	32					
Maximum resident blocks per multiprocessor	24		32		48	64
Maximum resident threads per multiprocessor	768		1024		1536	2048
Number of 32-bit registers per multiprocessor	8 K		16 K		32 K	64 K
Maximum shared memory per multiprocessor	16 KB				48 KB	
Local memory per thread	16 KB				512 KB	
Maximum number of instructions per kernel	2 million				512 million	

For convenience, blocks can organize threads in up to three dimensions. Figure 2.4 presents an example of a two-dimensional grid containing two-dimensional thread blocks. The actual structure of the blocks and the grid depends on the problem being tackled and in most cases is directly related to the structure of the data being processed. For example, if the data is contained in a single array, then it makes sense to use a one-dimensional grid with single dimensional blocks, each processing a specific region of the array. On the other hand, if the data is contained in a matrix then it could make more sense to use a bi-dimensional grid in which one dimension is used for the column and another one for the row. In this specific scenario the blocks could also be organized using two dimensions, such that each block would process a distinct rectangular area of the matrix.

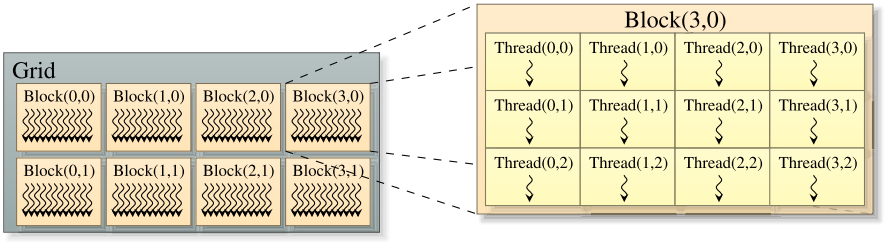


Fig. 2.4 Example of a kernel grid

Choosing the adequate block size and structure is fundamental to maximize the kernels' performance. Unfortunately, it is not always possible to anticipate which block structure is the best and changing it may require rewriting kernels from scratch. Threads within a block can cooperate among themselves by sharing data and synchronizing their execution to coordinate memory accesses. However, the number of threads comprising a block can not exceed 512 or 1024 depending on the GPU compute capability (see Table 2.1). This limits the scope of synchronization and communication within the computations defined in the kernel. Nevertheless, this limit is necessary in order to leverage the GPU high-core count by allowing threads to be distributed across all the available cores.

Blocks are required to execute independently: it must be possible to execute them in any arbitrary order, either in parallel or in series. This requirement allows the set of thread blocks which compose the grid to be scheduled in any order across any number of cores, enabling applications that scale well with the number of cores present on the device.

Scalability is a fundamental issue, since the key to performance in this platform relies on using massive multi-threading to exploit the large number of device cores and hide global memory latency. To achieve this, we face the challenge of finding the adequate trade-off between the resources used by each thread and the number of simultaneously active threads. The resources to manage include the number of registers, the amount of shared (on-chip) memory used per thread, the number of threads per multiprocessor and the global memory bandwidth [194].

CUDA provides a set of intrinsic variables that kernels can use to identify the actual thread location in the domain, allowing each thread to work on separate parts of a dataset [42]. Table 2.2 identifies those built-in variables [164].

Table 2.2 Built-in CUDA kernel variables

Variable	Description
gridDim	Dimensions of the kernel grid.
blockDim	Dimensions of the block.
blockIdx	Index of the block, being processed, within the grid.
threadIdx	Thread index within the block.
warpSize	Warp size in threads (see Section 2.4.2, page 26).

Listing 2.1 presents a simple kernel that computes the square of each element of vector **x**, placing the result in vector **y**. Kernel functions are declared by using the qualifier `__global__` and can not return any value (i.e. its return type must be `void`).

The actual number of threads is only defined when the kernel function is called. To this end, we must specify both the grid and the block size by using the new CUDA execution configuration syntax ($<<<\cdots>>>$). Listing 2.2 demonstrates the steps necessary to call the square kernel previously defined in Listing 2.1. These usually involve allocating memory on the device, transfer the input data from the host to the device, define the number of blocks and the number of threads per block

Listing 2.1 Example of a CUDA kernel function. CUDA specific keywords appear in blue.

```
__global__ void square(float * x, float * y, int size) {
    int idx = blockIdx.x * blockDim.x + threadIdx.x;
    if (idx < size) y[idx] = x[idx] * x[idx];
}
```

Listing 2.2 Example for calling a CUDA kernel function.

```
//...

float x[SIZE];
float y[SIZE];

int memsize= SIZE * sizeof(float);

// Fill vector x
// ...

// Allocate memory on the device for the vectors x and y
float * d_x;
float * d_y;
cudaMalloc((void**) &d_x, memsize);
cudaMalloc((void**) &d_y, memsize);

// Transfer the array x to the device
cudaMemcpy(d_x, x, memsize, cudaMemcpyHostToDevice);

// Call the square kernel function using blocks of 256
    threads
const int blockSize = 256;
int nBlocks = SIZE / blockSize;
if (SIZE % blockSize > 0) nBlocks++;
square<<<nBlocks, blockSize>>>(d_x, d_y, SIZE);

// Transfer the result vector y to the host
cudaMemcpy(y, d_y, memsize, cudaMemcpyDeviceToHost);

//release device memory
cudaFree(d_x);
cudaFree(d_y);

//...
```

for each kernel, call the appropriate kernel functions, copy the results back to the host and finally release the device memory.

In the code presented (see Listings 2.1 and 2.2), each thread will process a single element of the array. Since the block size exerts a profound impact on the kernel performance, usually this is one of the most important aspects taken

into consideration when choosing the block size and consequently the number of blocks. Hence, the actual number of threads (number of blocks $\times$ block size) will most likely be greater than the size of the array being processed. As a result, it is frequent to have some idle threads, within the grid blocks, as depicted in Figure 2.5.

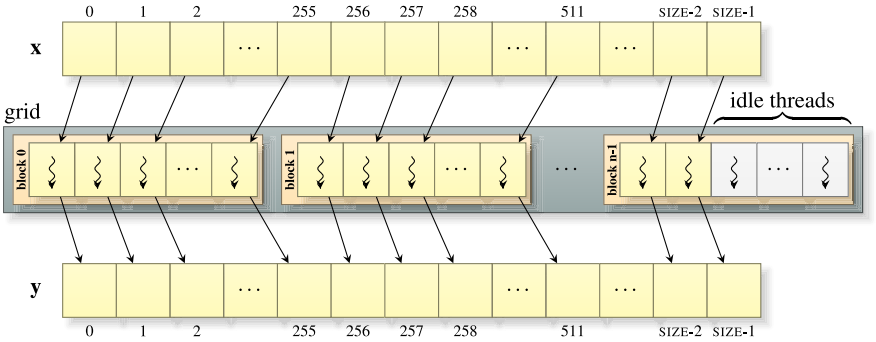


Fig. 2.5 Execution of the `square` kernel grid blocks (see Listings 2.1 and 2.2)

2.4.2 CUDA Architecture

The CUDA programming model is supported by an architecture built around a scalable array of multi-threaded Streaming Multiprocessors (SMs), as depicted in Figure 2.6. Each SM contains several Scalar Processor (SP) cores (also referred to as CUDA cores), according to the compute capability of the devices as shown in Table 2.3.

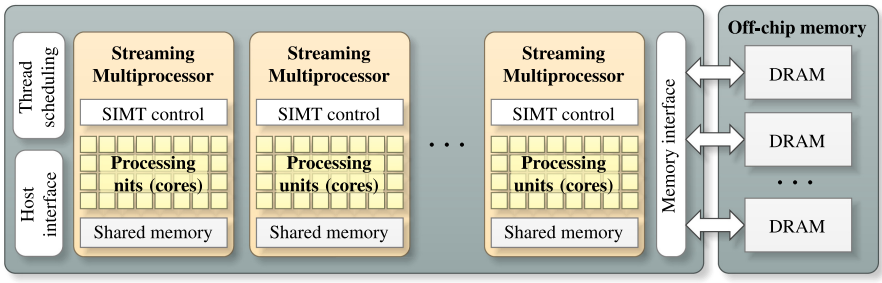


Fig. 2.6 NVIDIA (GPU) device architecture

Table 2.3 Number of Scalar Processor (SP) cores per Streaming Multiprocessor (SM), according to the compute capability of the device (GPU)

Compute Capability	**1.x**	**2.0**	**2.1**	**3.0**
Number of cores per multiprocessor	8	32	48	192

Figure 2.7 shows a detailed diagram of an SM [79]. Although an SP core resembles a general-purpose processor, similar to those found in a x86 core, it is in fact much simpler, reflecting its heritage as a pixel shader processor. Each core contains a pipelined Floating-Point Unit (FPU), a pipelined integer unit, some logic for dispatching instructions and operands to these units, and a queue for holding the results [79]. The cores lack their own general-purpose register files, L1 caches, multiple function units for each data type and load/store units for retrieving and storing data. Instead those resources are shared between all the cores of the SM. The latter also contains Special Function Units (SFUs) to handle complex math operations (e.g. square roots, reciprocals, Sines and Cosines) [79].

The SMs are optimized for single-precision floating point, since this is enough for traditional 3D graphics applications. In fact, the support for double-precision floating point, on NVIDIA GPUs, was only recently added in order to address the needs of scientific and High-Performance Computing (HPC) applications [162]. Currently double-precision instructions are only supported on devices with compute capability 1.3 or above [164].

Implementing a 64-bit FPU in each core would roughly double the amount of floating-point computational logic and wiring, making GPUs larger, costlier and more power demanding, without bringing any real-benefits for its primary market (consumer 3D graphics) [79]. Hence, there is still a significant disparity between single-precision and double-precision computations. For example, in devices based on the Fermi architecture (released in 2010), single-precision computations are twice as fast as double-precision computations [79].

When a program on the host invokes a kernel grid, its blocks are enumerated and distributed to the SMs with available execution capacity. Each block runs entirely on a single SM and when its execution is complete, new blocks are launched on the vacated SMs (see Figure 2.8). The underlying idea consists of distributing the workload across all the SMs, which depending on the resources (e.g. shared memory, registers) required by each block, may be able to handle several blocks simultaneously. Hence, the number of blocks running simultaneously depends not only on the number of SMs of the device but also on the amount of resources allocated for each block.

While a typical x86 processor has only a few cores, each usually running two threads, a CUDA GPU is able to run thousands of threads, fast-switching among them at every clock cycle [79]. To this end, the SMs employ a new architectural model: Single-Instruction Multiple-Thread (SIMT) (see Figure 2.6). The multiprocessor SIMT unit creates, manages, schedules, and executes groups of 32 parallel threads called warps. Thus it is advantageous to use a block size that is a multiple of 32, since the blocks are divided into warps.

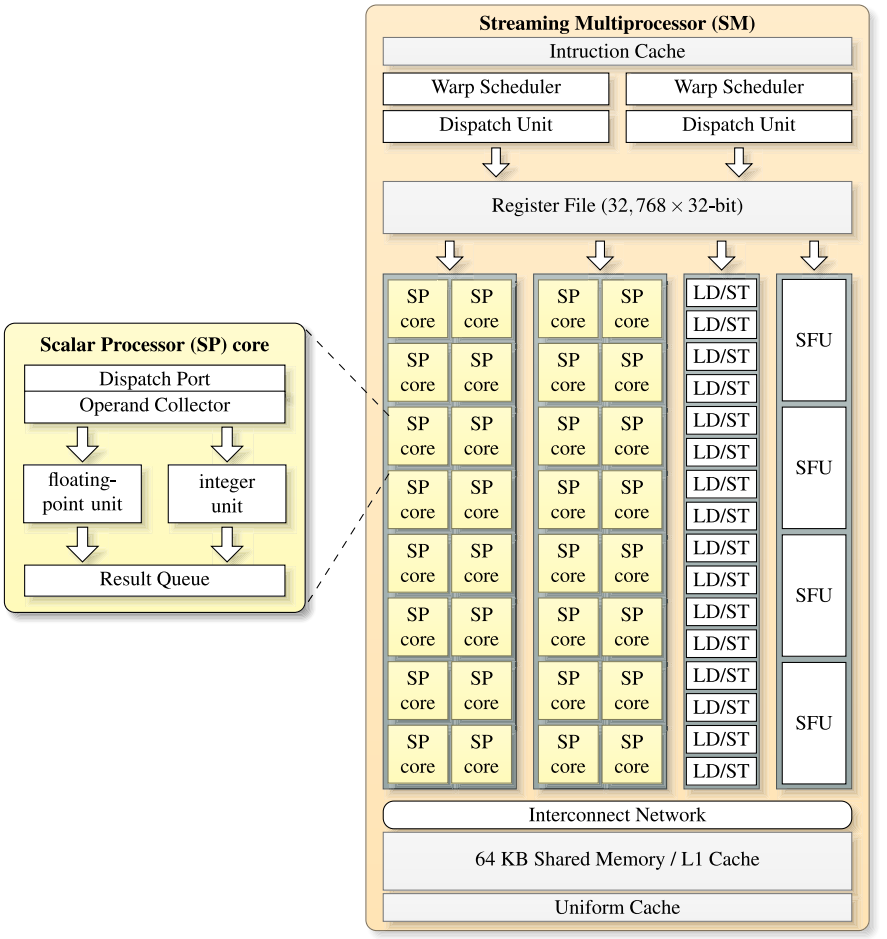


Fig. 2.7 Diagram of a Fermi Streaming Multiprocessor (SM)

Conditional branches cause warps to be serially executed for each path taken, disabling the threads that do not belong to that specific path, as illustrated in Figure 2.9. Thus, full efficiency is obtained when all the warp threads agree on their execution path [164].

Additionally, it is important to arrange the data so that coherence in the memory accesses, carried out by adjacent warp threads, is achieved in order to maximize the kernels' performance. Structuring the data appropriately is fundamental to create global memory access patterns that may allow the hardware to coalesce groups of reads or writes of multiple data items into a single operation [163]. The requirements for obtaining coalesced memory accesses vary according to the device compute capability. Devices with lower computing capabilities impose

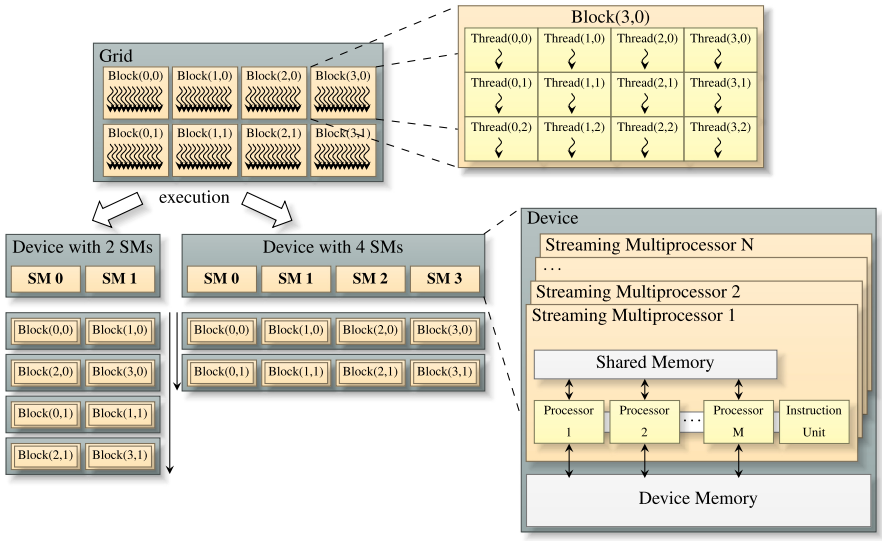


Fig. 2.8 Execution of a kernel grid on different devices (GPUs)

tighter restrictions than more recent ones. Nevertheless, it is usually possible to enforce coalesced memory accesses by guaranteeing that the threads access the memory in a sequential manner. Figure 2.10 illustrates both coalesced and non-coalesced memory access patterns. Non-coalesced patterns require additional memory transactions that ultimately reduce the instruction throughput [164].

In Ryoo et al. [194] the major principles for identifying which data parallel algorithms can benefit from a CUDA implementation are highlighted. Namely, a much larger number of threads (thousands or even millions, depending on the problem) than those required by traditional multi-core systems are needed to hide the global memory latency. Thus, we need to define threads at a finer granularity. Moreover, grouping threads appropriately in order to avoid memory conflicts, non-sequential memory accesses (within warps) and divergent control flow decisions (within warps), may have a significant impact in the performance. Additionally, the adequate use of the shared memory to reduce bandwidth usage and redundant execution is also an important factor.

2.5 GPUMLib Architecture

The GPUMLib framework was developed in the context of this Thesis. Its main components are presented in Figure 2.11. At its core, the library contains a set of CUDA kernels that support the execution of ML algorithms on the GPU. Usually,

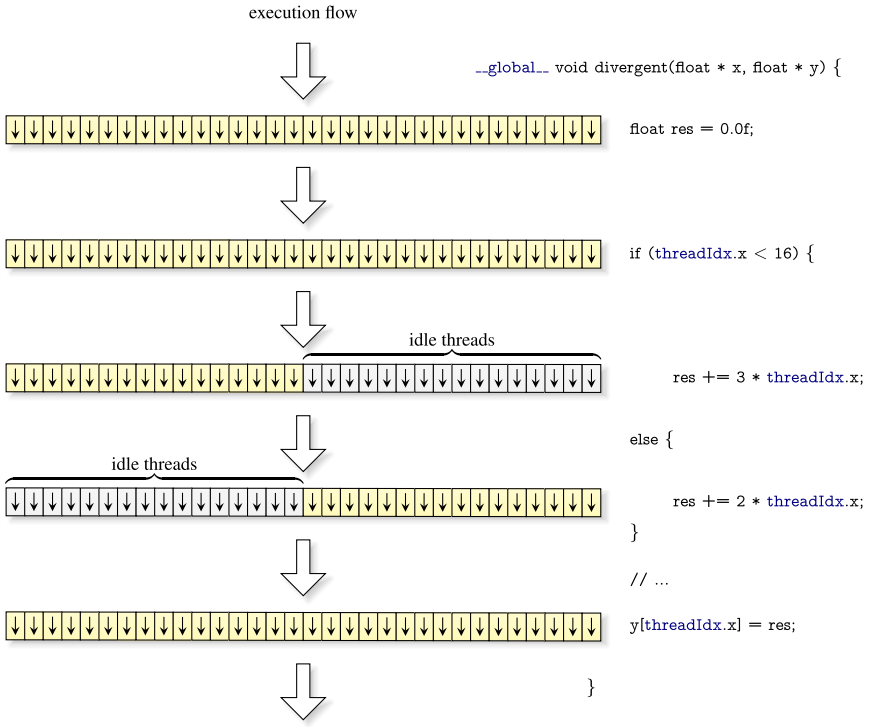


Fig. 2.9 Warp divergence effects. Each rectangle with an arrow represents a warp thread that is either active or idle depending on the execution branch.

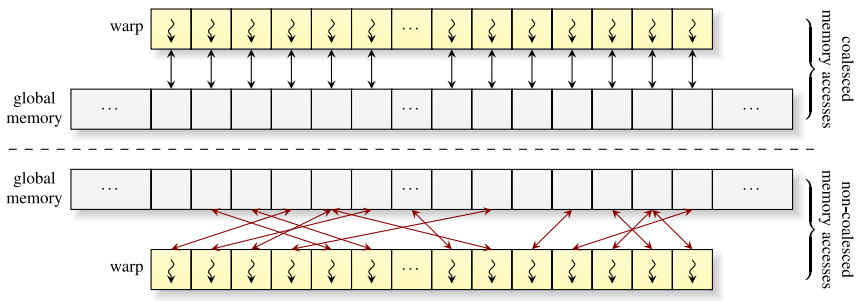


Fig. 2.10 Coalesced versus non-coalesced memory access patterns. It is assumed that the size of each data element does not prevent coalesced memory accesses.

in order to implement an ML algorithm on the GPU, several kernels are required. However, the same kernel might be used to implement different algorithms. For example, the Back-Propagation (BP) and the Multiple Back-Propagation (MBP) algorithms share the same kernels (see Section 3.3).

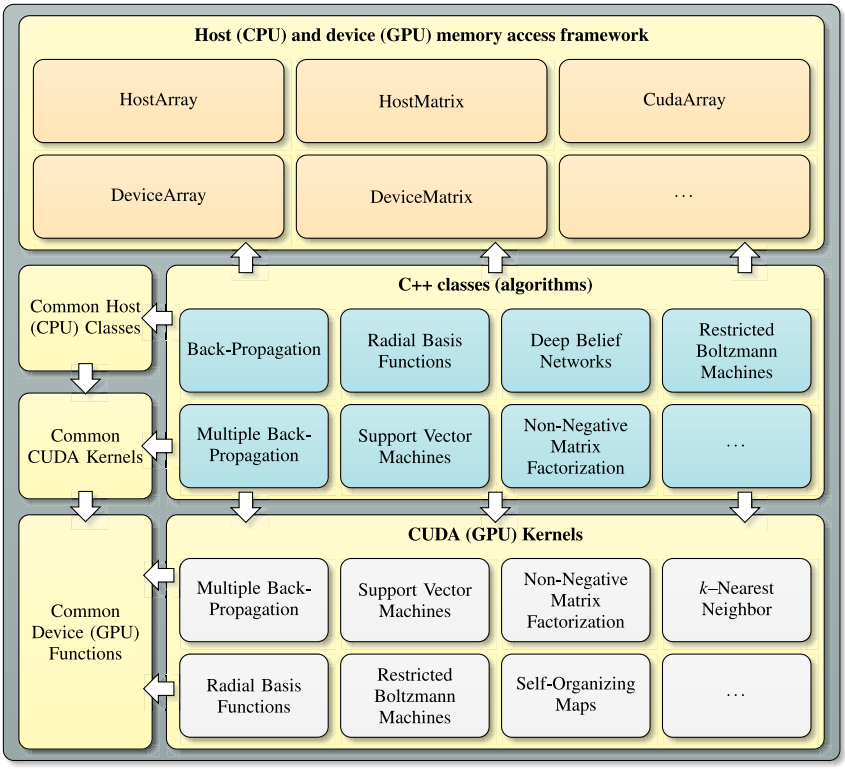


Fig. 2.11 Main components of the GPUMLib

Each ML algorithm has its own C++ class that is responsible for: transferring the information (inputs) needed by the algorithm to the device (GPU); calling the algorithm kernels in the proper order; and transferring the algorithm outputs and intermediate values back to the host. This model allows non-GPU developers to take advantage of GPUMLib, without requiring them to understand the specific details of CUDA programming.

GPUMLib provides a standard memory access framework to support the tasks of memory allocation and data transfer between the host and device (and vice-versa) in an effortless and seemly manner. Table 2.4 describes the classes currently supported by the GPUMLib memory access framework. To illustrate the advantages of using this framework, Listing 2.3 rewrites the code of Listing 2.2 (see page 24), using

Table 2.4 GPUMLib memory access framework classes

Class	Description
HostArray	Array contained in the host memory.
HostMatrix	Matrix contained in the host memory.
DeviceArray	Array contained in the device memory.
DeviceMatrix	Matrix contained in the device memory.
CudaArray	Array contained both in the host and in the device.
CudaMatrix	Matrix contained both in the host and in the device.
DeviceAccessibleVariable	Variable contained in the host memory, which is page-locked and accessible by the device.

Listing 2.3 Example for calling a CUDA kernel function using the GPUMLib memory access framework classes.

```
//...

// No need to explicitly allocate and release memory
CudaArray<float> x(SIZE);
CudaArray<float> y(SIZE);

// Fill vector x in the host as usual
// ...

// Transfer the array x to the device
x.UpdateDevice();

// Call the square kernel function using blocks of 256
    threads
const int blockSize = 256;
int nBlocks = SIZE / blockSize;
if (SIZE % blockSize > 0) nBlocks++;
square<<<nBlocks, blockSize>>>
    (x.DevicePointer(), y.DevicePointer(), SIZE);

// Transfer the result vector y to the host
y.UpdateHost();

//...
```

the CudaArray class. Notice that the new code is much more intuitive and less error-prone than the original one.

Among the classes of the memory access framework, a class is included to represent GPU matrices (DeviceMatrix), which provides a straightforward and efficient way of performing GPU matrix computations (multiplication and transpose). Its implementation takes advantage of the CUBLAS library (CUDA Basic Linear Algebra Subprograms (BLAS)), which is part of CUDA, to perform matrix multiplications, due to its high-performance.

Moreover, since the order in which the elements of the matrix are stored has a considerable effect on the kernels performance, the `DeviceMatrix` class supports both row-major and column-major orders, fitting the needs of the users. In row-major, the elements of each row are placed together in sequential memory locations while the elements of each column are stored in non-sequential addresses. This implies that kernels can access the elements of each row in a coalesced manner, but they will be unable to do the same for the elements in each column. On the other hand, if the matrix is stored in column-major then the elements of each column are stored in sequential memory locations while the elements of each row are stored in non-sequential addresses. In this case the elements of each column can now be accessed in a coalesced manner, but the elements of each row may only be accessed in a non-coalesced manner. Figure 2.12 depicts the organization of elements of a matrix in the memory, according to the selected order.

In terms of common classes, currently GPUMLib implements a class (`Reduction`) as well as a set of GPU kernels for performing common reduction tasks, such as finding the minimum, maximum, sum or average of a set of elements. Reduction is a process in which we gradually perform the intended operation in parallel on two elements at a time, thus effectively reducing the number of elements to be processed to a half in each iteration, until a single element is found. Figure 2.13 illustrates this process for a sum operation in coalesced manner.

Additionally GPUMLib also implements a class (`Random`) for simplifying the task of generating random numbers on the GPU that uses the CURAND library, which is part of the CUDA toolkit.

The latest version of GPUMLib implements the ML algorithms listed in Table 2.5. The core of the library as well as the BP, MBP, NSIM, ATS, NMF, RBMs and DBNs were developed in this Thesis framework. Additionally, as part of this endeavor, the Radial Basis Function (RBF) networks were developed by Quintas [179] and the SVMs were developed by Gonçalves [75] as part of their Master's Thesis which were supervised by this book's authors.

Table 2.5 GPU parallel algorithms implemented in version 0.2.0 of GPUMLib

Algorithm/Architecture
Back-Propagation (BP)
Deep Belief Networks (DBNs)
Multiple Back-Propagation (MBP)
Non-Negative Matrix Factorization (NMF)
Radial Basis Function (RBF) networks
Restricted Boltzmann Machines (RBMs)
Support Vector Machines (SVMs)
Autonomous Training System (ATS)
Neural Selective Input Model (NSIM)

Since good documentation plays a major role in the success of any software library, GPUMLib provides extensive quality documentation and examples to

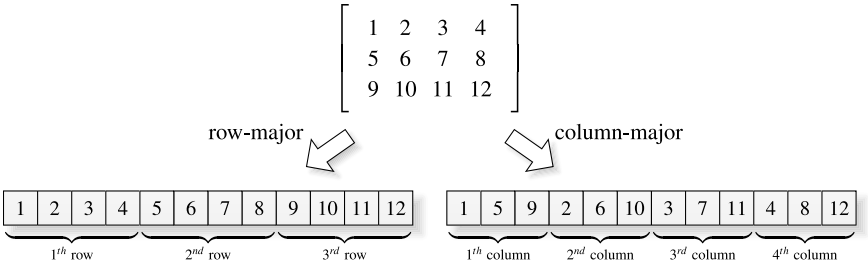


Fig. 2.12 Row-major versus column-major orders

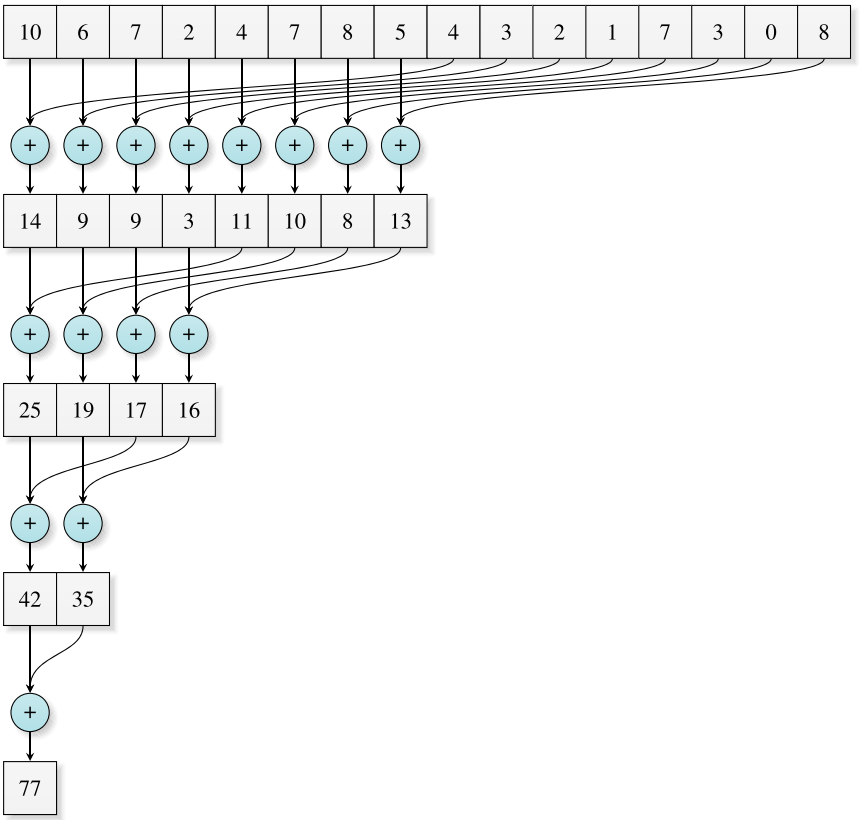


Fig. 2.13 Example of a sum reduction

ease its usage and development. Moreover, the library is practical, easy to use and extend, and does not require full understanding of the details inherent to GPU computing. The library is released under the GNU General Public

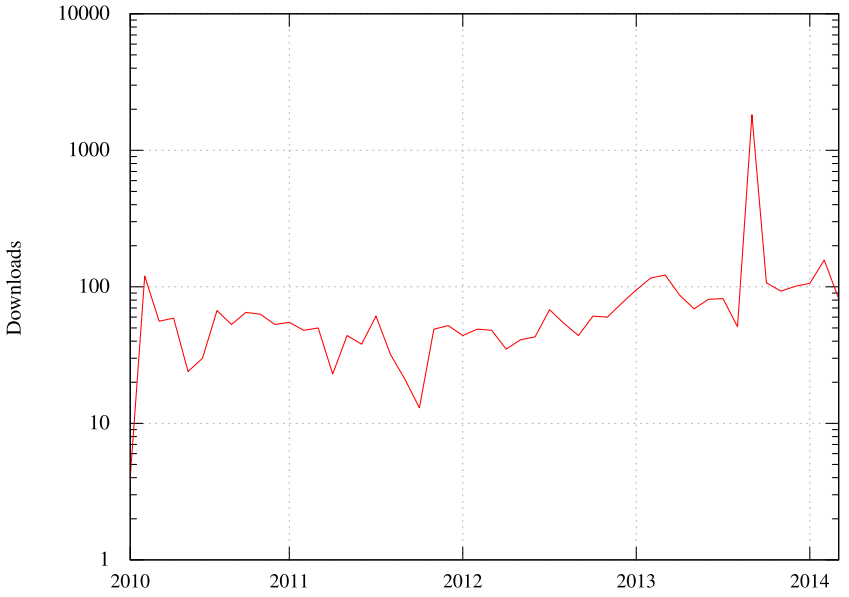


Fig. 2.14 Evolution of the number of downloads of GPUMLib

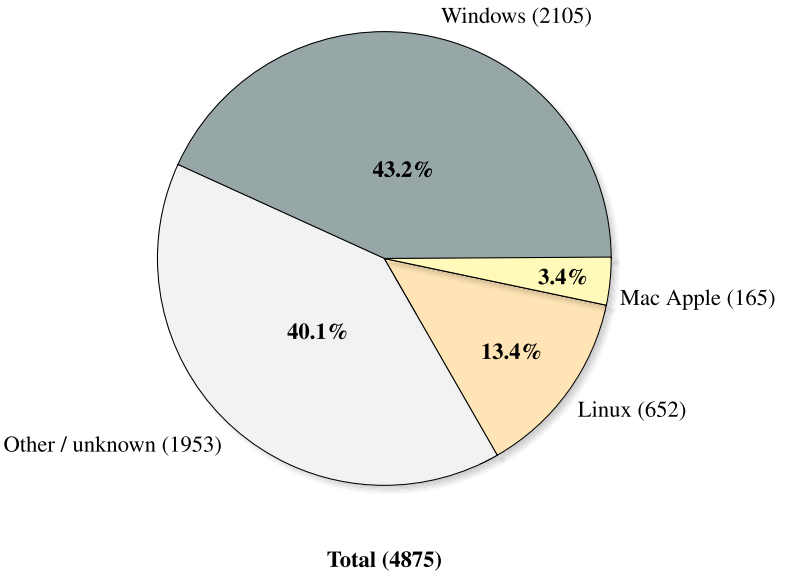


Fig. 2.15 GPUMLib downloads according to the operating system

License and its source code, documentation and examples can be obtained at `http://gpumlib.sourceforge.net/`.

GPUMLib does not intend to replace existing ML libraries such as WEKA (`http://www.cs.waikato.ac.nz/ml/weka/`) or KEEL (`http://www.keel.es/`) duplicating the work that has already been done, but rather to complement them. In this sense we envision the integration of GPUMLib in other ML libraries and we expect to provide the tools necessary for its integration with other ML software at a later phase.

Since its release, GPUMLib has attracted the interest of numerous people (see Figure 2.14), using a wide-range of platforms (see Figure 2.15), benefiting researchers all over the world (see Figure 2.16). Moreover, GPUMLib has received a 5 star award from the soft82.com editors, which according to them is given to products that are considered to be excellent and above average in their category.

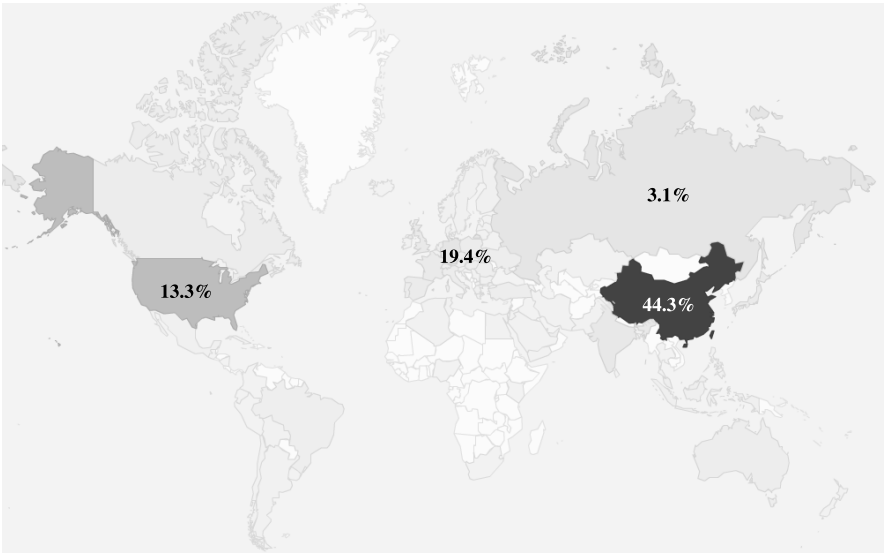


Fig. 2.16 GPUMLib downloads per country. Regions with a higher number of downloads are represented with darker colors.

2.6 Conclusion

This chapter has introduced the GPU as a novel and compelling solution to cope with the increasingly real-time sources of data for which ML algorithms demand expensive computational resources. Despite the increasingly availability of computational resources currently tools have been outpaced by the ever-demanding complex tasks to solve, therefore parallel implementations of ML algorithms become crucial for developing real-world applications. In conformity with this scenario, the GPU is particularly well positioned to fulfill this need, given its

availability, high-performance and relative low-cost [129]. However, developing programs for the GPU is significantly much harder than for traditional architectures. Hence, researchers may not have the skills or the time required to implement algorithms from scratch on this platform. To lessen this problem, in this chapter we presented a new open-source GPU ML library (GPUMLib) that can efficiently take advantage of the GPU parallel architecture and is able to provide considerable speedups, allowing to easily select the building blocks necessary to create ML software [139, 129].

In parts Parts II and III of this book we detail the GPUMLib parallel implementations of the algorithms developed and compare and analyze its results with their standalone counterparts.

Part II
Supervised Learning

Chapter 3
Neural Networks

Abstract. In this chapter we review the basic aspects of Neural Networks (NNs) and investigate techniques for reducing the amount of time necessary to build NN models. With this goal in mind, we present the details of a GPU parallel implementation of the Back-Propagation (BP) and Multiple Back-Propagation (MBP) algorithms. In particular, regarding the CUDA implementation of the BP and MBP algorithms we include both the adaptive step size and the robustness techniques which overall improve the algorithms stability and training speed. In the sequel, the training process is shown to be decomposed into three sequential phases: forward, robust learning and back-propagation. For each stage we point out the details for designing efficient kernels and provide the respective models of execution. Despite the benefit through the use of GPU, an automatic generator of topologies Autonomous Training System (ATS) algorithm is given. The approach tries to mimic the heuristics that are usually employed for model selection in a step-by-step constructive based error evaluation. Its advantage is to reduce significantly the effort necessary for building NN models. A final experimental section supports the effectiveness of the proposed systems. Finally, the software configuration parameters, the results and discussion on benchmarks and real case studies are presented.

3.1 Back-Propagation (BP) Algorithm

Despite being motivated by the parallel processing capabilities of the human brain, artificial NNs (referred in this book simply by NNs) have little in common with their biological counterparts [174, 57]. Nevertheless, over time, they have proven to be able to solve complex problems in many different domains (e.g. medical diagnosis, speech recognition, economics, business, image processing, intelligent control, time series prediction, chemical industry, computing, engineering, environmental science and nanotechnology) and new applications are continuously being found [106, 94, 223, 196, 236, 241]. Unfortunately, building an NN solution is a computationally

N. Lopes and B. Ribeiro, *Machine Learning for Adaptive Many-Core Machines – A Practical Approach,* Studies in Big Data 7,
DOI: 10.1007/978-3-319-06938-8_3,

expensive task, which often requires a substantial amount of time and effort. In particular, in relation to BP and MBP algorithms, depending on the complexity of the problem, in most cases several NNs, with different configurations, must be trained before achieving a good solution. This is a drawback, especially for challenging and computationally demanding problems involving large datasets, where the long training times alone may prevent high quality solutions from being found [133, 123]. Hence, creating GPU implementations of these algorithms is highly desirable.

The BP is one of the most well-known and extensively used ML algorithms. In fact, it is so successful that over 90% of the real-world commercial and industrial NN applications use this algorithm [157, 255].

3.1.1 Feed-Forward (FF) Networks

The BP is a supervised learning algorithm for training Feed-Forward (FF) NNs. These networks, also called Multi-Layer Perceptrons (MLPs), or BP networks (when trained with the BP algorithm), are comprised of several interconnected processing units (neurons) organized in layers, such that the information flows exclusively in a forward direction (from the input layer to the output layer). In other words, there are no connections between any two neurons within the same layer or from the units of any layer to the units of previous layers. Commonly, these networks present two or three layers of processing units, in which the neurons of each layer are fully-connected to the neurons of the posterior layer. Figure 3.1 presents the typical architecture of a three-layer FF network. Note that since the sole purpose of the units within the input layer consists of transferring/distributing the input data to the neurons in the next layer (no processing is carried out by these units), this particular layer is not considered when determining the number of layers, l, of a network.

Each connection defines the direction and flow of information between two neurons, i and j, as illustrated in Figure 3.2. From the point of view of neuron i this is an output connection while from the point of view of neuron j this is an input connection. Each connection has an associated weight, W_{ij}, which defines its strength, allowing the original input signals to be amplified or shrink according to its value. Typically, the input signals, which correspond to the previous layer neuron outputs, are multiplied by the connection weight, before being further processed. Therefore, since the connections model the effect of the original signal in the neurons, connections are said to be excitatory for positive weight values and inhibitory for negative values. Moreover, when a connection weight is zero, it becomes irrelevant in the context of the network, since no information will actually flow between the two neurons.

Each neuron, j, starts by gathering the information signals, fed by the previous neurons, through its input connections, which is then summed together with a bias, b_j, in order to compute its activation, a_j. The bias can be considered to

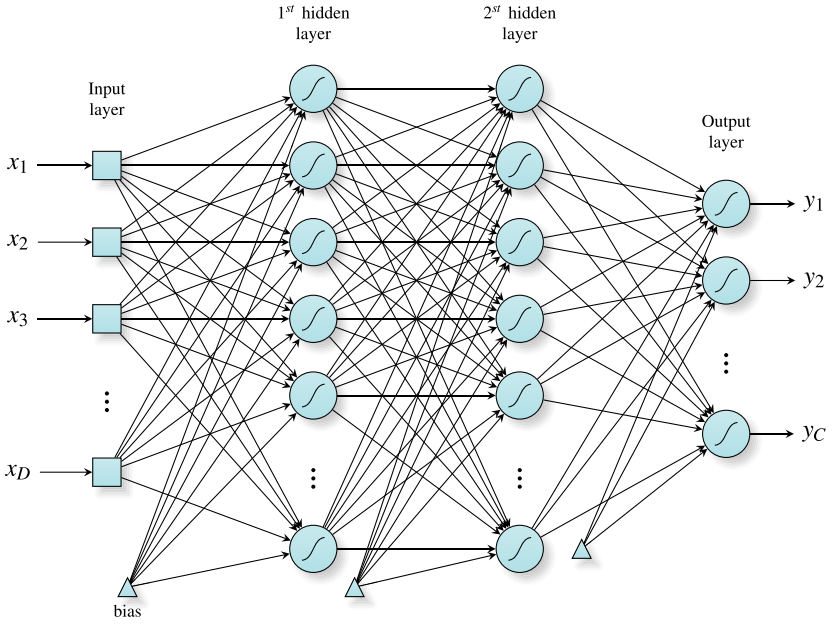


Fig. 3.1 Three-layer feed-forward network

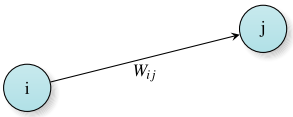


Fig. 3.2 Connection between two neurons

be the weight of an extra connection, whose input signal remains constantly set to 1. This allows the BP algorithm to adjust the bias weight together with the remaining weights [18, 174]. Accordingly, assuming that the neuron j contains I input connections, its activation, a_j, is given by (3.1):

$$a_j = \sum_{i=0}^{I} W_{ij} y_i \,. \tag{3.1}$$

The activation is then used to compute a single neuron output value, y_j, by applying a typically non-linear activation/transfer function to the computed activation, a_j, and the result is sent to the subsequent units through the output connections. Figure 3.3 illustrates this process.

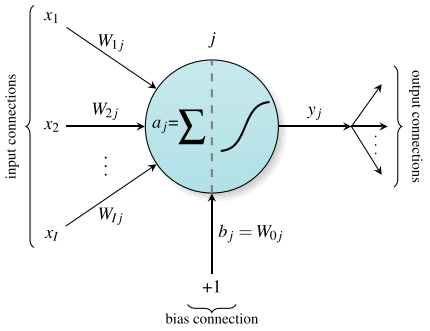


Fig. 3.3 Neuron architecture

Accordingly, the output, y_j, of neuron j is given by (3.2):

$$y_j = \phi(a_j) = \phi(\sum_{i=0}^{I} W_{ij} y_i) \,, \tag{3.2}$$

where $\phi(x)$ is the neuron activation function. Typically, the (logistic) sigmoid function, given by (3.3), is used for this purpose:

$$\sigma(x) = \frac{1}{(1+e^{-x})} \,. \tag{3.3}$$

The sigmoid is a non-linear and non-decreasing function, whose output is clamped between 0 and -1. Figure 3.4 presents a graphical plot of this function. Note that we can actually use any other activation function, provided that it has a first derivative. Nevertheless, the activation function has a huge impact on the complexity and performance of both the BP algorithm and the resulting models [209, 174, 57] as it reshapes the geometry of the transformations generated by the networks with implications in the training speed and generalization capabilities [174]. Moreover, the activation function is strongly correlated with the number of adaptive parameters required to model complex decision borders [57]. Therefore its choice must be careful pondered. In this context, the sigmoid is the most interesting and commonly used activation function because: (*i*) it significantly outperforms other functions providing better generalization models; (*ii*) it requires less training time than most functions; and (*iii*) computing its derivate is a very fast and straightforward process [209]. In addition, it is generally believed that the activity of biological neurons is regulated by a sigmoidal transfer function [57].

An FF network with a single hidden layer of units with continuous non-linear sigmoidal activation functions is a universal approximator, i.e. it can learn any arbitrary measurable function with the desired degree of accuracy, provided that a sufficient number of neurons is specified [91, 53, 64].

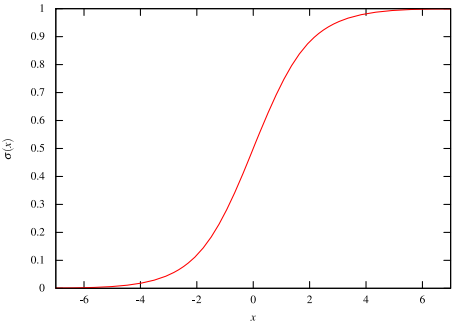


Fig. 3.4 Sigmoid function

3.1.2 Back-Propagation Learning

Two phases may be distinguished when training a network with the BP algorithm: a forward and a backward phase. In the forward phase, also called forward-propagation [18], the input layer distributes the incoming signals to the next layer, which in turn computes its outputs and sends the resulting signals to the subsequent layer, and so on until the results of the output layer corresponding to the model outputs are finally produced. At this point the error, E, between the targets (desired outputs), $\mathbf{t}$, and the actual network outputs, $\mathbf{y}$, can be computed. Usually this is accomplished with the quadratic error function[1]:

$$E = \frac{1}{2}\sum_{o=1}^{C}(t_o - y_o)^2 \,, \tag{3.4}$$

In the backward (back-propagation) phase, the errors of the network are propagated backwards, layer by layer, starting at the output layer, so that the weights of the input connections of each layer are adjusted to reduce the error between the network outputs and the corresponding targets. The weights are adjusted in an iterative process, according to the gradient descent rule, i.e. in the opposite direction of the gradient of the error function with respect to the network weights. Hence, using η as a learning rate factor, the weight change, ΔW_{ij}, is given by (3.5):

$$\Delta W_{ij} = -\eta \frac{\partial E}{\partial W_{ij}} \,. \tag{3.5}$$

Since the error, E, depends indirectly on W_{ij} through the neuron activation, a_j, we can apply the chain rule for the partial derivatives in order to obtain (3.6):

[1] Online training mode is considered.

$$\begin{aligned}\Delta W_{ij} &= -\eta \frac{\partial E}{\partial a_j}\frac{\partial a_j}{\partial W_{ij}} \\ &= -\eta \frac{\partial E}{\partial a_j} y_i \,.\end{aligned} \tag{3.6}$$

By defining the local gradient δ_j as in (3.7):

$$\delta_j = -\frac{\partial E}{\partial a_j}\,, \tag{3.7}$$

we can write (3.6) as (3.8):

$$\Delta W_{ij} = \eta \delta_j y_i \,. \tag{3.8}$$

Using the chain rule once again, (3.7) can be written as (3.9):

$$\begin{aligned}\delta_j &= -\frac{\partial E}{\partial y_j}\frac{\partial y_j}{\partial a_j} \\ &= -\frac{\partial E}{\partial y_j}\phi'(a_j) \,.\end{aligned} \tag{3.9}$$

Thus, for an output neuron, $j = o$, the local gradient is given by (3.10):

$$\delta_o = (t_o - y_o)\phi'(a_o) \,. \tag{3.10}$$

In the case of a hidden neuron, $j = h$, its output is actually contributing to the errors of all the neurons, in the next layers, that share the same (input) connections as the neuron h output connections. Therefore, assuming the next layer is the output layer, the local gradient, δ_h, can be obtained by applying the chain rule [18]:

$$\begin{aligned}\delta_h &= -\phi'(a_h)\frac{\partial E}{\partial y_h} \\ &= -\phi'(a_h)\sum_{o=1}^{C}\frac{\partial E}{\partial a_o}\frac{\partial a_o}{\partial y_h} \\ &= -\phi'(a_h)\sum_{o=1}^{C}\frac{\partial E}{\partial a_o}\frac{\partial}{\partial y_h}\sum_{i=0}^{I} W_{io} y_i \\ &= -\phi'(a_h)\sum_{o=1}^{C}\frac{\partial E}{\partial a_o} W_{ho} \,.\end{aligned} \tag{3.11}$$

and using (3.7) we can finally write (3.11) as (3.12):

$$\delta_h = \phi'(a_h)\sum_{o=1}^{C}\delta_o W_{ho} \,. \tag{3.12}$$

Together (3.8), (3.10) and (3.12) allow to recursively update all the weights in the network. In practice, however, a momentum term, $0 \leq \alpha < 1$, is usually included in

(3.8) in order to improve the algorithm's convergence. Accordingly, instead of (3.8) we can use (3.13):

$$\Delta W_{ij} = \eta \delta_j y_i + \alpha \Delta W_{ij}^{(\text{old})} . \tag{3.13}$$

The momentum has a stabilizing effect when the gradient component, $\frac{\partial E}{\partial W_{ij}}$, oscillates in consecutive updates and an accelerating effect when it presents the same sign, see (3.5). Overall, this simple modification (to the weight update equation) not only accelerates the training process but also prevents the learning procedure from getting trapped in a shallow local minimum of the error manifold [82].

Another simple acceleration technique, although memory consuming, is the adaptive step size technique, which consists of using an individual learning rate (step size) parameter, η_{ij}, for each weight connection, W_{ij}, instead of a global learning rate. At each iteration, the step sizes η_{ij} are adjusted according to the successive signs of the gradient components, using (3.14) [5]:

$$\eta_{ij} = \begin{cases} u\eta_{ij}^{(\text{old})} & \text{if } (\frac{\partial E}{\partial W_{ij}})(\frac{\partial E}{\partial W_{ij}})^{(\text{old})} > 0 \\ d\eta_{ij}^{(\text{old})} & \text{if } (\frac{\partial E}{\partial W_{ij}})(\frac{\partial E}{\partial W_{ij}})^{(\text{old})} < 0 \end{cases} \tag{3.14}$$

where $u > 1$ (up) represents the increment factor for the step size and $d < 1$ (down) the decrement factor. When two consecutive updates have the same direction the step size of that particular weight is increased. For updates with opposite directions the step size is decreased, thus avoiding oscillations in the training process due to excessive learning rates. The underlying idea of this procedure consists of finding near-optimal step sizes that would allow bypassing ravines on the error surface. This technique is especially effective for ravines that are (almost) parallel to some axis [5].

Even though the step sizes are reduced in the presence of oscillations (gradient components with opposite directions), it is possible, under certain circumstances, for the step sizes to become excessively large. This results in the increase of the error cost function, from one epoch to another. A similar increase may also occur in a curved ravine when too much momentum is acquired [5]. To avoid both situations, it is possible to implement a robustness method that is triggered when the error grows over a predefined threshold (e.g. 0.1%). This method basically consists of [5]: (*i*) setting the weights back to the values they had in the epoch with the lowest error achieved so far; (*ii*) reducing the step size parameters by a pre-defined factor, $0 < r < 1$, (e.g. 0.5) and (*iii*) set the momentum memories to zero.

The combined procedure (adaptive step size, robustness method and momentum) results in a very effective training algorithm that works well in practice [5].

3.2 Multiple Back-Propagation (MBP) Algorithm

An NN can be viewed as an adaptive black box model whose parameters (weights) are adjusted by the training procedure, so that the network as a whole acts as a

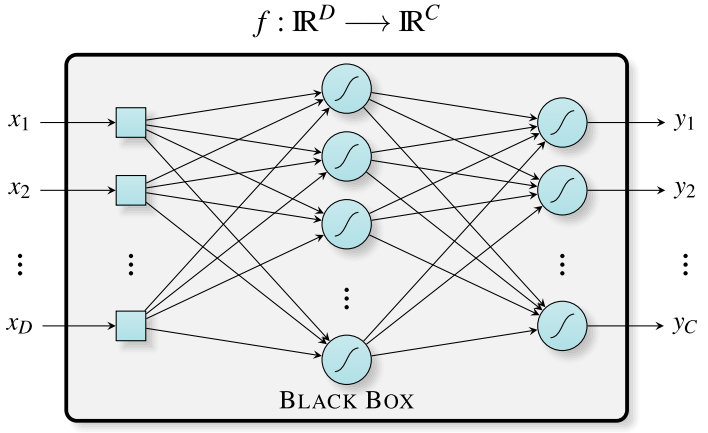


Fig. 3.5 A neural network viewed as a black box system that maps D inputs into C outputs

mapping function, $f : \mathbb{R}^D \longrightarrow \mathbb{R}^C$, that attempts to fit the observed (training) data (see Figure 3.5).

Rationally, we want the resulting NN model to resemble as close as possible the subjacent model that governs the real data distribution. Hence, having a single model that covers all the input space might not be the best solution. In particular, for complex problems a divide and conquer strategy may be more appropriate. Thus, it might be preferable to create several localized models that can take advantage of the specific characteristics of their operating domain and that when combined together could mimic better the model governing the true data distribution. In other words, it may be possible to obtain a better fitting model with improved generalization by using different mapping functions, each covering a specific region of the input space. A similar principle is used by ensemble methods (also referred to as committees of classifiers), which combine several sub-models in order to create classifiers that often present better generalization performance than their constituent models, provided that the integrated classifiers are diverse and accurate [58, 240, 221]. In particular, in the Mixture of Experts (ME) architecture, the outputs of a set of experts (models) are combined in a hierarchical modular structure, by using a gating network that divides the input space into a set of nested regions [254, 58]. Both the gate and the expert parameters are estimated separately, typically using the Expectation-Maximization (EM) algorithm, such that the gate will create a soft division of the input space in which each expert is assigned to a specific partition region [254].

Given enough information about the problem being tackled, it is possible to divide the input space into several regions of interest (e.g. different operating model regimes) and associate to each one a specific tailored model. This can be viewed as if we decompose f in several simpler sub-functions. However, for the majority of

the cases, such knowledge is not available and manually partitioning the input space becomes impractical.

Nevertheless, it is possible to partition the input space without the proviso of explicit information. For example, the RBF networks perform an implicit partition of the input space by assigning localized neurons that respond only to the samples that are in the vicinity of its center. From the point of view of a specific sample (pattern) it is as if all the other neurons, whose center is further away, did not exist. This can be viewed as if different groups of similar patterns (in the same space partition) have associated their own network.

Biological arguments also favor this idea. The human brain contains highly-specialized regions, responsible for dealing specifically with certain cognitive aspects. In particular, there are cortical regions specialized not only for basic sensory and motor processes but also for the high-level perceptual analysis that will selectively react to single categories of visually presented objects (e.g. faces, places, bodies, words) [101].

3.2.1 Neurons with Selective Actuation

In the same manner that the brain engages distinct areas (neurons) to respond to different *stimuli*, it would be useful for (artificial) NNs to activate distinct neurons in response to different *stimuli*. This would allow neurons to become specialized in certain patterns, reacting only when confronted with them, while ignoring the rest. The idea consists of activating a different set of neurons for each similar set of patterns, allowing the input space to be divided into several parts, thus creating and associating different virtual network models (for each group of similar *stimuli*) that share the same infra-structure.

Using this idea, we want neurons to be able to react distinctly to each pattern. Hence, we can define the contribution of a given neuron, j, to the network output, by incorporating an importance factor, m_j, in equation (3.2), which specifies its relevance for the sample (*stimulus*) being presented to the network. Such neurons are designated by neurons with selective actuation[122, 121] and the equation governing its output is given by (3.15):

$$y_j = m_j \phi(a_j) = m_j \phi(\sum_{i=0}^{I} W_{ij} y_i) \,. \tag{3.15}$$

The farther from zero m_j is, the more important the neuron (contribution) becomes. On the other hand, when m_j is zero the neuron becomes completely irrelevant and one can interpret such a value as if the neuron is not present in the network [122, 121].

Notice that if we consider all the importance factors m_j to be constant and equal to one, i.e., if all neurons are equally important to the network regardless of the

pattern being presented, then (3.15) becomes identical to the standard neuron output equation (3.2).

Figure 3.6 shows two alternative representations of a neuron with selective actuation. As we shall see, the actual contribution of these neurons to the network outputs is fine-tuned according to the space localization of the samples presented to the network. Therefore, they can become specialized in a specific set of samples belonging to space regions in which they present high-importance values, while ignoring the remaining samples localized in other (low-importance) regions.

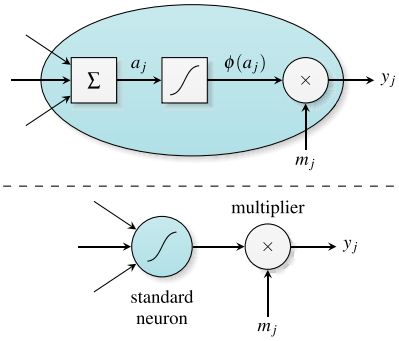


Fig. 3.6 Selective actuation neuron architecture

A neuron with selective actuation is inherently a non-linear device that is capable of solving the XOR problem, even when its constituent neurons use a linear activations function (see Figure 3.7).

3.2.2 Multiple Feed-Forward (MFF) Networks

The importance factors, m_j, are determined by a so-called space network that receives the same inputs as the network with selective actuation neurons (main network). The latter can only calculate its outputs after the space network outputs, m_j, are evaluated. Thus the two networks must function in a collaborative manner, as a whole, and therefore must be trained together. Note that both selective actuation neurons and standard neurons can coexist in the main network [122, 121].

The resulting network is called a Multiple Feed-Forward (MFF) or an MBP network. Figure 3.8 illustrates the relationship between the two networks that integrate an MFF network.

By computing the importance factors of the main network, the space network is implicitly dividing the input space, creating seamless partitions that have associated different models. Hence, from the point of view of a selective actuation neuron

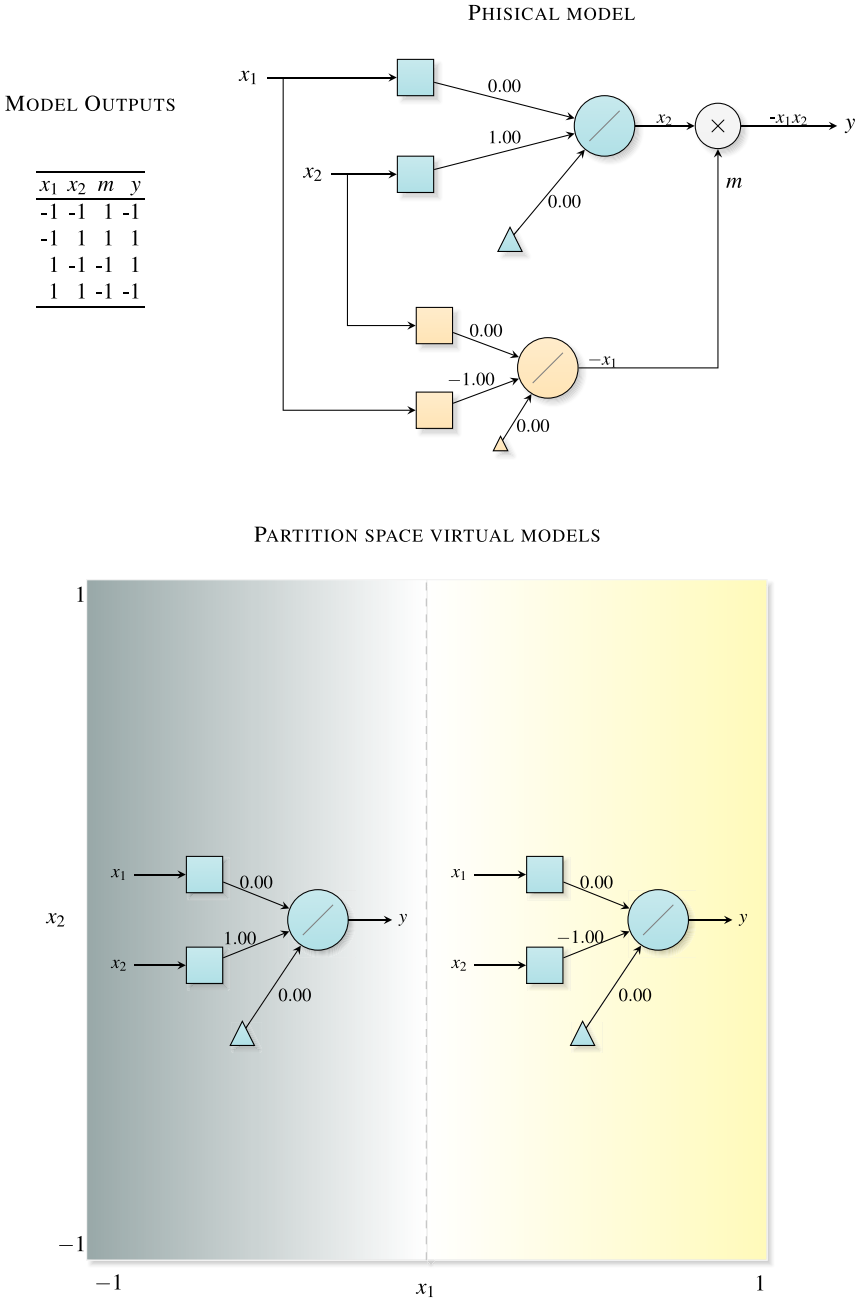


Fig. 3.7 Architecture of a selective actuation neuron, with linear activation functions, which solves the XOR problem

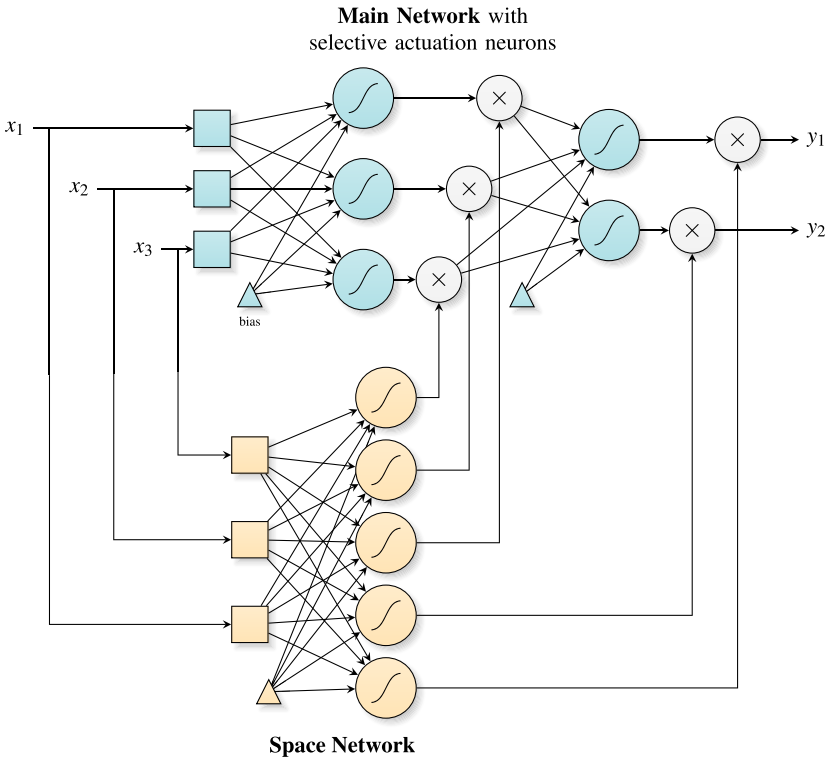


Fig. 3.8 Example of a multiple feed-forward network

integrating those models, some data points can be interpreted as being more important than others [122, 121].

3.2.3 Multiple Back-Propagation (MBP) Algorithm

MFF networks have two contributions for their output errors: (*i*) the weights of the main network; and (*ii*) the weights of the space network (or in other words the importance given to each neuron with selective actuation). Therefore, minimizing the error E between the target outputs and the MFF outputs implies adjusting the weights of both networks. To this end, an algorithm named Multiple Back-Propagation (MBP) was devised [122, 121].

In the MBP algorithm, the main network weights are adjusted according to the gradient descent method, using (3.8) as in the BP algorithm. However, due to the introduction of the importance factor, the local gradients for the output

neurons δ_o and for the hidden neurons δ_h are now respectively given by (3.16) and (3.17):

$$\delta_o = (t_o - y_o) m_o \phi'(a_o) \,, \tag{3.16}$$

$$\delta_h = m_h \phi'(a_h) \sum_{o=1}^{C} \delta_o W_{ho} \,. \tag{3.17}$$

Jointly, (3.8), (3.16) and (3.17) recursively allow to adjust the main network weights. Note that, once again, if m_j is constant and equal to one, these equations are identical to the corresponding BP equations (see (3.10) and (3.12)). Thus, MBP can be considered as a generalization of the BP algorithm [122, 121].

As said before, in order to minimize the errors it is necessary to adjust the weights of the space network as well. By doing so, we are changing the soft division of the input space, seeking a more suitable and proper partition.

In order to adjust space network weights, the variation of the importance factors can be computed by using the gradient descent method as well, as stated in (3.18):

$$\Delta m_j = -\frac{\partial E}{\partial m_j} \,. \tag{3.18}$$

Considering the importance factor of an output neuron $o = j$ (of the main network), (3.18) can be written as (3.19):

$$\Delta m_o = (t_o - y_o) \phi(a_o) \,, \tag{3.19}$$

Regarding the importance factor of a hidden neuron $h = j$ (of the main network), (3.18) can be written as (3.20):

$$\begin{aligned} \Delta m_h &= -\sum_{o=1}^{C} \frac{\partial E}{\partial a_o} \frac{\partial a_o}{\partial m_h} \\ &= \sum_{o=1}^{C} \delta_o \frac{\partial}{\partial m_h} \sum_{i=0}^{I} W_{io} y_i \\ &= \sum_{o=1}^{C} \delta_o \frac{\partial}{\partial m_h} \sum_{i=0}^{I} W_{io} m_i \phi(a_i), \end{aligned} \tag{3.20}$$

and finally we obtain (3.21):

$$\Delta m_h = \sum_{o=1}^{C} \delta_o W_{ho} \phi(a_h) \,. \tag{3.21}$$

Employing (3.19) and (3.21), we can compute the desired values of the space network, using $m_j + \Delta m_j$, and then apply the BP algorithm to correct the weights of the space network.

Collectively, the MFF networks and the MBP algorithm compose an architecture that is in most cases preferable to the use of standard BP networks [122, 121] and has yielded good results in several applications, such as financial data analysis [28], electricity consumption forecasting [76], analysis of market orientation [210] and character recognition [39].

3.3 GPU Parallel Implementation

The CUDA implementation of the BP and MBP algorithms features both the adaptive step size and the robustness techniques described earlier (see page 45), which overall improve the algorithm's stability and training speed. Accordingly, the training process can be decomposed in three sequential phases (per epoch): forward, robust learning and back-propagation.

3.3.1 Forward Phase

The forward phase is implemented by two kernels (`FireLayer` and `FireOutputLayer`) whose objective consists of calculating the outputs of a given layer. This phase proceeds as follows: If a space network exists, `FireLayer` is called for each one of its layers, until the space network outputs are determined. Then `FireLayer` is called, once again, for each one of the hidden layers of the main network and finally, `FireOutputLayer` is called to determine the main network outputs. Figure 3.9 illustrates the sequence (from left to right) in which the host calls these kernels, considering an MBP network encompassing a hidden layer with J selective actuation neurons and an output layer with C standard neurons. A single layer is considered for the space network.

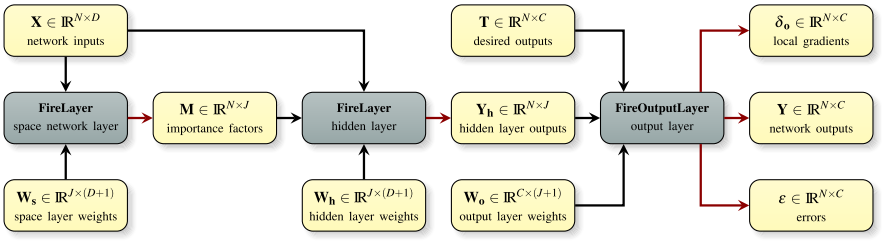


Fig. 3.9 Model of the kernels executed (in each epoch) to complete the forward phase of an MBP network

Regarding the batch training mode, there are two sources of parallelism: First the outputs of the neurons, within a layer, can be computed in parallel; and second, the training samples can be processed independently. Accordingly, the kernels were designed to operate on a generic network layer with J neurons, calculating their outputs for all the N samples. Thus, considering the neuron as the smallest processing element of an NN, we would end up with NJ processing threads. However, for many problems this number is insufficient because as we said before, CUDA requires a large number of threads, running simultaneously, in order to hide memory latency efficiently. Moreover, there are many situations where the output layer consists of a single neuron, in which case we would execute only N threads when processing that layer. Thus, one needs to define threads at a much finer granularity to take full advantage of the GPU high number of cores [194] (see Section 2.4.2, page 28). Although conceptually the neuron is the smallest processing element of an NN, in order to increase the number of threads we need to use a different approach. Namely, it is actually possible to perform a simple computation at the connections level: each connection can multiply its weight by the incoming input ($W_{ij}y_i$). By doing so, we manage to increase the number of threads by a factor of $(I+1)$, where I is the number of inputs of the layer. Moreover, we can take advantage of the fast shared memory to sum up the values computed by each thread (connection) within the block (neuron), using a reduction process and then computing the output of the neuron for the active sample.

Listing 3.1 shows the version of the `FireLayer` kernel that is used when the number of connections (including the bias) does not exceed 512 (due to performance reasons, there are two versions for each one of the `FireLayer` and `FireOutputLayer` kernels). All the matrices are in row-major order, to keep the kernel memory accesses coalesced.

Note that the forward phase could be implemented without using the `FireOutputLayer` kernel. However, we noticed that part of the data needed to calculate the local gradients (of the output layer) and the Root Mean Square Error (RMSE) of the network was already in the SM registries and shared memory (see Figure 3.9 and Listing 3.1) which are significantly faster than the global (device) memory. Thus, this kernel was created to increase the performance of the resulting implementation by taking advantage of the data already present in the SMs. Besides calculating the layer outputs, $\mathbf{Y} \in \mathbb{R}^{N\times C}$, this kernel also computes the local gradients of the output layer, $\delta_{\mathbf{o}} \in \mathbb{R}^{N\times C}$, the local gradients of the space network neurons, $\delta_{\mathbf{s}} \in \mathbb{R}^{N\times C}$ (associated with the selective actuation neurons in the output layer) and the (square of the) neurons' errors, $\varepsilon \in \mathbb{R}^{N\times C}$, where each element (corresponding to sample n and output o) is given by $\varepsilon_{no} = (T_{no} - Y_{no})^2$.

Listing 3.1 FireLayer kernel.

```
#define INPUT threadIdx.x
#define N_INPUTS_INCL_BIAS blockDim.x
#define N_INPUTS (N_INPUTS_INCL_BIAS - 1)
#define BIAS 0
#define NEURON threadIdx.y
#define N_NEURONS blockDim.y
#define PATTERN blockIdx.x

__device__ void sumInputWeight(int c, float * x, float * w) {
    extern __shared__ float xw[];

    xw[c] = w[c];
    if (INPUT > BIAS) {
        xw[c] *= x[PATTERN * N_INPUTS + (INPUT - 1)];
    }
    __syncthreads();

    int es = N_INPUTS_INCL_BIAS;
    for(int s = (es >> 1); es > 1; s = (es >> 1)) {
        int nextNumberElemSum = s;
        if (es & 1) nextNumberElemSum++;
        if (INPUT < s) xw[c] += xw[c + nextNumberElemSum];
        es = nextNumberElemSum;
        __syncthreads();
    }
}

__global__ void FireLayer(float * x, float * w, float * m,
    int moffset, int selNeurons, float * y) {
    extern __shared__ float xw[];

    int conn = NEURON * N_INPUTS_INCL_BIAS + INPUT;
    sumInputWeight(conn, x, w);

    if (INPUT == 0) {
        int n = PATTERN * N_NEURONS + NEURON;
        float o = CUDA_SIGMOID(xw[conn]);
        if (m != NULL) {
            o *= m[PATTERN * selNeurons + NEURON + moffset];
        }
        y[n] = o;
    }
}
```

3.3.2 Robust Learning Phase

In this phase, we start by calculating the RMSE, using the `CalculateRMS` kernel which receives the network errors, $\varepsilon \in \mathbb{R}^{N \times C}$, previously produced by the `FireOutputLayer` kernel.

The host will then decide whether or not to stop the training process. However, transferring information between the device and the host (and vice-versa) is particularly time-consuming. Thus, the training process can not be put on hold while waiting for the host to receive the error. Depending on the size of the dataset and the particular device being used, several training epochs might occur (with tens of kernels being called) during the interval of time necessary for the host to obtain the RMSE. Hence, the host will actually base its decision on an estimate (old value), which is periodically obtained by asynchronously querying the RMSE, instead of relying on the actual RMSE value.

If the host decides to continue the training, the `RobustLearning` kernel is then called, to improve the stability and convergence of the algorithm. This kernel compares the current (device) RMSE with the best error found so far. If the current error is smaller then (*i*) the best RMSE is updated and (*ii*) the NN weights are stored. Otherwise, if the error exceeds a given threshold: (*i*) the best weights, found so far, are restored, (*ii*) the step sizes multiplied (reduced) by the robustness factor, r, and (*iii*) the momentum memories are set to zero.

3.3.3 Back-Propagation Phase

The back-propagation phase is supported mainly by two kernels: `CorrectWeights` and `CalcLocalGradients`. The latter determines the local gradient of the neurons of a hidden layer, $\delta_{\mathbf{h}} \in \mathbb{R}^{N \times J}$. If there are neurons with selective actuation, the corresponding local gradients, $\delta_{\mathbf{s}} \in \mathbb{R}^{N \times J}$, of the space network are also calculated. The task of the `CorrectWeights` kernel consists of adjusting the weights, $\mathbf{W_h} \in \mathbb{R}^{J \times (D+1)}$, and the corresponding step sizes, $\eta_{\mathbf{h}} \in \mathbb{R}^{J \times (D+1)}$, of a given layer.

Essentially the back-propagation phase proceeds as follows: First, the kernel `CalcLocalGradients` is repeatedly called in order to determine the local gradients of each hidden layer, starting in the last hidden layer and proceeding backwards until the local gradients of the first hidden layer are known (note that the local gradients of the output layer were already determined by the `FireOutputLayer` kernel). To complete the training process, for the current epoch, the `CorrectWeights` kernel must be called for each layer (of both the main and space networks) in order to adjust its weights. It is important to notice that these will only execute their code if the RMSE has not surpassed a pre-defined threshold (see the, previous, robust learning phase Section). This is necessary, because, as we state before, the host has no direct access to the information on the device and transferring it would heavily downgrade the performance of the algorithm. Thus, the host will always call those kernels and they will abort execution if needed. Figure 3.10 illustrates the sequence

(from left to right, top to bottom) in which the kernels are called by the host, in order to perform the back-propagation phase, for the same network that was considered in Figure 3.9.

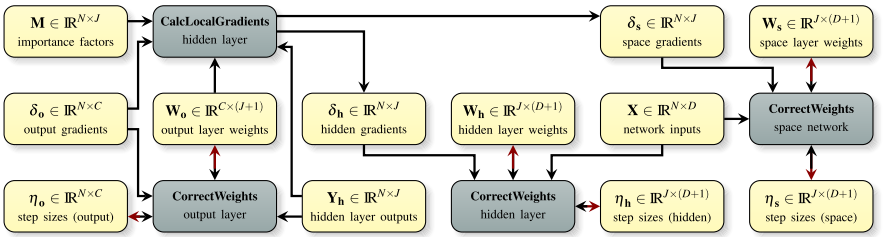


Fig. 3.10 Model of the kernels executed (in each epoch) in the back-propagation phase of an MBP network

3.4 Autonomous Training System (ATS)

Although the GPU can reduce significantly the time required for training NNs, building high-quality solutions still requires a large amount of effort and time. In particular, finding an adequate network topology can be a tedious and difficult process. Typically, several NNs, with different configurations, must be trained before achieving a good solution. Thus, the quality of the resulting system depends largely on the effort spent on this task. In this context, an Autonomous Training System (ATS) that actively searches for better solutions, adjusting the topology as needed, can be a very important tool for improving the quality of the resulting NN systems.

The proposed ATS is specifically designed for classification problems. However, it can easily be adjusted for regression problems. The ATS makes use of the GPU parallel implementation, described in the previous Section. It trains several NNs while adjusting their topology to improve the quality of the solutions found. The system starts by training an NN using the initial set up configuration and topology. Thereafter, the ATS evaluates the resulting (NN) model and adjusts conveniently the number of hidden neurons. A new NN is subsequently trained and its performance is compared with the best NN found so far. These results are in turn used to determine the number of hidden neurons of the next NN and the process is repeated until the stopping criteria is met.

Algorithm 1 presents the ATS algorithm for pattern recognition, which uses the sum between the number of false positives (fp) and false negatives (fn) to assert the quality of the networks. Note that we can easily replace this metric by another one that is more adequate (e.g. sensitivity, RMSE), according to the particularities of the problem being tackled. Each time the ATS trains an NN the resulting information is

Algorithm 1 Autonomous Training System.

Input: d_{train} ▷ Training dataset.
Input: d_{test} ▷ Test dataset.
Input: $nets_{train}$ ▷ Number of networks to train.
Input: h_{ini} ▷ Initial number of hidden neurons.
$fpn_{best} \leftarrow \infty$
$nets_{trained} \leftarrow 0$
$inc \leftarrow 0$
$h \leftarrow h_{ini}$
$h_{best} \leftarrow h$
$direction \leftarrow down$
repeat
    $network \leftarrow$ **new** network with h hidden neurons
    Train($network$, d_{train})
    $nets_{trained} \leftarrow nets_{trained} + 1$
    $fp \leftarrow$ FalsePositives($network$, d_{test})
    $fn \leftarrow$ FalseNegatives($network$, d_{test})
    $fpn \leftarrow fp + fn$
    Log $network$, fp, fn, RMSE($network$, d_{test})
    if $fpn \leq fpn_{best}$ **then**
        $fpn_{best} \leftarrow fpn$
        $h_{best} \leftarrow h$
        $inc \leftarrow inc + 1$
        SaveNetwork($network$)
    end if
    if $h < h_{best}$ **then**
        if $direction = down$ **then**
            $direction \leftarrow up$
            if $inc > 1$ **then** $inc \leftarrow inc - 1$
        end if
    else if $h > h_{best}$ **then**
        if $direction = up$ **then**
            $direction \leftarrow down$
            if $inc > 1$ **then** $inc \leftarrow inc - 1$
        end if
    end if
    if $direction = up$ **then**
        $h \leftarrow h + inc$
    else if $h > 1$ **then**
        $h \leftarrow h - inc$
    else
        $direction \leftarrow up$
    end if
until $nets_{trained} \geq nets_{train}$

logged and if the trained NN turns out to be better than the previous ones, it will be saved.

The proposed approach tries to mimic the heuristics that we use for model selection. Although far from perfect, it has proven to yield good results (see

Section 3.5) and constitutes a working basis on top of which new improvements can be build.

3.5 Results and Discussion

3.5.1 Experimental Setup

The experimental setup was conducted using the CUDA implementation, described earlier in Section 3.3, and the Multiple Back-Propagation software: a highly optimized application, developed in C++, for training NNs [125]. This software has been extensively tested and widely used by neural networks researchers and practitioners. Its latest version and source code, featuring our CUDA implementation, can be freely obtained at `http://mbp.sourceforge.net/`.

The CPU version was benchmarked using the computer system 1 with an Intel Core 2 running at 2.4 GHz and the GPU version on the computer systems 1 and 2, respectively with an 8600 GT and a GTX 280 devices (see Tables A.1 and A.2, on page 202, for more information on the systems and devices).

In our testbed experiments we compare the CPU and GPU versions of the algorithms on well-known benchmarks, as well as on a real-world case study, regarding the detection of Ventricular Arrhythmias (VAs). These are described in more detail in Appendix A.4, but we reproduce here, in Table 3.1, the main characteristics of their training datasets for readability purposes.

Table 3.1 Main characteristics of the training datasets used in the MBP experimental setup

Dataset (Benchmark)	**Samples (*N*)**	**Features (*D*)**	**Classes (*C*)**
Sinus cardinalis	101	1	1
Two-spirals	194	2	1
Sonar	104	60	1
Forest cover type	11,340	54	7
Poker hand	25,010	85	10
Ventricular arrhythmias	19,391	18	1

For a fair comparison of the algorithms, the initial weights of the NNs, trained with the CPU and the GPU, were set to identical random values. Furthermore, all the data was transferred to the GPU prior to the training.

With the exception of the *two-spirals* benchmark, all the networks presented in this study had a single hidden layer with J neurons. Moreover, in the case of the *sinus cardinalis* and *two-spirals* benchmarks we use the same topology configurations that we have used in Lopes and Ribeiro [122]. Hence, in the latter benchmark, the networks had a second hidden layer with 10 neurons.

In addition, only the first hidden layer of the main network (of the MBP networks) contained neurons with selective actuation. Finally, the step sizes, η, and the momentum terms, α, were initialized to 0.7; and whenever the robust learning technique was used, a reducing factor of 0.5 and a tolerance of 0.1% were chosen.

3.5.2 Benchmark Results

Concerning both the *sinus cardinalis* and the *two-spirals* benchmarks, we have trained 10 networks (for each configuration), until the RMSE was less than 0.01. Moreover, we have used the adaptive step size technique with an increment u of 1.05 and a decrement d of 0.95 (the robust training technique was not used). Tables 3.2 and 3.3 present the average GPU speedups and the corresponding standard deviations respectively for the *sinus cardinalis* and for *two-spirals* benchmarks.

Table 3.2 Speedups ($\times$) for the *sinus cardinalis* problem

Topology	Hidden units (J)	System 1 (8600 GT)	System 2 (GTX 280)
BP	7	3.98 ± 0.19	5.48 ± 0.21
	9	4.66 ± 0.16	7.15 ± 0.13
	11	5.44 ± 0.13	8.43 ± 0.15
MBP	5	4.37 ± 0.10	6.08 ± 0.13
	7	5.73 ± 0.07	7.99 ± 0.10
	9	6.77 ± 0.09	10.24 ± 0.12

Table 3.3 Speedups ($\times$) for the *two-spirals* problem

Topology	Hidden units (J)	System 1 (8600 GT)	System 2 (GTX 280)
BP	25	7.68 ± 1.01	32.84 ± 4.78
	30	7.96 ± 0.68	39.22 ± 3.36
	35	7.55 ± 0.42	39.61 ± 2.49
MBP	15	9.89 ± 0.76	32.85 ± 2.59
	20	9.75 ± 0.16	38.10 ± 0.70
	25	10.01 ± 0.31	42.98 ± 1.27

The results obtained show that the GPU implementation of the BP and MBP algorithms can in fact deliver enormous speedups over the corresponding CPU implementation.

In the *two-spirals* benchmark, the GTX 280 device can be over 40 times faster than the CPU. Results that took only 20 seconds on the GPU, required almost 15 minutes on the CPU. Figure 3.11 visually shows the performance improvements of the GPU over the CPU.

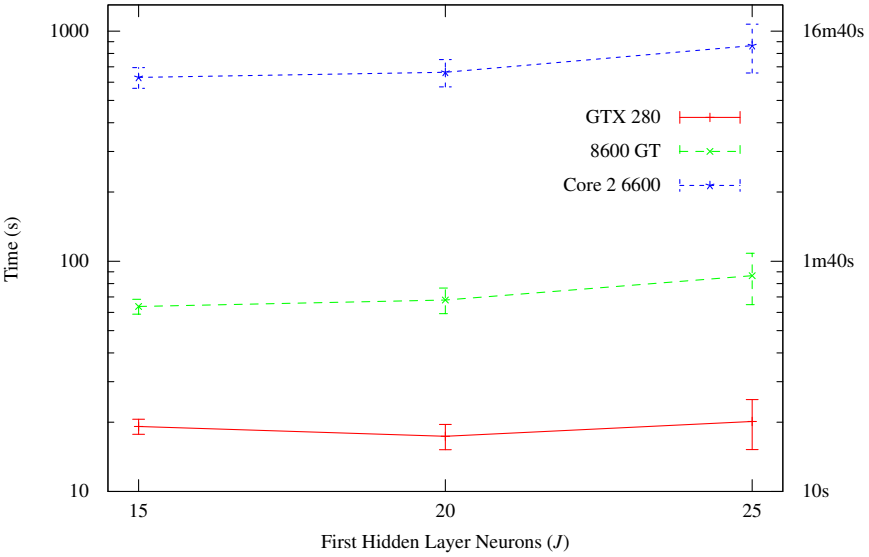


Fig. 3.11 *Two-spirals* training time (MBP algorithm)

In the *sinus cardinalis* benchmark, the results are not so impressive, because the reduced number of neurons and samples makes it less parallelizable. Consequently, many of the cores of the GTX 280 were idle during the training process and it was not possible to take complete advantage of this GPU. Thus, the difference between the analyzed devices is not so remarkable as in the *two-spirals* benchmark. On the other hand, this test confirms that, even for small problems, the GPU can provide a significant speedup (up to 10× in this case). Moreover, even when using an old model device (8600 GT), it is possible to obtain significant speedups (almost 7× in the *sinus cardinalis* and up to 10× in the *two-spirals* problems).

Although the training process starts at the same point of the error surface (identical weights were used for the CPU and GPUs), we will most likely end up on different (or slightly different) points of the error surface, according to the hardware used. There are two factors accountable for this situation. First, as we said before, transferring data from the GPU to the CPU is time consuming. Therefore, we use an estimate of the RMSE to avoid stopping the training process while the error is being transferred. As a result (on the GPU) the NNs will most likely be trained for a few more epochs after the desired error has been reached. Second, there are discrepancies between the floating-point results given by the CPU and those given

by the GPUs. Therefore, slightly different paths on the error surface might be taken depending on the gradient computation results. As a result, the number of epochs required to train an NN will (most likely) differ, depending on the hardware platform used.

Even though the GPU networks will be trained for more epochs than those needed to reach the desired error, our tests demonstrate that training the networks on the CPU does not always require less epochs. For example, in the *sinus cardinalis* benchmark the 8600 GT required less epochs than the CPU in 51.67% of the cases, whilst the GTX 280 required less epochs in 58.33% of the cases. In practice, for the vast majority of networks, the discrepancy of epochs is very small and only residually affects the speedups. Therefore, for the subsequent tests, we decided to use the number of epochs trained per minute, instead of the time required to reach a predefined RMSE. Moreover, in the remaining tests, 30 networks for each configuration were trained to establish confidence bounds and ensure statistical significance using standard deviation.

In the *forest cover* benchmark, we selected 7 out of the 14 different hidden neurons configurations, chosen in Blackard and Dean [19]. The robust learning technique and adaptive step size (with an increment u of 1.1 and a decrement d of 0.9) were used. Figures 3.12 and 3.13 show the number of epochs trained per minute for the *forest cover* problem, using respectively the BP and MBP algorithms.

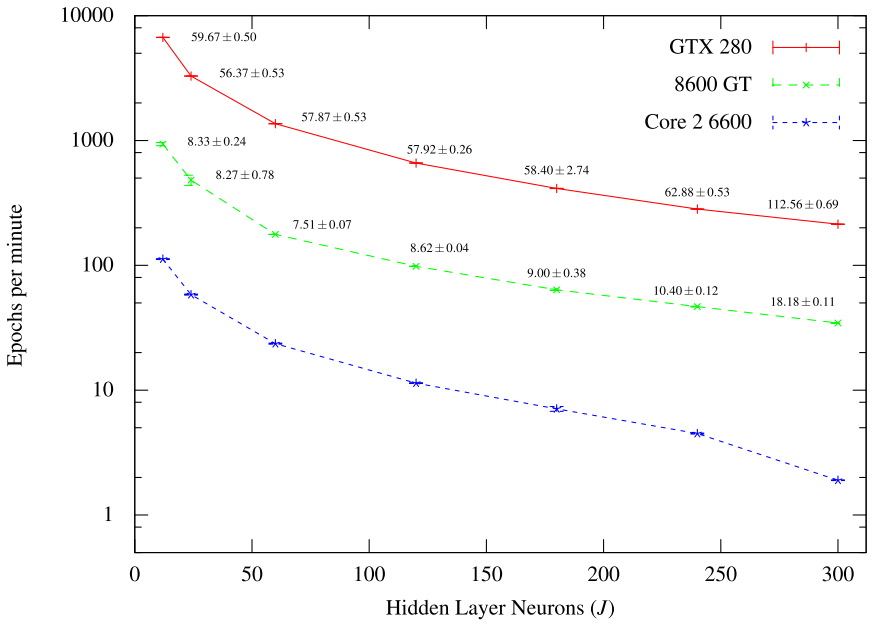


Fig. 3.12 Number of epochs per minute using the BP algorithm for the *forest cover* problem. The GPU speedups are shown near the corresponding lines.

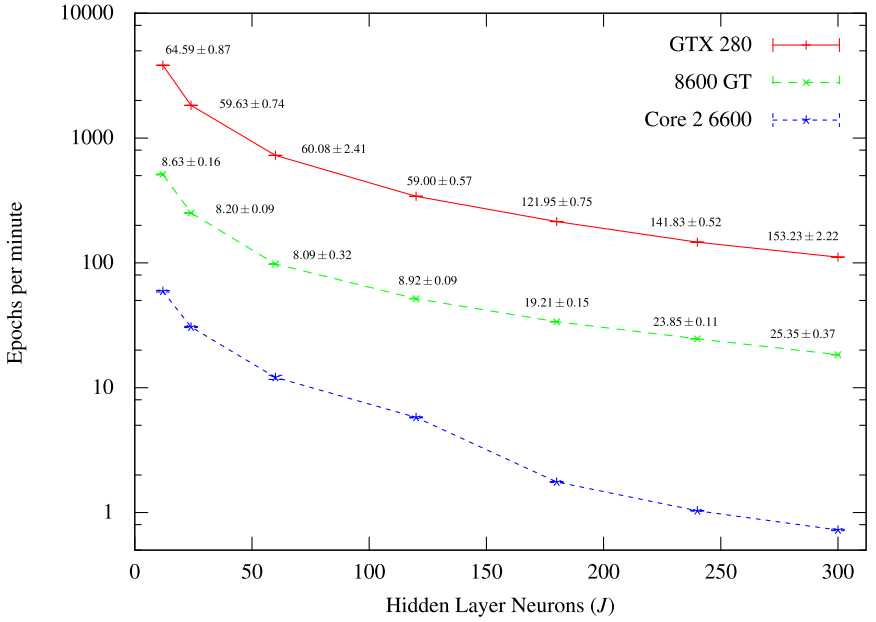


Fig. 3.13 Number of epochs per minute using the MBP algorithm for the *forest cover* problem. The GPU speedups are shown near the corresponding lines.

Since this problem has a much larger training dataset than the previous benchmarks, the speedups attained are significantly higher. This occurs because having a larger number of samples and/or connections to process results in additional operations that can be performed in parallel. Thus, the more complex the problem is, the greater is the benefit of a GPU implementation. This also explains why the speedups for the MBP are usually greater than the corresponding speedups for the BP algorithm, since the MBP networks have an additional (space) layer that can be processed in parallel.

Figures 3.14 and 3.15 present the number of epochs trained per minute for the *poker hand* problem, using respectively the BP and MBP algorithms. The settings were the same as in the *forest cover* problem. Comparatively to the former, the *poker hand* benchmark has more connections (per layer), which translate into a larger number of CUDA threads and higher speedups (up to 30× on a 8600 GT and 178× on a GTX 280 device). Moreover, training a single epoch for an MBP network with 300 hidden neurons, requires more than 5 minutes on the CPU. However, the GTX 280 performed 34 epochs per minute, whist the 8600 GT allow for 11 epochs per minute.

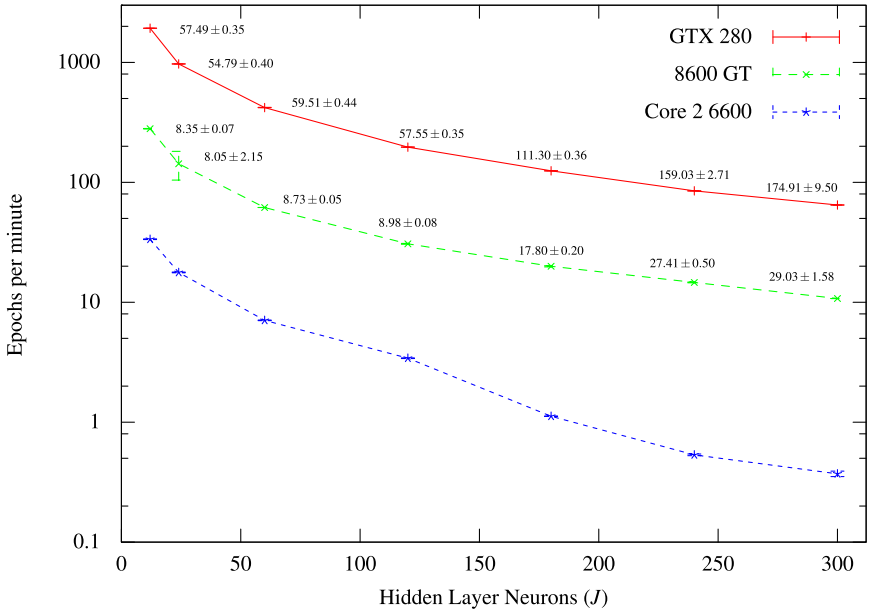


Fig. 3.14 Number of epochs per minute using the BP algorithm for the *poker* problem. The GPU speedups are shown near the corresponding lines.

3.5.3 Case Study: Ventricular Arrhythmias (VAs)

Concerning the real-world, VAs problem (see page 219), Figures 3.16 and 3.17 show the number of epochs trained per minute according to the hardware used respectively for the BP and MBP algorithms.

Figures 3.18 and 3.19 show the corresponding speedups, respectively for an 8600 GT (computer system 1) and a GTX 280 device (computer system 2). Note that the GTX 280 can account for a reduction on the training time of over 50× for the MBP algorithm.

Since currently our implementation does not support using a validation set, preliminary tests were conducted in order to determine when to stop training. Subsequently, using the information collected, we have trained several networks, during one million epochs, varying the number of hidden neurons. It is worth mentioning that during the preliminary tests a few NNs were trained up to 3 million epochs, requiring almost 9 hours of training with a GTX 280 (computer system 2). We estimate that the CPU would require almost three weeks to train a single network (of these) [123].

Table 3.4 shows the results of the best models (sensitivity, specificity and accuracy) for both algorithms. Note that considering the human costs involved, it is preferable to diagnose healthy people with ventricular arrhythmias (false positives),

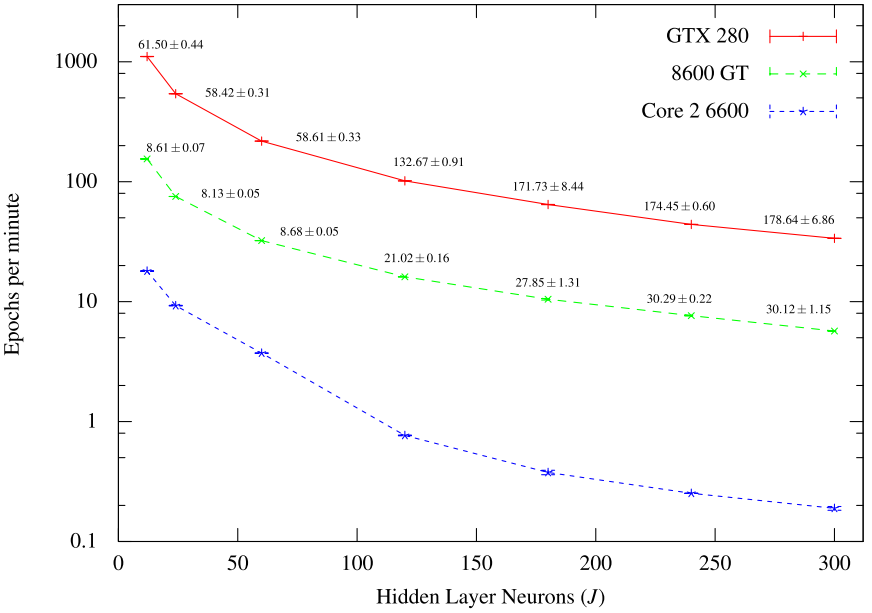


Fig. 3.15 Number of epochs per minute using the MBP algorithm for the *poker* problem. The GPU speedups are shown near the corresponding lines.

Table 3.4 Performance results (%) for the ventricular arrhythmias problem

	BP ($J = 14$)			**MBP** ($J = 13$)		
Metrics	**Train**	**Test**	**Validation**	**Train**	**Test**	**Validation**
Sensitivity	98.07	95.94	94.67	97.42	95.54	94.47
Specificity	99.84	99.62	99.61	99.87	99.68	99.70
Accuracy	99.70	99.33	99.23	99.68	99.36	99.30

rather than missing a faulty heart condition (false negatives). Therefore a classifier with high-sensitivity is preferable.

The best network trained with the BP algorithm has 14 hidden neurons ($J = 14$) and the best trained with the MBP has 13 hidden neurons with selective actuation ($J = 13$). It is important to note that the results, which improve those published in Ribeiro et al. [187] and in Marques [149], would not be obtained without the speedups provided by the GPU [123].

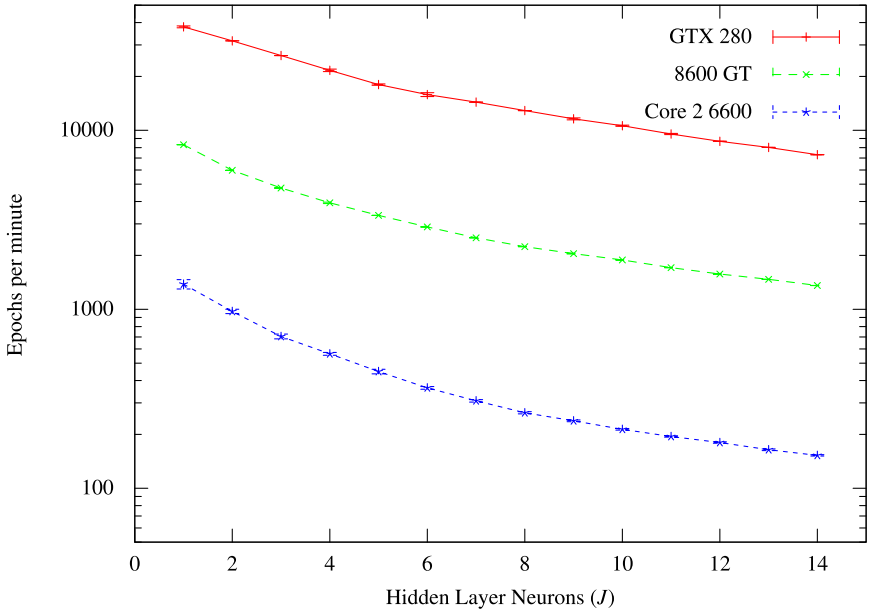


Fig. 3.16 Number of epochs per minute for the Ventricular Arrhythmias case study, using the BP algorithm

3.5.4 ATS Results

In order to validate the ATS, we intentionally choose a small problem (the *sonar* benchmark) to conduct more exhaustive tests. The system described in Algorithm 1 was tested using 1, 10 and 20 initial hidden neurons to analyze its capacity of adjusting the number of neurons. For each one of the six tests (both for BP and MBP networks), a total of 10,000 NNs were trained. In the case of MBP networks, we started with a single hidden neuron (with selective actuation), monitored the evolution of J and found out that it rapidly converges to five, corresponding to the best network found. Figure 3.20 empirically shows that almost half of the networks trained by the ATS had 5 hidden neurons and the vast majority of the networks (more than 99%) had between 4 and 6 neurons. Thus, the ATS system clearly privileges the topologies with better chances of finding high-quality solutions. Altogether, a total of 10 networks (out of 10,000) were saved by the ATS. Moreover, we checked that the best NN excels the results obtained in Lopes and Ribeiro [121]. Additionally, it is worth mentioning that the best network found in Lopes and Ribeiro [121] was also an MBP network with 5 hidden neurons.

When the initial number of the hidden neurons was changed to 10, the best NN found with $J = 6$ had the same number of false negatives (fn) and false positives (fp) as the previous one. However, when the initial number of hidden

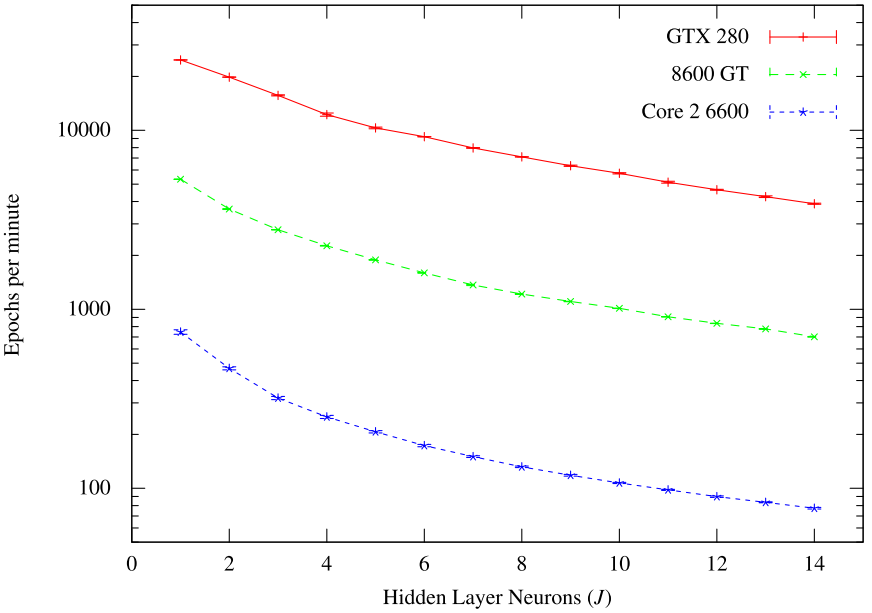


Fig. 3.17 Number of epochs per minute for the Ventricular Arrhythmias case study, using the MBP algorithm

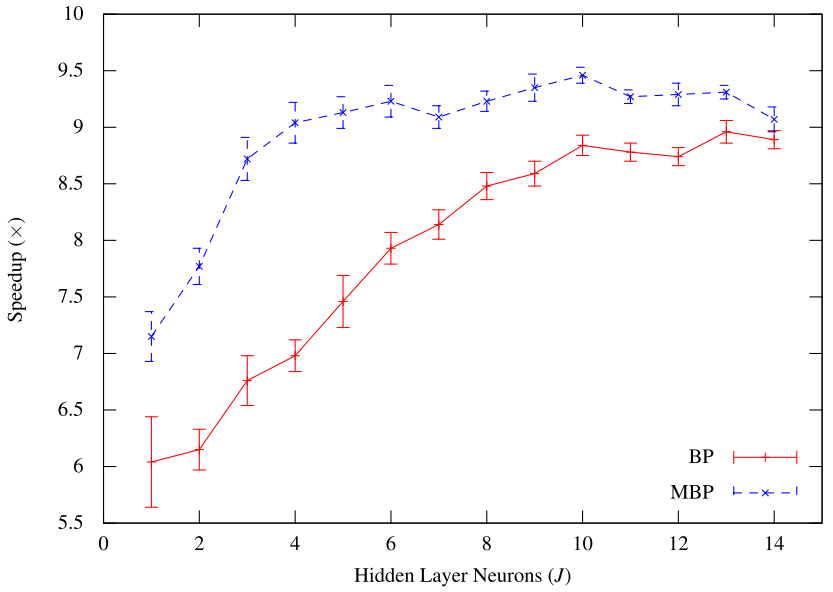


Fig. 3.18 Speedup (×) obtained for the Ventricular Arrhythmias case study, using an 8600 GT

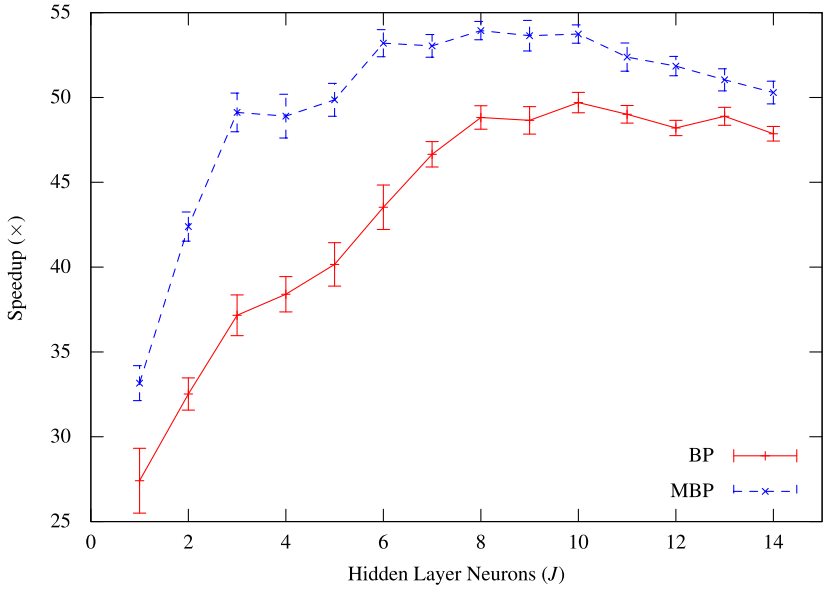


Fig. 3.19 Speedup (×) obtained for the Ventricular Arrhythmias case study, using a GTX 280

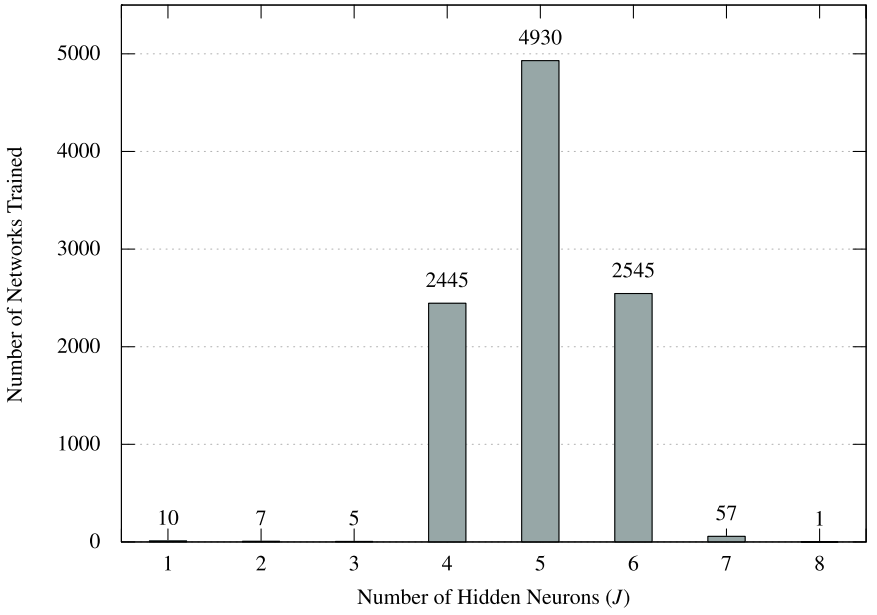


Fig. 3.20 Networks trained, according to the number of hidden neurons

neurons was set to 20 the best NN found (with 20 neurons) presented worst results. This seems to indicate that although the initial number of neurons is not a crucial parameter, setting it to low values might increase the chances of finding better solutions. The number of hidden neurons is one of the most influential parameters in an NN model. Intuitively, increasing the number of hidden units, increases the probability of avoiding local minima. This, however, is accomplished by reducing the generalization capabilities of the network, leading to the bias-variance dilemma. By varying the number of hidden neurons, two contrasting solutions, both leading to poor generalization cases, can be obtained: (*i*) a solution whereby the network model has too many free parameters that accommodate both the underlying mapping function and the noise inherent to the training data (over-fitting) and (*ii*) a solution where the local characteristics of the true mapping function are "ignored" (under-fitting), due to an insufficient number of hidden neurons. Increasing the number of free parameters (weights) leads to more flexible models with low bias and a high variance. On the other hand, a reduction on the number of free parameters results in more rigid models with high bias and low variance. Hence, we aim at finding an adequate number of parameters (i.e. hidden neurons) that correspond to the optimal balance between the bias and the variance, which will yield models that present the best predictive capabilities [18]. If the ATS starts with a small number of hidden neurons, corresponding to an under-fitting solution, it will most likely increase the number of free parameters (hidden units), such that we gradually move toward optimal models with the ideal balance between the bias and the variance. On the other hand, when starting with a large number of neurons, corresponding to an over-fitting solution, there is no guarantee that we will actually move towards optimal models, because even when the ATS reduces the number of neurons the resulting models may still present an excessive number of parameters (especially for small datasets) and therefore will not necessarily yield better solutions.

Finally, in the tests performed for the BP networks, as expected, none of the $30,000$ networks trained yielded a solution as good as the one obtained for the MBP networks.

Overall, the results obtained demonstrate that the ATS is capable of finding high-quality solutions without human-intervention. To further validate this system, Section 7.6 presents an additional experiment, in which the ATS is used on the Yale face database (see page 144). The results obtained corroborate the ones already presented here, demonstrating that the ATS is a very promising and powerful system.

3.5.5 Discussion

The previous benchmarks demonstrate that the GPU (and more specifically our implementation) can have a significant impact on the design of NN solutions, by significantly reducing the amount of time spent in the learning phase. This alleviates one of the major drawbacks of NNs, making their use more attractive

even in circumstances where they would typically be disregarded. Nevertheless, a real world problem, such as the VAs, can better exemplify how the GPU can make the difference.

We found that there is a strong correlation (88%) between the speedups and the average number of threads per layer. Figure 3.21 reports the computer system 2 (GTX 280) speedups (based on the data collected in the experiments), according to the average number of threads per layer. Clearly, more samples (N), neurons (J) and inputs (I) will result in more threads to process ($NJ(I+1)$) in a given layer, which will in turn translate into greater GPU speedups. This confirms that the GPU scales better than the CPU when handling large-datasets and complex problems.

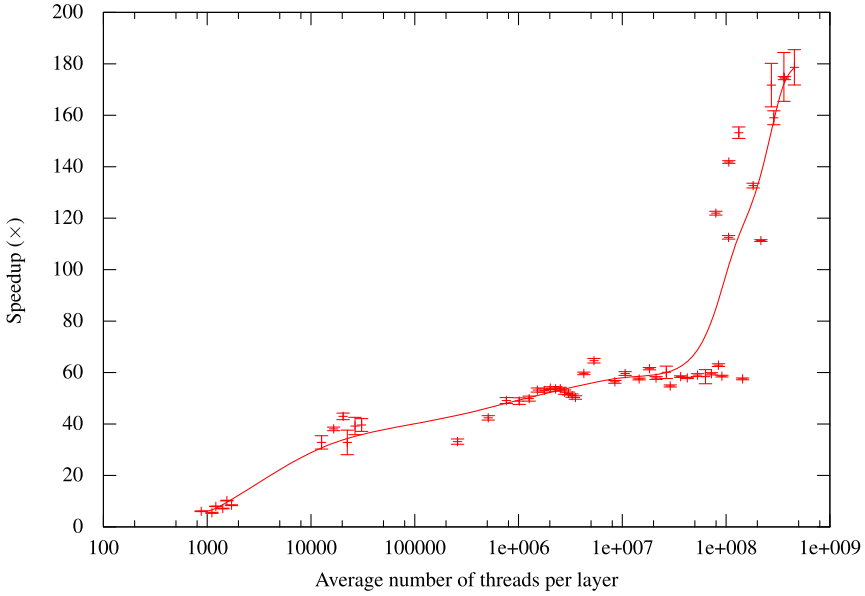


Fig. 3.21 Speedups (×) versus processing threads

3.6 Conclusion

This chapter reviews from both a theoretical and practical point of view the main issues in the BP and MBP algorithms. We have given detailed explanation of the kernels needed to their GPU implementation in CUDA. In particular, the models of flow execution are illustrated in all the phases needed to implement learning and recall with neural networks. Taking into account the previous chapters background it was possible to build a thread for unifying these quite useful ML models. Notwithstanding the gain in execution times due to the GPU platform, the chapter addresses an ATS which takes into account a constructive approach guided by an error evaluation factor. Moreover, an extensive experimentation in large datasets is described paving the way for Big Data mining in real applications.

Chapter 4
Handling Missing Data

Abstract. In this chapter we address the problematic of dealing with missing data in Neural Networks (NNs). Missing data is an ubiquitous problem with numerous and diverse causes. Therefore, handling Missing Values (MVs) properly is a crucial issue. Usually, pre-processing techniques, such as imputation, are used for estimating the missing data values. However, they limit considerably the scope of application of NNs. To circumvent this limitation we introduce a novel Neural Selective Input Model (NSIM) that accommodates different transparent and bound models, while providing support for handling MVs directly. By embedding the mechanisms to support MVs we can obtain better models that reflect the uncertainty caused by unknown values. Experiments on several datasets with both different distributions and proportion of MVs show that the NSIM approach is very robust and yields good to excellent results. Furthermore, the NSIM performs better than the state-of-the-art imputation techniques either with higher prevalence of MVs in a large number of features or with a significant proportion of MVs, while delivering competitive performance in the remaining cases.

4.1 Missing Data Mechanisms

Incomplete data is an unavoidable problem for most real-world databases, which often contain missing data [105, 102]. In particular, in domains such as gene expression microarray experiments or clinical medicine, databases routinely miss pieces of information [228, 147]. Missing Values (MVs) can exist either by design (e.g. a survey questionnaire may allow people to leave unanswered questions) or by a combination of several other factors which prevent the data from being collected and/or stored. The reasons for the prevalence of MVs include among others, sensors failure, malfunction of the equipment used to record the data, data transmission problems, different patients performing different medical diagnosis tests according to their doctor and insurance coverage, merging two or more databases with a different set of attributes [68, 24, 160]. Independently of

N. Lopes and B. Ribeiro, *Machine Learning for Adaptive Many-Core Machines – A Practical Approach,* Studies in Big Data 7,
DOI: 10.1007/978-3-319-06938-8_4,

the causes associated to the existence of MVs, the fact is that most scientific data procedures are not designed to handle them [200]. In particular in the ML area, many of the most prominent algorithms (e.g. SVMs, NNs) fail to consider MVs at all. Nevertheless, handling them in a properly manner has become a fundamental requirement for building accurate models and failure to do so usually results in models with large errors [68]. To circumvent the Missing Values Problem (MVP), ML algorithms usually rely on data preprocessing techniques such as imputation for estimating the missing data. Hence, in this case, estimated data will have the same relevance and credibility of real-data. Thus, wrong estimates of crucial variables can substantially weaken the capacity of generalization of the resulting models and originate in unpredicted and potentially dramatic outcomes [130]. Moreover, estimation methods such as imputation were conventionally developed and validated under the assumption that MVs occur in a random manner. Nevertheless, this assumption does not always hold in practice [228].

The presence of MVs in data observations is one of the most frequent problems that must be faced when building ML systems [142]. Hence, given an input matrix $\mathbf{X}$, we can build a binary response indicator matrix, $\mathbf{K} \in \{0,1\}^{N \times D}$ such that:

$$K_{ij} = \begin{cases} 1 & \text{if } X_{ij} \text{ is observed} \\ 0 & \text{if } X_{ij} \text{ is missing} \end{cases} . \tag{4.1}$$

Assuming that we sort the rows (input vectors) of $\mathbf{X}$ by their number missing of variables (features). Then we can divide $\mathbf{X}$ into the observed input matrix, $\mathbf{X_{obs}}$, containing the samples for which all the variables (features) values are known, and into the unknown input matrix, $\mathbf{X_{miss}}$ containing the samples that have variables with MVs:

$$\mathbf{X} \equiv \begin{bmatrix} \mathbf{X_{obs}} \\ \mathbf{X_{miss}} \end{bmatrix} , \tag{4.2}$$

we can define the conditional distribution for the missing data as:

$$p(\mathbf{K} \mid \mathbf{X}, \xi) = p(\mathbf{K} \mid \mathbf{X_{obs}}, \mathbf{X_{miss}}, \xi) , \tag{4.3}$$

where ξ denotes the unknown parameters which define the missing data mechanism [68, 154].

Little and Rubin [119] define three types of missing data mechanisms according to their causes: MAR, MCAR and NMAR.

4.1.1 Missing At Random (MAR)

The data is said to be MAR if the causes for the *missingness* are independent of the missing variables, but traceable or predictable from other observed variables [68]. In such cases we can define the conditional distribution for the missing data as (4.4):

$$p(\mathbf{K} \mid \mathbf{X_{obs}}, \mathbf{X_{miss}}, \xi) = p(\mathbf{K} \mid \mathbf{X_{obs}}, \xi) \,. \tag{4.4}$$

Examples of data MAR occur in the following cases: while answering questions in a survey, project managers may skip those related to small projects more often than those related to larger projects, because they may remember less details about smaller projects [154]; a sensor occasionally fails due to power outages, preventing the data acquisition process from taking place [68]. In both cases, the cause for the *missingness* is not directly tied to the variables containing the MVs but rather to other external influences. In the first case the MAR assumption can apply, because the predictor "project size" explains the likelihood of the value to be missing [154]. Similarly, the power outages in the second case explain why the sensor data is missing.

4.1.2 Missing Completely At Random (MCAR)

Data is said to be MCAR when the probability that a given variable is missing is independent of the variable itself and of any other external influences of interest, i.e. the reason for the MVs is completely random. This condition can be expressed as (4.5):

$$p(\mathbf{K} \mid \mathbf{X_{obs}}, \mathbf{X_{miss}}, \xi) = p(\mathbf{K} \mid \xi) \,. \tag{4.5}$$

Examples of this mechanism include the following [68]: a biological sample is accidentally contaminated by the researcher collecting the data; a page from a questionnaire is unintentionally lost.

4.1.3 Not Missing At Random (NMAR)

The alternative for data MAR or MCAR is to consider that the data is NMAR. This is the case when the pattern of data *missingness* depends on the missing variables themselves. A typical example of data NMAR is in the case of a personal survey involving private questions, whose nature will most likely leave them unanswered. In this scenario, unless the survey can reliably measure variables that are strongly related to those containing MVs, the MAR and MCAR assumptions are violated and we must consider that data is NMAR [154].

When data is NMAR valuable information is lost and there is no general method for handling MVs properly. Otherwise, the missing data mechanism is termed ignorable and its cause can simply be ignored, allowing the simplification of the methods for handling MVs [68].

4.2 Methods for Handling Missing Values (MVs) in Machine Learning

According to Laencina et al. [68], there are four types of methods for handling MVs: case deletion, missing data imputation, model-based procedures and ML methods for handling missing data. Our view is that the latter can be further classified into preprocessing techniques and algorithms with built-in support for MVs. Figure 4.1 presents an overview of ML procedures to handle MVs.

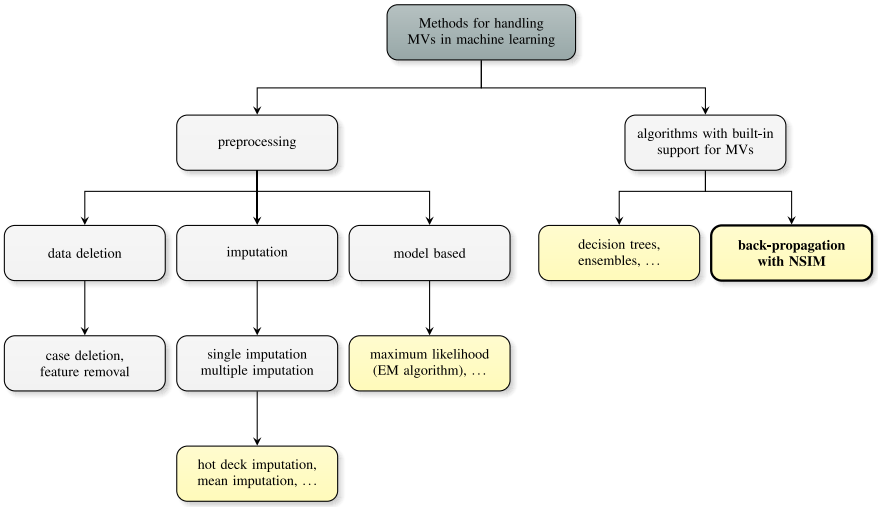


Fig. 4.1 Overview of the types of techniques for handling MVs in ML

Since many algorithms cannot directly handle MVs, a common practice is to rely on data preprocessing techniques. Usually, this is accomplished by using imputation or simply by removing instances (case deletion) and/or features containing MVs [68, 228, 142, 154, 116, 10, 105]. A review of the methods and techniques to deal with this problem, including a comparison of some well-known approaches, can be found in Laencina et al. [68].

Removing features or instances containing a large fraction of MVs is a common (and appealing) approach for dealing with the MVP, because it is a simple process and reduces the dimensionality of the data (therefore potentially reducing the complexity of the problem). This is a very conservative strategy which guarantees that errors are not introduced in the dataset [24]. However, it is not applicable when the MVs cover a large proportion of the instances, or when their presence in essential attributes is large [10, 24]. In some cases, MVs represent only a small fraction of the data but they spread throughout a large number of instances, rendering the option of case deletion inviable. Such a scenario usually happens for datasets containing a large number of features with MVs [142]. Moreover, for some problems the number

of samples available is reduced and removing instances with MVs is simply not affordable. Furthermore, discarding data may damage the reliability of the derived models [24]: if the eliminated instances are dissimilar to the remaining ones, the resulting models may not be able to properly capture the underlying data distribution and consequently will suffer from poor generalization performance [142]. Likewise, by removing features it is assumed that their information is either irrelevant or it can be compensated by other variables. However, this is not always the case and features containing MVs may have critical information which cannot be compensated for by the information embedded in the remaining features.

An alternative for deleting data containing MVs consists of estimating their values. Naturally, this process can introduce noise into the dataset and if a variable value is not meaningful for a given set of instances any attempt to substitute the MVs by an estimate is likely to lead to invalid results [24]. Many algorithms have been developed for this purpose (e.g. mean imputation, regression imputation, hot deck imputation, weighted k-nearest neighbor approach, Bayesian principle component analysis, local least squares) [68, 228, 116, 119]. In the NN domain, an example of such an approach consists of using an Hopfield network, whose neurons are considered both inputs and outputs, as a auto-associative memory [203, 3]. Basically, when the Hopfield network receives a noisy or incomplete pattern, it will iterate to a stable state that best matches the unknown input pattern [3, 239]. Independently of the method chosen, wrong estimates of crucial variables can substantially weaken the capacity of generalization of the resulting models. Moreover, models based on imputed (estimated) data consider MVs as if they are the real ones (albeit their value is not known) and therefore, the resulting conclusions do not show the uncertainty produced by the absence of such values [142]. Furthermore, statistically, the variability or correlation estimations can be strongly biased [142].

Multiple imputation techniques (e.g. metric matching, bayesian bootstrap) take into account the variability produced by the absence of MVs, by replacing each MV with two or more acceptable values, representing a distribution of possibilities [142]. However, although the variability is taken into account, MVs will still be treated as if they are real. Furthermore, estimation methods were conventionally developed and validated under the assumption that data is MAR. Notwithstanding, this assumption does not always hold in practice. In the particular case of microarray experiments the distribution of missing data is highly non-random due to technical and/or experimental conditions [228].

Preprocessing methods have the advantage of allowing the same data to be used by different algorithms. Nevertheless, the burden of preprocessing data, which already accounts for a significant part of the time that is spent in order to build an ML system, is further increased. Moreover, additional knowledge is required to provide a better foundation for making decisions on choosing strategic options, namely, methods and tools available to handle MVs [137].

Finally, these methods result in the loss of information when the *missingness* is by itself informative. This is the case when the MVs distribution provides valuable information for the classification task, that is lost when the MVs are replaced

by their respective estimates [68]. For example, in the real-world bankruptcy problem, presented later in Section 4.5, distressed companies tend to omit much more financial information than healthy ones [130]. A simple explanation for this behavior is that in general companies in the process of insolvency try to conceal their true financial situation from its stakeholders (suppliers, customers, employees, creditors, investors, etc.). Thus, in this particular case, the specialized knowledge that a particular set of data variables is missing can play an important role in the construction of better models [130, 137].

Algorithms with built-in support for handling MVs offer numerous advantages over (more) conventional ones:

1. The amount of time spent in the preprocessing phase is reduced;
2. Their performance is consistent and does not depend on the knowledge of proper methods and tools for handling MVs (e.g. some practitioners may rely on ad-hoc solutions and obtain less reliable models than those built with better skilled techniques);
3. Noise and outliers are not inadvertently injected in the data and the uncertainty associated to the missing data is preserved;
4. It can be decided whether the missing information is informative (or not), resulting in better models.

Despite these advantages, most algorithms are incapable of handling MVs, mostly because in many cases that would complicate their methods to the point that they could become impractical and in many situations there would not be any real gains. Fortunately, this is not always the case and, in the next Section, we present an elegant and simple solution that empowers NNs with the ability of dealing directly with the ubiquitous MVP [137].

4.3 NSIM Proposed Approach

The building blocks of the proposed NSIM are the selective actuation neurons, described earlier in Section 3.2 (see page 47).

Let us consider that each row of the response matrix, $\mathbf{K}$, contains the response indicator vector of a specific sample, $\boldsymbol{\kappa}_\mathbf{i} \in \{0,1\}^D$. Furthermore, to simplify the notation, let $\boldsymbol{\kappa} = [\kappa_1, \kappa_2, \ldots, \kappa_D]$ denote the response indicator vector of a generic sample where κ_i is a random variable with Bernoulli distribution representing the act of obtaining the value of x_i ($\kappa_i \sim Be(p_i)$). In order to deal with the missing data values, we propose transforming the values of x_i by taking into consideration κ_i as shown in (4.6):

$$\tilde{x}_i = f(x_i, \kappa_i) \,. \tag{4.6}$$

This transformation can be carried out by a selective actuation neuron, j, designated by selective input, which receives a single input, x_i, and has an importance factor m_j identical to κ_i. Consequently, (4.6) can be replaced with (4.7), by taking advantage of (3.15):

$$\tilde{x}_i = \kappa_i \phi(W_{ij} x_i + b_j) \; . \tag{4.7}$$

If the value x_i can not be obtained then the selective input associated to it will behave as if it did not exist, since κ_i will be zero. On the other hand, if the value of x_i is available ($\kappa_i = 1$), the selective input will actively participate on the computation of the network outputs. This can be viewed as if there are two different models, bound to each other, sharing information. One model for the case where the value of x_i is known and another one for the case where it can not be obtained (is missing). Figure 4.2 shows the physical model (NSIM) of a network containing a selective input and the two conceptual models inherent to it. A network with I selective inputs will have 2^I different models bonded to each other and constrained in order to share information (network weights) [137].

As we said before, an NN acts as an adaptive non-linear mapping function, $f : \mathbb{R}^D \longrightarrow \mathbb{R}^C$. Consequently, the goal of the training procedure consists of adjusting the free parameters (weights) of the network $(W_1, W_2, \ldots, W_P)$, so that the resulting model fits adequately the observed (training) data. Hence, we can view the aforementioned strategy as if we decompose f into sub-functions that share information (parameters) among each other. For the simpler case of a network having a single selective input, f could be decomposed into two different functions, $f_1(W_1, W_2, \ldots, W_s)$ and $f_2(W_1, W_2, \ldots, W_s, W_{s+1}, \ldots, W_P)$, that share s parameters. For the network presented in Figure 4.2, s corresponds to the number of weights of model 1 and P to the number of weights of model 2. It is guaranteed that all the models share at least s parameters, corresponding to the number of weights that the network would have if the inputs with MVs were not considered at all [127].

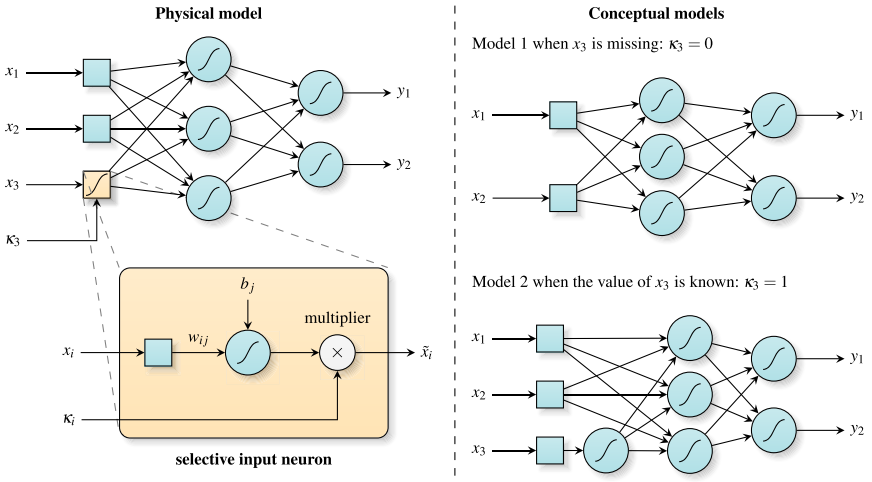


Fig. 4.2 Physical and conceptual models of a network with a selective input ($j = 3$)

Although conceptually there are multiple models, from the point of view of the training procedure there is a single model (NSIM), f with P adjustable parameters (weights). When a pattern is presented to the network, only the parameters directly or indirectly related to the inputs with known values are adjusted (observe equations (3.8), (3.16) and (3.17)). Thus, only the relevant (conceptual) models will be adjusted [127].

The NSIM presents a high degree of robustness, since it is prepared to deal with faulty sensors. If the system which integrates the NSIM realizes a given sensor has stopped working, it can easily deactivate (discard) all the models inherent to that specific sensor, by setting $\kappa_i = 0$. Thus, consequently the best model available for the remaining sensors working properly will be considered. In addition, the NSIM does not require MAR or MCAR assumptions to hold, since only the known data is used actively to build the model [137].

4.4 GPU Parallel Implementation

The GPU implementation of the NSIM extends the BP and MBP CUDA parallel implementation previously presented in Section 3.3 (see page 52). Altogether, a total of three new kernels were added to the referred implementation.

In order to calculate the outputs of the selective input neurons, $\tilde{x}_i$, a kernel, named `FireSelectiveInputs`, was created. This kernel, whose code is show in Listing 4.1, assumes that standard inputs may coexist with selective inputs. Thus, it should be launched with one thread per input and sample (regardless of the type of the inputs – selective or standard). Moreover, since our implementation considers the batch training mode, the $\tilde{x}_i$ variables will be calculated simultaneously for all the training patterns (samples) and the threads should be grouped in blocks containing all the inputs of a given sample. Of course, for standard inputs the value of $\tilde{x}_i$ must match to the original input ($\tilde{x}_i = x_i$). Therefore, to differentiate standard inputs from the selective ones, the value of the weights and bias of the standard inputs is set to zero – the kernel checks this condition to determine which type of input is being handled. Therefore, thread divergence is avoided when all the inputs are selective inputs. Thus, the maximum performance of this kernel is obtained when we treat all the inputs as selective inputs.

For the back-propagation phase two additional kernels were created: `CalcLocalGradSelectiveInputs` and `CorrectWeightsSelectiveInputs`. The first one calculates the local gradients of the selective input neurons for all the samples and the latter is responsible for adjusting the weights of the selective input neurons. As in the case of the `FireSelectiveInputs` kernel, maximum performance is achieved when all the inputs are considered to be selective inputs. A complete and functional implementation of this method was integrated also in the Multiple Back-Propagation software (previously mentioned in page 58).

Listing 4.1 FireSelectiveInputs kernel.

```
#define NEURON threadIdx.x
#define NUM_NEURONS blockDim.x
#define PATTERN blockIdx.x

__global__ void FireSelectiveInputs(float * inputs, float *
    weights, float * bias, float * outputs) {
    int idx = PATTERN * NUM_NEURONS + NEURON;

    float o = inputs[idx];

    if (isnan(o) || isinf(o)) { // missing value
        o = 0.0;
    } else {
        float w = weights[NEURON];
        float b = bias[NEURON];

        if (w != 0.0 || b != 0.0) o = tanh(o * w + b);
    }

    outputs[idx] = o;
}
```

4.5 Results and Discussion

4.5.1 Experimental Setup

To evaluate the performance of the NSIM, we carried out extensive experiments on several datasets with different distributions and proportion of MVs. Table 4.1 presents the main characteristics of the databases and an overview of the proportion and distribution of the MVs in each database, after preprocessing. Additional details on the datasets can be found in Section A.4.

In the experiments we use 5-fold stratified cross-validation partitions. For statistical significance 30 different NNs were trained in each partition and algorithm.

The results of the NSIM were compared with single imputation and multiple imputation methods. Multiple imputation is considered one of the most powerful approaches for estimating MVs [10]. For single imputation the version 3.7.5 of the Weka software package was used [80]. For multiple imputation, the NORM (Multiple imputation of incomplete multivariate data under a normal model) software was used [199]. NORM uses the Expectation-Maximization algorithm either with the maximum-likelihood or the posterior mode estimates. Since the maximum-likelihood estimate fails for many of the databases in Table 4.1, the posterior mode was used. In this context, the EM algorithm is used for fitting models to incomplete data, by capitalizing on the relationship between the unknown parameters associated to the data model and the missing data itself [160].

Table 4.1 Main characteristics, proportion and distribution of the MVs for the UCI database benchmark experiments (after data preprocessing). Note that the average (avg.) and the standard deviation (stdev.) of MVs per feature does not include features without MVs.

				Proportion	Features	MVs per feature	
Database	*N*	*D*	*C*	**of MVs (%)**	**with MVs**	**(avg. %)**	**(stdev. %)**
Annealing	898	47	5	49.22	37 (78.72%)	62.52	37.04
Audiology	226	93	24	2.85	23 (24.73%)	11.54	22.43
Breast cancer	699	9	2	0.25	1 (11.11%)	2.29	0.00
Congressional	435	16	2	5.63	16 (100.00%)	5.63	5.41
hepatitis	155	19	2	5.67	15 (78.95%)	7.18	11.05
Horse colic	368	92	2	15.26	59 (64.13%)	23.79	16.01
Japanese credit	690	42	2	0.97	32 (76.19%)	1.27	0.24
Mammographic	961	5	2	3.37	5 (100.00%)	3.37	3.22
Mushroom	8124	110	2	1.11	4 (3.64%)	30.53	0.00
Soybean	683	77	19	8.73	76 (98.70%)	8.85	6.03

4.5.2 Benchmark Results

Table 4.2 presents the macro-average F-Measure performance (%) according to the algorithms used to train the NNs and the methods chosen for handling the MVs.

Table 4.2 Macro-average F-Measure performance (%) according to the methods used to handle the MVs and the algorithms used to train the NNs

	NSIM		Single Imputation		Multiple Imputation	
Database	**BP**	**MBP**	**BP**	**MBP**	**BP**	**MBP**
Annealing	97.13 ± 02.69	**97.77 ± 02.55**	91.93 ± 06.79	93.22 ± 06.65	93.04 ± 07.40	93.16 ± 06.53
Audiology	56.46 ± 14.16	58.64 ± 13.08	53.73 ± 14.56	55.33 ± 13.25	56.76 ± 14.16	**61.07 ± 14.72**
Breast cancer	95.05 ± 01.59	95.42 ± 01.69	95.38 ± 01.21	**95.53 ± 01.67**	95.01 ± 01.53	95.37 ± 01.66
Congressional	93.20 ± 02.07	94.30 ± 01.58	93.84 ± 01.86	94.12 ± 02.24	**94.83 ± 01.79**	94.70 ± 01.46
hepatitis	70.10 ± 06.23	73.55 ± 05.99	72.63 ± 07.96	72.20 ± 07.53	**75.89 ± 10.35**	75.44 ± 09.98
Horse colic	84.31 ± 02.56	84.86 ± 02.44	83.29 ± 02.80	83.11 ± 02.78	87.30 ± 02.06	**87.37 ± 02.30**
Japanese credit	**81.98 ± 02.45**	81.50 ± 02.81	81.43 ± 02.53	81.07 ± 02.25	81.27 ± 01.93	81.59 ± 02.45
Mammographic	**81.62 ± 01.74**	81.07 ± 01.97	79.78 ± 02.28	78.23 ± 02.56	79.93 ± 02.00	79.45 ± 02.36
Mushroom	99.97 ± 00.02	99.96 ± 00.02	**99.98 ± 00.02**	**99.98 ± 00.01**	99.97 ± 00.02	**99.98 ± 00.01**
Soybean	93.43 ± 00.68	**94.34 ± 00.94**	91.63 ± 01.26	92.87 ± 00.85	92.54 ± 00.68	93.48 ± 00.66

As expected, the MBP algorithm performs better than BP regardless of the method used to handle MVs. On average the F-Measure performance of MBP excels the one of BP by 0.20%, 0.50% and 0.82% respectively for single imputation, multiple imputation and NSIM methods. Using the Wilcoxon signed rank test, for the case of the NSIM, the null hypothesis of BP having an equal or better F-Measure than the MBP algorithm is rejected at a significance level of 0.05.

Comparing our method with the use of single imputation, we can verify that our method outperforms single imputation both for the BP and MBP algorithms, respectively by 0.96% and 1.58% on average. This considerable gain in terms of F-Measure performance, especially in the case of the MBP algorithm, is validated by Wilcoxon signed rank test: the null hypothesis of the MBP models created using single imputation having an equal or better F-Measure than those with the NSIM is rejected at a significance level of 0.01.

In contrast with single imputation, multiple imputation yields better results than the NSIM, concerning the BP algorithm (+0.33% on average). However, if we make use of the statistical evidence and adopt the MBP networks, then both approaches will perform similarly (on average multiple imputation performs better than NSIM by less than 0.02%).

Note however that if we consider only the datasets for which the proportion of MVs represents at least 5% of the whole data (*annealing*, *congressional*, *hepatitis*, *horse colic* and *soybean*), then concerning the MBP algorithm, our method excels the multiple imputation on average by 0.14%. These results seem intuitive since in principle multiple imputation works better when the proportion of MVs is smaller, in which case more data is available for validating the estimates inferred. Moreover, they assume particular relevance, if we consider that the appropriate choice of the method for handling MVs is especially important when the fraction of MVs is large [10].

Another important consideration is the impact that the proportion of features with MVs has in each method. If we look at the datasets containing a high-proportion of features with MVs, then the F-Measure performance of the NSIM is once again superior to the corresponding performance of multiple imputation. Considering only the datasets for which at least $3/4$ of the features contain MVs (*annealing*, *congressional*, *hepatitis*, *Japanese credit*, *mammographic* and *soybean*) we can verify that, for the MBP algorithm, on average the NSIM improves the F-Measure performance by 0.79% relative to the multiple imputation method. This shows that our model tends to perform better than multiple imputation when the MVs spread throughout a large proportion of the features.

Additionally, the NSIM presents the best solution in terms of system integration, in particular for hardware realization. Multiple imputation requires the system to include not only the multiple imputation algorithm itself, but also all the data needed for computing the adequate estimates. While tools such as the MBP software can generate code for any NN, to our knowledge there are no such tools or open source code (in general purpose languages) which would allow to easily embed multiple imputation techniques in the NN systems. Moreover, the time and memory constraints necessary for the imputation process to take place would in many cases render such systems useless.

4.5.3 Case Study: Financial Distress Prediction

This study involves the bankruptcy problem described in more detail in Appendix A.5 (see page 217). The original French companies database, contained on average over 4% of MVs for each financial ratio in each year. However, some ratios had almost a third of the data missing. What is more interesting is that if we consider only the data from distressed companies, then the average of MVs for the financial ratios increases to 42.35%. In fact, it is observed that there are many features that contain less than one quarter of the data. We are unsure why this happens, but one possible explanation is that the affected firms could be trying to hide information from the markets. Nevertheless, this highlights the fact that knowing that some information is missing could be as important as having access to the information itself. Thus, in this respect our model is advantageous, since it preserves the missing information (unlike imputation methods). As expected, when looking at the data of each company (sample) we found similar results: overall, on average only 3 or 4 ratios are missing; however when considering only the distressed firms, roughly 37 ratios per sample are missing. Moreover, there are companies for which all the ratios are unknown.

To create a workable and balanced dataset, we started by selecting all the instances of the database associated to the distressed companies, whose number of unknown ratios did not exceed 70 (we considered that at least approximately 20% of the 87 ratios should contain information). Accordingly, a total of $1,524$ samples associated to distressed companies were chosen. Subsequently, we selected the same number of samples associated to healthy companies, in order to obtain a balanced dataset. The samples were chosen so that the MVs were uniformly distributed by all the ratios. The resulting dataset contains $3,048$ instances – a number over 5 times greater than the number of samples that could be obtained in previous work [235, 189], using imputation methods. The resulting dataset contains on average 27.66% of missing values per ratio. Moreover, on average 24 ratios per sample are missing.

Table 4.3 presents the results of the NSIM, with the MBP algorithm, using a 10-fold cross-validation. These excel by far the results previously obtained [235, 189] when imputation techniques were used and demonstrated the validity and usefulness of the NSIM in a real-world setting.

Table 4.3 Results of the NSIM for the bankruptcy problem

Metric	Results (%)
Accuracy	95.70 ± 1.42
Sensitivity	95.60 ± 1.61
Specificity	95.80 ± 1.83
Precision	95.82 ± 1.77
Recall	95.60 ± 1.61
F-measure	95.70 ± 1.35

For each fold, the ATS was used to train 100 MBP networks containing a single hidden layer. On average the NNs had 43.75 selective actuation neurons. Figure 4.3 presents an estimate of the speedups, for the bankruptcy problem, using the computer system 2 (GTX 280). Once again, the GPU implementation proved to be crucial for obtaining quality NN models.

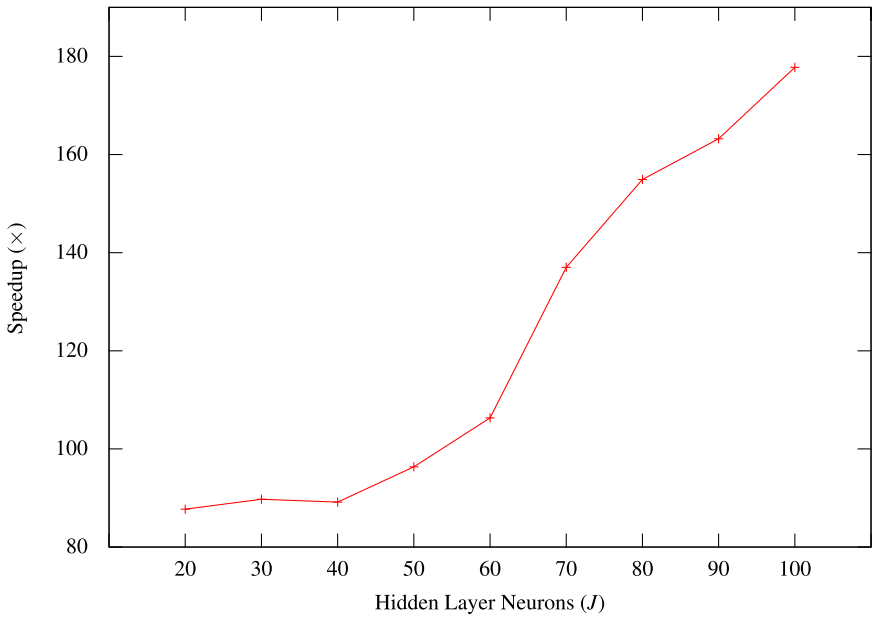


Fig. 4.3 GPU speedups obtained for the bankruptcy problem

One of the strengths of the NSIM relies on the possibility of using data with a large number of missing values. This is important because better (and more accurate) models can be built by incorporating and taking advantage of extra information. Moreover, instead of injecting inaccurate values into the system, as imputation methods do, the NSIM preserves the uncertainty caused by unknown values increasing the model utility, even when relevant information is missing.

4.6 Conclusion

This chapter has introduced a novel NSIM model to handle missing values in data sets. This is a realistic problematic since very often the data from sensors, internet transactions or from instruments simulators is incomplete or even missing. First the missing data mechanisms were introduced and then the literature existing techniques were more detailed. A broad coverage from randomness to imputation or even

probabilistic approaches is given. In the perspective of a crucial problem in ML we developed and presented a new approach where a physical model and conceptual models are interlaced together to address this problematic. The kernel for the GPU parallel implementation of the NSIM model is presented in the final part of the chapter as well as results and experiments in several benchmark datasets and in a real case of French companies financial distress prediction proving to be robust. A salient feature of the NSIM is that it presents the best solution in terms of system integration, in particular for hardware realization.

Chapter 5
Support Vector Machines (SVMs)

Abstract. This Chapter details a class of learning mechanisms known as the Support Vector Machines (SVMs). We start by giving the machine learning framework, define and introduce the concepts of linear classifiers, and describe formally the SVMs as large margin classifiers. We focus on the convex optimization problem and in particular we deal with the Sequential Minimal Optimization (SMO) which is crucial to proceed to implement the algorithm. Finally we detail issues of the SVMs implementation. Regarding the latter, several aspects related to CPU and GPU implementation are surveyed. Our aim is two fold: first, we implement the multi-thread CPU version, test it in benchmark data sets; then we proceed with the GPU version. We intend to give a clear understanding of specific aspects related to the implementation of basic SVM machines in a many-core perspective. Further developments can easily be extended to other SVM variants launching one step further the potential for big data adaptive machines.

5.1 Introduction

Support Vector Machines (SVMs) have become indispensable tools in the ML domain due to their generalization properties and regularization capability [231, 230]. SVMs are binary large margin classifiers, which have found successful applications in many scientific fields. These state-of-the-art machine learning methods have their roots in the principle of structural risk minimization [231, 206]. The rationale is that a function from a given class of functions tends to have a low expectation risk on data if it has a low empirical risk on the data sampled from the same distribution and at the same time the function class has a low Vapnik-Chervonenkis (VC)-dimension [231, 206]. The large margin SVM classifier embeds this idea by restricting the complexity of the function class.

The theory behind the SVM has its grounds on linear binary classifiers. These are, on their simplest form, used as a solution to problems with two linearly separable classes. Given a training set $\{\mathbf{x_i}, t_i\}_{i=1}^{N}$ of N training points in a D dimensional

N. Lopes and B. Ribeiro, *Machine Learning for Adaptive Many-Core Machines – A Practical Approach,* Studies in Big Data 7,
DOI: 10.1007/978-3-319-06938-8_5,

feature space where $\mathbf{x_i} \in \mathbb{R}^D$ and $t_i \in \{-1, 1\}$, the aim of SVMs is to induce a classifier which has good classification performance on unseen data points.

Essentially, their solution involves optimization techniques which are overwhelming important due to the requirements on the training speed, memory constraints and accuracy of optimization variables. However, the optimization tasks are numerically expensive and single-threaded implementations are hardly able to cope with the complex learning task. Thus SVMs can have high computational cost, even in the test phase [29]. Subsequently, multi-threaded implementations are crucial to circumvent this problem.

There are several optimization techniques and since SVMs have multiple alternatives [256], this issue still raises interest. Specifically if Big Data is addressed, the learning algorithms often require high-processing capabilities making current single-threaded implementations unable to scale with the demanding processing power needed. In this Chapter we will focus on the Sequential Minimal Optimization (SMO) widely used to train SVMs. We will present a multi-threaded implementation of the SMO, which reduces the numerical complexity by parallelizing the Karush-Kuhn-Tucker (KKT) conditions update, the calculation of the hyperplane offset and the classification task. Understanding the main steps used so far will be useful for other related optimization techniques that can be employed in other SVM variants.

In short, despite their advantages, SVMs usually require significant memory and computational burden for calculating the large Gram matrix [45]. Although, several methods have been proposed to circumvent this limitation [23, 90], there is still room for improving existing implementations by taking advantage of the abundance of distributed platforms choices. These options include customizable integrated circuits (e.g. Field-Programmable Gate Arrays – FPGAs), custom processing units (e.g. general-purpose Graphics Processing Units – GPUs), and many-core CPU machines, among others. Today, the multi-core architecture CPUs and GPUs, are present in modern baseline computers and will be addressed in this Chapter.

In the next sections we first give a brief introduction to SVMs, next we focus on a multi-threaded parallel CPU standalone SVM version (MT-SVM) which builds up from the scratch an implementation of the Sequential Minimal Optimization (SMO) algorithm. In the sequel a GPU implementation is presented under the CUDA environment framework. Finally, we present results in benchmark datasets for both implementations.

5.2 Support Vector Machines (SVMs)

SVMs were introduced by Vapnik [231, 51] based on the Structural Risk Minimization (SRM) principle, as an alternative to the traditional Empirical Risk Minimization (ERM) principle [207]. The rationale consists of finding an hypothesis h from an hypothesis space H, for which the lowest probability of error $Err(h)$ is guaranteed. Accordingly, given a training set containing N samples:

$$(\mathbf{x_1}, t_1), \ldots, (\mathbf{x_N}, t_N), \mathbf{x_i} \in \mathbb{R}^D, t_i \in \{-1, +1\}, \tag{5.1}$$

the SRM establishes a relationship between (*i*) the true error of the hypothesis h, $Err(h)$, and (*ii*) the error of the training set, Err_{train}, and the complexity of the hypothesis (5.2):

$$Err(h) \leq Err_{train}(h) + O\left(\frac{v\log(\frac{\eta}{v}) - \log(\eta)}{n}\right). \tag{5.2}$$

where v denotes the VC-dimension [230], which is a property of the hypothesis space H and indicates its expressiveness.

The upper bound of the error, expressed in the right term of (5.2), holds with a probability of at least $1-\eta$ and reflects the trade-off between the complexity of the hypothesis space and the training error [212]. A simple hypothesis space will result in under-approximating models, which present high training and true errors. On the other hand, a too-rich hypothesis space, i.e. with a large VC-dimension, will result in over-fitting models. This problem arises because, although a small training error will occur, the second term in the right-hand side of (5.2) will be large. Hence, it is crucial to determine the hypothesis space with sufficient complexity. In the SRM this is achieved by defining an enclosed structure of hypothesis spaces H_i, so that their respective VC-dimension v_i increases:

$$H_1 \subset H_2 \subset H_3 \subset \cdots \subset H_i \subset \ldots \quad \textit{and} \quad \forall i : v_i \leq v_{i+1} \tag{5.3}$$

This structure is *a priori* defined to find the index for which (5.2) is minimum. To build this structure the number of features is restricted. SVMs learn linear threshold functions of the type:

$$h(\mathbf{x}) = \text{sign}(\mathbf{w} \cdot \mathbf{x} + b) = \begin{cases} +1 & \text{if } \mathbf{w} \cdot \mathbf{x} + b > 0 \\ -1 & \text{otherwise,} \end{cases} \tag{5.4}$$

where $\mathbf{w} = [w_1, w_2, \ldots, w_N]$ are linear parameters (weights) of the model and b is the bias. Linear threshold functions with D features have a VC-dimension of $D+1$ [230].

The SVM aims to find the optimal decision hyperplane, i.e., we aim to find the hyperplane that gives the smallest generalization error, while minimizing the empirical error. This trade-off embodies the principle of SRM [231, 207, 229]. The rationale consists of finding the adequate balance to the trade-off between the capabilities of the model and the classification performance on the observed (training) data (bias-variance dilemma) [29]. In Figure 5.1 this trade-off is depicted where generalization error and empirical risk are represented in terms of the machine complexity given by the VC dimension. Within this principle the optimal hyperplane is the one orthogonal to the largest margin between data points from both classes. Therefore, the SVMs belong to a class of algorithms which are known as Maximum-Margin Classifiers [230, 212]. Notice that in Statistical Learning

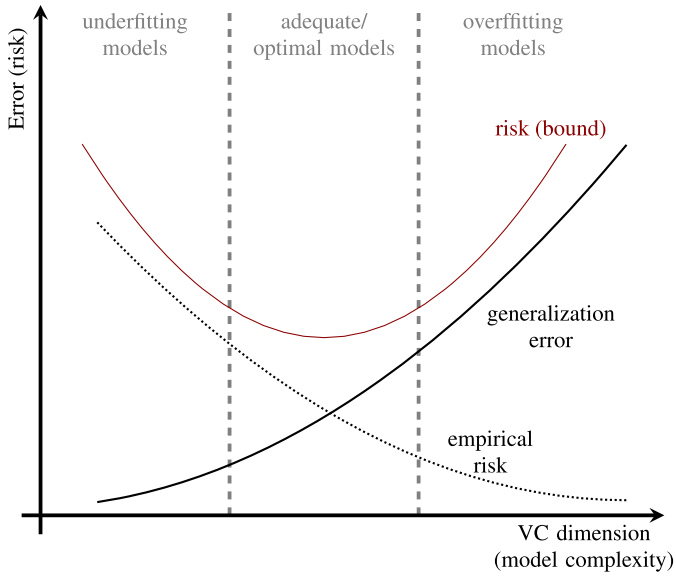


Fig. 5.1 Structural risk minimization balances generalization error and empirical risk (training error), in terms of the model complexity

Theory [231, 51] the machine capacity can be high if we have low VC-dimension, i.e., low complexity, even if the data dimensionality is high.

5.2.1 Linear Hard-Margin SVMs

In the simplest SVM formulation, a training set can be separated by at least one hyperplane, i.e. data are linearly separable and a linear model can be used:

$$y(\mathbf{x}) = \mathbf{w}^\top \mathbf{x} + b. \tag{5.5}$$

In this case, SVMs are called linear hard-margin and there is a weight vector $\mathbf{w}'$ and a threshold b' that correctly define the model. Basically there is a function of the form (5.5) that satisfies $y(\mathbf{x_i}) > 0$ for data points with $t_i = 1$ and $y(\mathbf{x_i}) < 0$ for data points with $t_i = -1$, so that for each training data sample $(\mathbf{x_i}, t_i)$:

$$t_i y(\mathbf{x_i}) = t_i(\mathbf{w}'\mathbf{x_i} + b') > 0. \tag{5.6}$$

As a rule, multiple hyperplanes exist that allow such separation without error (see Figure 5.2), and the SVM determines the one with largest margin ρ, i.e. furthest from the hyperplane to the closest training examples. The examples closer to the hyperplane are called Support Vectors (SVs) and have an exact distance of ρ to the

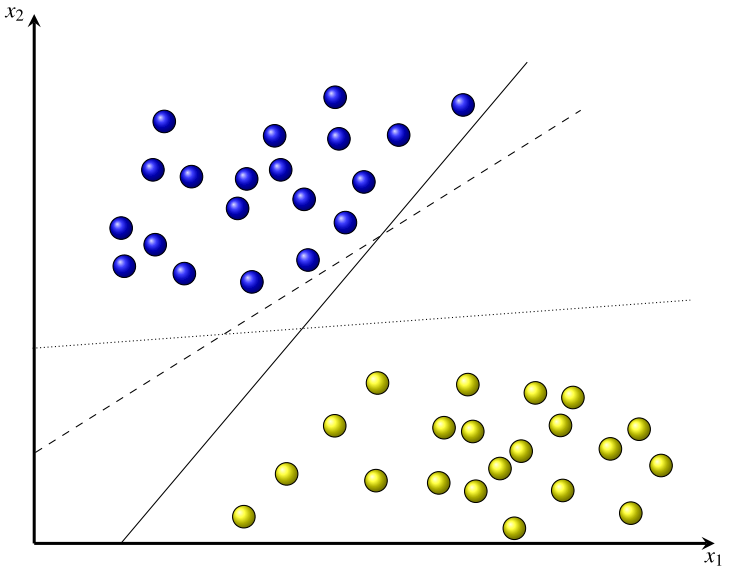


Fig. 5.2 Two possible hyperplanes that discriminate both classes

hyperplane. Figure 5.3 depicts one optimal separating hyperplane associated with "Direction 1" and seven SVs.

The distance of a data point to a hyperplane defined by $y(\mathbf{x}) = 0$, where $y(\mathbf{x})$ takes the form of (5.5), is given by $\frac{|y(\mathbf{x})|}{||\mathbf{w}||}$ [18]. As for now, the interest is only in solutions for which all data samples are correctly classified, so that $t_i y(\mathbf{x_i}) > 0$, the distance of a data point $\mathbf{x_i}$ to the decision surface is given by (5.7):

$$\frac{t_i y(\mathbf{x_i})}{||\mathbf{w}||} = \frac{t_i(\mathbf{w}^\top \mathbf{x_i} + b)}{||\mathbf{w}||}. \tag{5.7}$$

The margin is then given by the perpendicular distance to the closest point $\mathbf{x_i}$ of the data set, and we wish to optimize the parameters $\mathbf{w}$ and b in order to maximize this distance. Thus, the maximum margin solution is found by solving (5.8):

$$\arg\max_{\mathbf{w},b}\left\{\frac{1}{||\mathbf{w}||}\min_i[t_i(\mathbf{w}^\top \mathbf{x_i} + b)]\right\}. \tag{5.8}$$

Using a canonical representation of the decision hyperplane, all data points satisfy (5.9):

$$t_i(\mathbf{w}^\top \mathbf{x_i} + b) \geq 1 \qquad i = 1, \ldots, N \tag{5.9}$$

When the above equality holds, the constraints are said to be active for that data point, while for the rest the constraints are inactive. There will always be at least one active constraint, since there will always be a closest point, and once the margin has

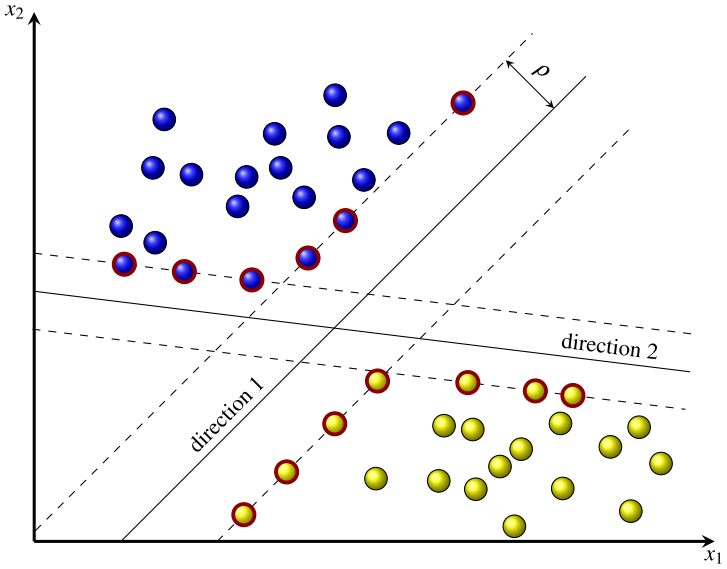


Fig. 5.3 Two possible separating hyperplanes. The optimal hyperplane is the one associated to "direction 1", which presents the largest margin ρ.

been maximized, there will be at least two active constraints. Then, the optimization problem consists of maximizing $||\mathbf{w}||^{-1}$, which is equivalent to minimizing $||\mathbf{w}||^2$, and so we have to solve the primal optimization problem:

$$\begin{aligned} &\text{minimize: } \frac{1}{2}\mathbf{w}\cdot\mathbf{w}, \\ &\text{subject to: } \forall_{i=1}^{N} : t_i(\mathbf{w}\cdot\mathbf{x_i}+b) \geq 1 \end{aligned} \tag{5.10}$$

These constraints establish that all training examples should lie on the correct side of the hyperplane. Using the value of 1 on the right-hand side of the inequalities enforces a certain distance from the hyperplane:

$$\rho = \frac{1}{||\mathbf{w}||}, \tag{5.11}$$

where $||\mathbf{w}||$ denotes the L_2-norm of $\mathbf{w}$. Therefore minimizing $\mathbf{w}\cdot\mathbf{w}$ is equivalent to maximizing the margin. The weight vector $\mathbf{w}$ and the threshold b describe the optimal (maximum margin) hyperplane.

Vapnik [231] showed there is a relation between the margin and the VC-dimension. Considering the hyperplanes $h(\mathbf{x}) = \text{sign}(\mathbf{w}\cdot\mathbf{x}+b)$ in an D-dimensional space as a hypothesis, if all examples $\mathbf{x_i}$ are contained in a hyper-circle of diameter R, it is required that, for examples $\mathbf{x_i}$:

$$\text{abs}(\mathbf{w}\cdot\mathbf{x_i}+b) \geq 1 \tag{5.12}$$

and then this set of hyperplanes has a VC-dimension, v, bounded by:

$$v \leq \min\left(\left\lfloor \frac{R^2}{\rho^2} \right\rfloor, D\right) + 1, \tag{5.13}$$

where $\lfloor c \rfloor$ is the largest integer not greater than c. Thus, the larger the margin, the lower the VC-dimension becomes. Moreover, the VC-dimension of the maximum margin hyperplane does not necessarily depend on the number of features, but on the Euclidean length of the weight vector, $||\mathbf{w}||$, optimized by the SVM. Intuitively this means that the true error of a separating maximum margin hyperplane is close to the training error even in high-dimensional spaces, as long as it has a small weight vector.

The primal constrained optimization problem (5.10) is numerically difficult to handle. Therefore, one introduces Lagrange multipliers, $\alpha_i \geq 0$, with one multiplier for each of the constraints in (5.10), obtaining the Lagrangian function:

$$L(\mathbf{w}, b, \alpha) = \frac{1}{2}||\mathbf{w}||^2 - \sum_{i=1}^{N} \alpha_i [t_i(\mathbf{w}^\top \mathbf{x_i} + b) - 1], \tag{5.14}$$

where $\alpha = [\alpha_1, \ldots, \alpha_N]$. Note the minus sign in front of the Lagrangian multiplier term, because we are minimizing with respect to $\mathbf{w}$ and b, and maximizing with respect to α. Setting the derivatives of $L(\mathbf{w}, b, \alpha)$ with respect to $\mathbf{w}$ and b to zero, we obtain the following two conditions:

$$\mathbf{w} = \sum_{i=1}^{N} \alpha_i t_i \mathbf{x_i}, \tag{5.15}$$

$$0 = \sum_{i=1}^{N} \alpha_i t_i. \tag{5.16}$$

Eliminating $\mathbf{w}$ and b from $L(\mathbf{w}, b, \alpha)$ gives the dual representation of the maximum margin problem:

$$\text{maximize: } \sum_{i=1}^{N} \alpha_i - \frac{1}{2} \sum_{i=1}^{N} \sum_{j=1}^{N} t_i t_j \alpha_i \alpha_j (\mathbf{x_i} \cdot \mathbf{x_j}) \tag{5.17}$$

$$\text{subject to: } \sum_{i=1}^{N} t_i \alpha_i = 0 \tag{5.18}$$

$$\forall i \in [1 \cdots N] : 0 \leq \alpha_i \tag{5.19}$$

The matrix $\mathbf{K}$ with $K_{ij} = t_i t_j (\mathbf{x_i}.\mathbf{x_j})$ is commonly referred to as the Hessian matrix. The result of the optimization process is a vector of Lagrangian coefficients $\alpha = [\alpha_1, \ldots, \alpha_N]$ for which the dual problem (5.17) is optimized. These coefficients can then be used to construct the hyperplane, solving the primal optimization problem (5.10) just by linearly combining training examples (5.20).

$$\mathbf{w} \cdot \mathbf{x} = \left(\sum_{i=1}^{N} \alpha_i t_i \mathbf{x_i} \right) \cdot \mathbf{x} = \sum_{i=1}^{N} \alpha_i t_i (\mathbf{x_i} \mathbf{x}) \qquad \text{and} \qquad b = t_{SV} - \mathbf{w} \cdot \mathbf{x_{SV}} \tag{5.20}$$

Only Support Vectors (SVs) have a non-zero α_i coefficient. To determine b from (5.17) and (5.20), an arbitrary support $\mathbf{x_{SV}}$ vector with its class label t_{SV} can be used.

5.2.2 Soft-Margin SVMs

The last Section considered Hard-margin SVMs where a binary classification problem can be solved by a hyperplane. However, most real world problems are non-linear. In this case where the data set is not linearly separable but we would still like to learn a linear classifier, a loss on the violation of the linear constraints has to be introduced. There is no solution to the optimization problems (5.10) and (5.17), therefore it will be preferable to allow some errors in the training data, as indicated by the structural risk minimization [230].

In the next formulation when trying to find the largest margin, the SVM should allow some misclassified samples. Intuitively, these should be penalized since they fall on the wrong side of the decision hyperplane while we want to minimize the classification errors. The SVM will then try to minimize the influence of difficult samples on the optimization problem. This is known as the soft margin SVM, because it "softens" the constraints regarding the misclassification of the patterns considered harder to classify. The SVs are not required to be positioned exactly on their canonical hyperplanes, but they are allowed to go beyond the other classes' hyperplane (see Figure 5.4). The decision hyperplane is now built from both correct and incorrect samples, using all support vectors [201].

This is the rationale of the soft-margin SVMs. They minimize the weight vector, like the hard-margin SVMs, but they simultaneously minimize the number of training errors by the introduction of slack variables, ξ_i. The primal optimization problem of (5.10) is now reformulated as:

$$\text{minimize: } \frac{1}{2}\mathbf{w} \cdot \mathbf{w} + \mathrm{C} \sum_{i=1}^{N} \xi_i, \tag{5.21}$$

$$\text{subject to: } \forall_{i=1}^{N} : t_i \left[\mathbf{w} \cdot \mathbf{x_i} + b \right] \geq 1 - \xi_i \tag{5.22}$$

$$\forall_{i=1}^{N} : \xi_i > 0. \tag{5.23}$$

If a training example is wrongly classified by the SVM model, the corresponding ξ_i will be greater than 1. Therefore $\sum_{i=1}^{N} \xi_i$ constitutes an upper bound on the number of training errors.

The constraints are then updated to accommodate some misclassified samples, in the form of slack variables ξ_i. These measure "how far" the sample $\mathbf{x_i}$ is from the correct decision hyperplane's side, or, in another words, how "deep" they fall on the wrong side [231]. A visual example can be seen in Figure 5.4. Samples which are on

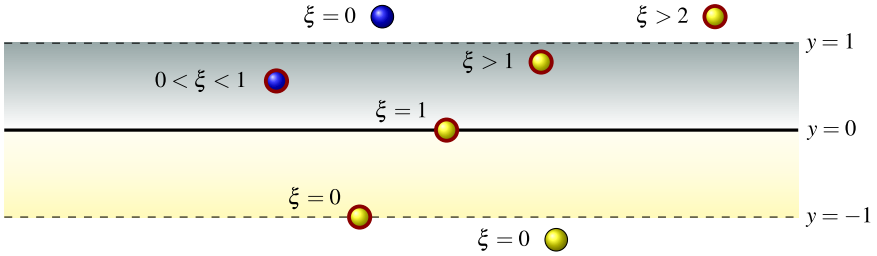


Fig. 5.4 An example with slack variables. Samples marked with red circles correspond to support vectors.

the correct side of the boundary and beyond their classes' canonical hyperplane have $\xi_i = 0$, because their misclassification error is zero. The same applies for samples placed exactly on their classes' canonical hyperplane. Because half of the margin's length is 1, samples which are beyond this hyperplane but not on the other classes' region have $0 < \xi_i < 1$. Samples on the other side of the optimal hyperplane have $\xi_i > 1$.

The factor C in (5.21) is a parameter that allows the trade-off between training error and model complexity. It controls the balance between the margin and the empirical loss, i.e., the structural risk. A larger C allows the training error sum to reach a higher value, which forces the SVM to be more specific to the problem at hand and may lead to overfit the data. In this case, the SVM ends up with a smaller margin. On the other hand, with a lower bound in the error sum the SVM turns out to be more generic. This may underfit the training data and, consequently, the margin gets larger. The C parameter must be carefully chosen in order to achieve the best model.

Again, this primal problem is transformed in its dual counterpart, for computational reasons. The dual problem is similar to the hard-limit SVM (5.17), except for the C upper bound on the Lagrange multipliers α_i:

$$\text{maximize: } \sum_{i=1}^{N} \alpha_i - \frac{1}{2} \sum_{i=1}^{N} \sum_{j=1}^{N} t_i t_j \alpha_i \alpha_j (\mathbf{x_i} \cdot \mathbf{x_j}) \tag{5.24}$$

$$\text{subject to: } \sum_{i=1}^{N} t_i \alpha_i = 0 \tag{5.25}$$

$$\forall i \in [1 \cdots N] : 0 \leq \alpha_i \leq \mathrm{C} \tag{5.26}$$

As before, all training examples with $\alpha_i > 0$ are called support vectors. To differentiate between those with $0 \leq \alpha_i < \mathrm{C}$ and those with $\alpha_i = \mathrm{C}$, the former are called unbounded SVs and the latter bounded SVs.

From the solution of (5.24) the classification rule can be computed exactly, as in the hard-margin SVM:

$$\mathbf{w} \cdot \mathbf{x} = \left(\sum_{i=1}^{N} \alpha_i t_i \mathbf{x_i} \right) \cdot \mathbf{x} = \sum_{i=1}^{N} \alpha_i t_i (\mathbf{x_i} \mathbf{x}) \qquad \text{and} \qquad b = t_{SV} - \mathbf{w} \cdot \mathbf{x_{SV}} \tag{5.27}$$

The only additional restriction is that the SV $(\mathbf{x_{SV}}, t_{SV})$ for calculating b has to be an unbounded SV.

5.2.3 The Nonlinear SVM with Kernels

An extension to nonlinear decision surfaces is necessary since real-life classification problems are hard to be solved by a linear method. While the Soft Margin SVM can give a good performance with non-linearly separable classes, it can be further improved by working in the so-called feature space [231, 230, 201].

The use of kernel functions delivers a powerful method to obtain generalized optimal separating hyperplane in an appropriate feature space [206]. Cover's theorem states that if the transformation is nonlinear and the dimensionality of the feature space is high enough, then input space may be transformed into a new feature space where the patterns are linearly separable with high probability. This nonlinear transformation is performed in an implicit way through the so-called kernel functions.

Definition: Kernel function. A kernel is a function κ that for all $\mathbf{u}$, $\mathbf{v}$ from a space χ (which need not to be a vector space) satisfies

$$\kappa(\mathbf{u}, \mathbf{v}) = \langle \Phi(\mathbf{u}), \Phi(\mathbf{v}) \rangle \tag{5.28}$$

where Φ is a mapping from the space χ to a Hilbert space F that is used called the feature space

$$\Phi : \mathbf{u} \in \chi \longmapsto \Phi(\mathbf{u}) \in F. \tag{5.29}$$

This allows the SVM to find a decision boundary which better discriminates both classes. The new feature space is of a higher dimension and created using the kernel trick by means of a projection $\Phi(\mathbf{x})$. Therefore with a non-linear kernel the margin corresponds to a linear boundary in this new feature space. A geometric interpretation can be gleaned in Figure 5.5. To verify a function is a kernel one widely used approach is to investigate the finitely positive semidefinite property [206]. A function κ satisfies this property if it is a symmetric function for which the kernel matrices K with $K_{ij} = \kappa(\mathbf{x}_i, \mathbf{x}_j)$ are positive semidefinite. As said above, the optimization task appears in a form of inner products (see equation 5.24) between samples of indexes i and j. In the training phase, we need only the kernel function and Φ does not need to be known since it is implicitly defined by the choice of kernel. Thus, we only need to compute the dot products and do not require, in the optimization task, the image data points $\Phi(\mathbf{x}_i)$ and $\Phi(\mathbf{x}_j)$. This is the so-called kernel trick.

Using the kernel trick, the optimization problem (5.24) – (5.26) for SVM becomes:

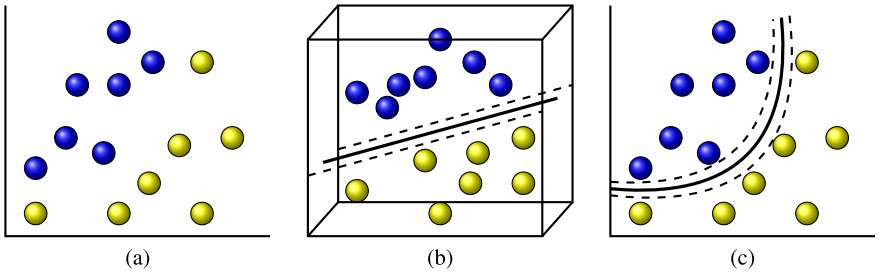


Fig. 5.5 The SVM working in a hypothetical higher feature space. (a) A non-linearly separable case. (b) A higher dimensional feature space where the hyperplane can discriminate both classes. (c) The margin projected back into the original feature space.

$$\text{maximize:} \sum_{i=1}^{N} \alpha_i - \frac{1}{2} \sum_{i=1}^{N} \sum_{j=1}^{N} \alpha_i \alpha_j t_i t_j \kappa(\mathbf{x}_i, \mathbf{x}_j) \tag{5.30}$$

$$\text{subject to:} \sum_{i=1}^{N} \alpha_i t_i = 0 \tag{5.31}$$

$$\forall i \in [1 \cdots N] : 0 \leq \alpha_i \leq \mathrm{C} \tag{5.32}$$

We add the kernel trick to the soft-margin SVM and we still control the structural risk with the penalty C. The function to be maximized is again convex and a Quadratic Programming (QP) solver can be used. Notice that the inner products are only required to be calculated in the kernel feature space. Therefore, before the optimization phase they are stored in the kernel matrix K:

$$K_{N\times N} = \begin{bmatrix} \langle \Phi(\mathbf{x_1}), \Phi(\mathbf{x_1}) \rangle & \langle \Phi(\mathbf{x_1}), \Phi(\mathbf{x_2}) \rangle & \cdots & \langle \Phi(\mathbf{x_1}), \Phi(\mathbf{x_N}) \rangle \\ \langle \Phi(\mathbf{x_2}), \Phi(\mathbf{x_1}) \rangle & \langle \Phi(\mathbf{x_2}), \Phi(\mathbf{x_2}) \rangle & \cdots & \langle \Phi(\mathbf{x_2}), \Phi(\mathbf{x_N}) \rangle \\ \vdots & \vdots & \ddots & \vdots \\ \langle \Phi(\mathbf{x_N}), \Phi(\mathbf{x_1}) \rangle & \langle \Phi(\mathbf{x_N}), \Phi(\mathbf{x_2}) \rangle & \cdots & \langle \Phi(\mathbf{x_N}), \Phi(\mathbf{x_N}) \rangle \end{bmatrix}$$

where K is square matrix with N^2 elements, since there are N vectors $\mathbf{x}$ in the training set.

Variant learning machines are constructed according to the different kernel function $\kappa(\mathbf{u}, \mathbf{v})$ and thus construct different hyperplanes in the feature space. Table 5.1 shows five kernels. A note is given to the polynomial kernel where the parameter a represents the function's derivative while $q > 0$ is the polynomial degree. In the Gaussian kernel the parameter γ controls the kernel width σ. Regarding the sigmoid kernel, the parameter a represents the variation at which the function increases while b sets the function's offset, i.e., where $\tanh(\cdot) = 0$. The Universal Kernel Function (UKF) kernel [257] is a kernel function based on Lorentzian function well-known in the field of statistics. The author's idea is that the kernel function unlike Gaussian kernel must turn very close points in the original

space into weakly correlated elements in the feature space. In this sense, it should have the following characteristics: a quick decrease in the neighborhood of zero point, and a moderate decrease toward infinity. The kernel is an approximation of the KMOD kernel proposed in [8]. Besides the parameter σ, the parameter α controls the decreasing speed around the zero point. The L is a normalization constant which is set to 1 for simplicity [257]. This kernel can provide a smaller number of SVs and thus improve computational costs from both the training and classification tasks. According to [9] the UKF kernel can yield better performance generalization. The SVM's classification for a given point $\mathbf{z}$ is given by:

$$y(\mathbf{z}) = \text{sign}\left(\sum_{i=1}^{N} \alpha_i y_i \kappa(\mathbf{x}_i, \mathbf{z}) + b\right) \tag{5.33}$$

Table 5.1 Summary of inner-product kernels

Kernel Function	Inner Product Kernel
Linear kernel	$\kappa(\mathbf{u},\mathbf{v}) = \langle \mathbf{u},\mathbf{v}\rangle = \sum_{i=1}^{D} \mathbf{u}_i \mathbf{v}_i$
Polynomial kernel	$\kappa(\mathbf{u},\mathbf{v}) = a \cdot (\langle \mathbf{u},\mathbf{v}\rangle + b)^q,$
Gaussian kernel	$\kappa(\mathbf{u},\mathbf{v}) = e^{-\gamma\|\mathbf{u}-\mathbf{v}\|^2}$
Multi-layer perceptron (sigmoid)	$\kappa(\mathbf{u},\mathbf{v}) = \tanh(a \cdot \langle \mathbf{u},\mathbf{v}\rangle + b)$
UKF	$\kappa(\mathbf{u},\mathbf{v}) = L(\|\mathbf{u}-\mathbf{v}\|^2 + \sigma^2)^{-\alpha}$

5.3 Optimization Methodologies for SVMs

There are several optimization methods to find the optimal Lagrangian multipliers needed to solve SVMs. For small and moderately sized problems (less than 5,000 instances), interior point algorithms are reliable adequate choices and precise. In [206] an overview of interior point algorithms is given focusing on the Newton-type predictor corrector algorithm. However, due to the Cholesky decomposition needed the computational cost is expensive, in particular for large scale-data since the matrix is scaled by the number of training instances.

For large-scale problems, one should resort to methods that explore sparsity of the dual variables. For memory storage requirements some methodologies for compact representation of the large kernel matrix can be employed. Among these methods [258], chunking and SMO [103] are quite popular. As the name suggests chunking starts with a subset (chunk) of training data and solves a small QP subproblem. Then it retains the support vectors and replace the other data in the chunk with a certain number of data violating conditions, and solves the second subproblem for a new classifier. Each subproblem is initialized with the solutions of

the previous subproblem. At the last step, all nonzero entries of the vector of α_i can be identified.

Osuna et al. [168] gave a theorem on the solution convergence of breaking down a large QP problem into a series of smaller QP subproblems. As long as at least one example violating the KKT conditions is added to the previous subproblem, the objective function is decreased at each step and has a feasible point satisfying all the constraints [175].

In the next section we will see in more detail the SMO algorithm [175, 103] which will serve as the basis for a multi-thread implementation in CPU and also in GPU platforms.

5.4 Sequential Minimal Optimization (SMO) Algorithm

A number of algorithms have been suggested for solving the dual problems. Traditional QP algorithms are not suitable for large size problems due to the following:

1. They require that the kernel matrix is computed and stored in memory, so it requires extremely large memory.
2. These methods involve expensive matrix operations such as the Cholesky decomposition of a large sub-matrix of the kernel matrix.
3. For practitioners who would like to develop their own implementation of an SVM classifier, other methods are advisable

The Sequential Minimal Optimization (SMO) algorithm was proposed by Platt in [175] to quickly solve the SVM QP problem. This algorithm is based on Osuna's decomposition scheme [168] and solves the smallest possible optimization task at each step. Using Osuna's theorem to ensure convergence, SMO decomposes the overall QP problem into QP subproblems. Unlike the Osuna's method SMO at each step, (i) chooses two Lagrange multipliers α_i and α_j to jointly optimize, (ii) finds the optimal values for these multipliers, and (iii) updates the SVMs to reflect the new optimal values. Both multipliers must satisfy the constraints defined in (5.31) [175, 35]. Algorithm 2 details the main steps of the soft-margin SMO algorithm using the kernel trick [35, 32].

Initially the α_i are set to 0 as they satisfy the constraints defined in (5.31). At each step, after choosing i_{high} and i_{low}, the new values for the two Lagrange multipliers α_i^{new} are computed as follows:

$$\alpha_{i_{low}}^{\text{new}} = \alpha_{i_{low}} + y_{i_{low}} \frac{b_{high} - b_{low}}{\eta} \tag{5.34}$$

$$\alpha_{i_{high}}^{\text{new}} = \alpha_{i_{high}} + y_{i_{low}} y_{i_{high}} (\alpha_{i_{low}} - \alpha_{i_{low}}^{\text{new}}) \tag{5.35}$$

where η is defined as:

Algorithm 2 Sequential Minimal Optimization (SMO) algorithm.

Require: $\mathbf{x_i} \in \chi$, $y_i \in \Omega$, $i \in \{1 \cdots n\}$
1: $i_{high} \leftarrow 0$
2: $i_{low} \leftarrow 0$
3: $b_{high} \leftarrow -1$
4: $b_{low} \leftarrow 1$
5: **for** $n \leftarrow 1$ **to** N **do**
6: $\alpha_i \leftarrow 0$
7: $f_i \leftarrow -y_i$
8: **if** $y_i = 1$ **and** $i_{high} = 0$ **then** $i_{high} \leftarrow i$
9: **if** $y_i = -1$ **and** $i_{low} = 0$ **then** $i_{low} \leftarrow i$
10: **end for**
11: **Update** $\alpha_{i_{low}}$ and $\alpha_{i_{high}}$
12: **repeat**
13: **for** $n \leftarrow 1$ **to** N **do**
14: Update optimality conditions f_i (see (5.37))
15: **end for**
16: **Compute** b_{high}, b_{low}, i_{high} and i_{low}
17: **Update** $\alpha_{i_{low}}$ and $\alpha_{i_{high}}$
18: **until** $b_{low} \leq b_{high} + 2\tau$

$$\eta = K(x_{i_{high}}, x_{i_{high}}) + K(x_{i_{low}}, x_{i_{low}}) - 2 \cdot K(x_{i_{high}}, x_{i_{low}}) \tag{5.36}$$

Naturally, $\alpha_{i_{low}}$ and $\alpha_{i_{high}}$ must satisfy (5.31). Thus, if $\alpha_{i_{low}}$ changes by δ then $\alpha_{i_{high}}$ changes by the same amount on the opposite direction $(-\delta)$. Next, the KKT conditions must be updated for each sample $\mathbf{x_i}$:

$$f_i = f_i^{old} + (\alpha_{i_{high}}^{new} - \alpha_{i_{high}}) y_{i_{high}} K(x_{i_{high}}, x_i) + (\alpha_{i_{low}}^{new} - \alpha_{i_{low}}) y_{i_{low}} K(x_{i_{low}}, x_i) \tag{5.37}$$

The indices of the next Lagrange multipliers i_{low} and i_{high} are chosen from two corresponding sets:

$$I_{low} = \{i : 0 < \alpha_i < \mathrm{C}\} \cup \{i : y_i > 0, \alpha_i = \mathrm{C}\} \cup \{i : y_i < 0, \alpha_i = 0\} \tag{5.38}$$

$$I_{high} = \{i : 0 < \alpha_i < \mathrm{C}\} \cup \{i : y_i > 0, \alpha_i = 0\} \cup \{i : y_i < 0, \alpha_i = \mathrm{C}\} \tag{5.39}$$

The optimality coefficients b_{low} and b_{high} are calculated as:

$$b_{low} = \max\{f_i : i \in I_{low}\} \tag{5.40}$$

$$b_{high} = \min\{f_i : i \in I_{high}\} \tag{5.41}$$

For simplicity, to choose i_{low} and i_{high} we use the first order heuristic [103]. For the next iteration, these indices are calculated as:

$$i_{low} = \arg\max\{f_i : i \in I_{low}\} \tag{5.42}$$

$$i_{high} = \arg\min\{f_i : i \in I_{high}\} \tag{5.43}$$

The algorithm is executed until the optimality gap $(b_{low} - b_{high})$ is smaller than a threshold 2τ:

$$b_{low} \leq b_{high} + 2\tau \Leftrightarrow b_{low} - b_{high} \leq 2\tau \tag{5.44}$$

where $\tau : 0 < \tau < 1$ is the tolerance of the solution optimality and in fact the stopping criteria. After converging, the parameter b can be calculated using (5.27)

5.5 Parallel SMO Implementations

In this section we describe both the parallel CPU and GPU implementations of the SMO algorithm. The CPU parallel implementation was developed in C++ using the Open Multi-Processing (OpenMP) API, which allows to write multi-threaded shared memory (also named Unified Memory Access (UMA)) applications in either C/C++ or Fortran and the GPU implementation using CUDA.

The approach consisted of identifying the SMO steps that are simultaneously responsible for large portions of the overall computation and that could be safely parallelized. Even though the SMO algorithm is in its essence sequential, our emphasis is that there are steps which can be safely parallelized. In fact, as it turns out, these steps correspond to the portions of the algorithm that are very computationally demanding [32, 35]. One of such steps, as noted by Cao et al. [32], is the KKT conditions update (using f_i and the kernel Gram matrix). Since each f_i can be computed independently, we can take full advantage of CPU and GPU multi-core architectures to implement this step.

Another step that can be accelerated is the computation of the next b_{low}, b_{high}, α_{low} and α_{high} values. Since this is accomplished by performing a first order heuristic search, this task can be executed in parallel by using a reduction process. Moreover, the offset b is also computed in parallel for each SV. Thus, the only sequential steps are the update of the Lagrange multipliers ($\alpha_{i_{low}}$ and $\alpha_{i_{high}}$) and the convergence verification (see steps 17 and 18 of Algorithm 2).

In the OpenMP CPU multi-thread implementation, referred as MT-SVM, the above parallel tasks are divided into several (nearly identical) parts, each performed by a distinct thread. Accordingly, each thread will process an equally sized group of samples. A master thread is responsible for splitting the execution into multiple threads and for waiting for them to complete their execution. Figure 5.6 shows the resulting fork-join model.

Prior to our SVMs GPU implementation, four other GPUs implementations existed: Catanzaro et al. gpuSVM [35], Herrero-Lopez et al. multiSV [84], Carpenter cuSVM [33] and Lin and Chien sparse SVM [118]. All of these are written in CUDA. The proposed GPU implementation follows the Catanzaro et al. gpuSVM work [35], which explores the most expensive computation bound step of the SMO algorithm – the update of KKT conditions. Moreover, their implementation also makes use of a second order heuristic from Fan et al. [61], which tries to choose the next Lagrange multipliers that maximize the change in

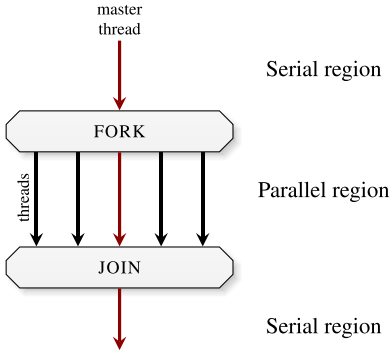


Fig. 5.6 OpenMP fork-join model

the objective function. However, using this heuristic may drastically reduce the algorithm performance. Therefore in our implementation, we choose to use a first order heuristic, in order to search for the next Lagrange multipliers to be updated. Finally, their algorithm uses a cache memory to place the most utilized kernel dot-products in order to reduce the amount of calculus and speedup the training process. Compared to the well-known Library for Support Vector Machines (LIBSVM) software, they presented a performance increase ranging from 9 to 35×, for the training implementation. In the classifier implementation, they parallelized the kernel dot product between the support vectors and the testing samples using the **CUBLAS!** (**CUBLAS!**) library, followed by a sum-reduce operation. Applying this approach, they obtained an average speedup of 110× [35].

The remaining implementations are similar to the one presented by Catanzaro et al.. The advantage of the Herrero-Lopez et al. multiSV implementation is that it executes different binary classifiers at the same time [84]. The Carpenter improves the Catanzaro et al. implementation by using mixed floating point arithmetic precision. Although most computations are performed in 32-bit (float) precision, some are carried out and stored in double precision (64-bit). This is of major importance for some data sets, like the *Forest cover type* (see Appendix A.4, page 207). Finally, Lin and Chien proposed the usage of sparse matrices for caching the kernel matrix. They claim an speedup over gpuSVM of 1.9× to 2.41× [118].

Considering our GPU implementation (GPU-SVM), each thread will work on an independent sample. Once again, the rationale is to maximize occupancy, by having enough threads executing in parallel. Altogether, five CUDA kernels were developed to implement the SMO algorithm: the `InitializeSMO`, which is called only once to initialize the algorithm; the `UpdateKKTConditions`, whose purpose is to update the KKT conditions; the `FirstOrderHeuristic1stPass` and `FirstOrderHeuristicFinalPass`, which are used to implement the first order heuristic search; and finally the `UpdateAlphas`, which updates the Lagrange (α) multipliers. Figure 5.7 shows the sequence of kernel calls performed in each iteration.

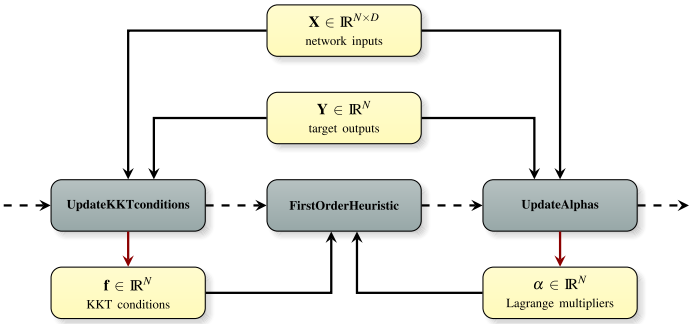


Fig. 5.7 SMO kernels calls in each iteration. In the FirstOrderHeuristic step two kernels (FirstOrderHeuristic1stPass and FirstOrderHeuristicFinalPass) are actually called.

Listing 5.1 CUDA device function for computing the UKF kernel dot product.

```
__device__ cudafloat DotProductKernelUKF(
    int i0, int i1, cudafloat * samples, int n_samples,
    int num_dimensions, cudafloat * kernel_args)
{
    // K(x1,x2) = a(||x1 - x2||^2 + b^2)^-c
    cudafloat sum_dif_squared = 0;
    for (int i = 0; i < num_dimensions; i++) {
        cudafloat x0_i = samples[n_samples * i0 + i];
        cudafloat x1_i = samples[n_samples * i1 + i];

        cudafloat _dif = x0_i - x1_i;
        cudafloat _dif_sq = _dif * _dif;
        sum_dif_squared += _dif_sq;
    }

    cudafloat a = kernel_args[0];
    cudafloat b = kernel_args[1];
    cudafloat c = kernel_args[2];

    return a * CUDA_POW(sum_dif_squared + b * b, -c);
}
```

As in the MBP GPU implementation, the convergence of the SMO algorithm is checked by asynchronously querying the device. This is needed to avoid putting the training process on hold while waiting for the host to receive the required information.

One of the main advantages of our implementation is that it features the UKF kernel, which can improve the models performance generalization. Listing 5.1 shows a CUDA device function that computes the dot product for this kernel.

Table 5.2 Datasets and RBF kernel parameters used in the experiments

Dataset (benchmark)	**Samples** (N)	**Features** (D)	C	γ
Adult	32,561	14	1.00	0.100
Breast cancer	569	30	3.00	0.050
German credit data	1,000	59	1.00	0.050
Haberman's survival	306	3	1.00	1.000
Heart - Statlog	270	20	0.10	0.050
Ionosphere	351	34	1.00	0.500
Sonar	208	60	3.00	0.050
Tic-Tac-Toe	958	9	1.00	0.001
Two-spirals	2,097,152	2	3.00	0.250
MP3 Steganalysis	1,994	742	0.10	0.250

5.6 Results and Discussion

In this Section, we evaluate the results of the CPU multi-thread implementation (MT-SVM) and the GPU-SVM implementations by comparing them with the well-known LIBSVM.

5.6.1 Experimental Setup

In order to evaluate the MT-SVM and the GPU-SVM implementations, with respect to the performance accuracy and speedup, we compared the results obtained for several benchmarks with the corresponding ones of the state-of-the-art LIBSVM (version 3.11) [40], which also uses the SMO algorithm. The LIBSVM was configured to use one Megabyte of cache, since currently our implementation [74] does not make use of a kernel cache. The tests were conducted using system 4 (see Table A.1, page 202). Moreover, in the case of the MT-SVM, the number of threads was set to 4. Additionally, the optimality gap τ was set to 0.01 (see (5.44)).

Currently the implementations are designed exclusively for binary tasks, thus we have specifically chosen two classes benchmarks. Table 5.2 lists the main characteristics of the datasets as well as the best RBF kernel parameters, which were determined by grid search. For more information on the datasets, please refer to Appendix A.4. Note that for the *Two-spirals* dataset, $2^{21} = 2,097,152$ samples were generated.

For each dataset, 10 experiments using 5-fold cross validation were performed.

5.6.2 Results on Benchmarks

Table 5.3 shows the classification performance of the models generated by MT-SVM, GPU-SVM and LIBSVM for the aforementioned datasets. Both the MT-SVM and the GPU-SVM yield competitive performance as compared to LIBSVM. The exception is the *Ionosphere* dataset, for which the SVM-GPU version fails to achieve a good performance. This may be explained by the usage of 32-bit floating precision arithmetic (float data type) and demonstrates that for some problems (64-bit) double precision is required. However, the GPU model could be used as a base for creating a better model, for example using the MT-SVM.

Table 5.3 MT-SVM, GPU-SVM and LIBSVM classification performance

	Accuracy (%)			F-Measure (%)		
Dataset	**MT-SVM**	**GPU-SVM**	**LIBSVM**	**MT-SVM**	**GPU-SVM**	**LIBSVM**
Adult	84.65±0.39	80.97±0.80	**84.72±0.38**	90.13±0.28	86.76±0.67	**90.38±0.24**
Breast Cancer	97.48±2.26	95.42±2.02	**97.76±1.49**	96.59±1.90	93.32±3.12	**96.96±2.02**
German	**73.61±1.65**	73.28±2.90	73.03±1.32	83.44±1.08	81.15±2.36	**83.46±0.82**
Haberman	71.85±4.42	72.72±4.06	**72.92±3.50**	82.14±3.13	83.11±2.66	**83.49±2.31**
Heart	83.18±4.64	**83.33±4.92**	82.37±4.96	**85.44±4.09**	84.73±4.70	85.30±4.00
Ionosphere	**89.66±3.38**	67.52±2.00	89.06±3.58	**91.16±3.13**	79.55±1.06	90.72±3.29
Sonar	85.77±4.90	**88.16±4.57**	84.65±4.78	87.30±4.39	**88.06±5.18**	86.83±4.00
Tic-tac-toe	97.70±1.22	**98.98±0.88**	97.72±1.26	98.28±0.90	**99.22±0.67**	98.29±0.93
Two-Spiral	**100.00±0.00**	**100.00±0.00**	**100.00±0.00**	**100.00±0.00**	**100.00±0.00**	**100.00±0.00**
MP3 Steganalysis	**97.05±0.87**	96.07±1.29	96.92±1.00	**97.06±0.88**	96.19±1.21	96.94±0.99

Table 5.4 shows the number of SVs of the models generated by MT-SVM, GPU-SVM and LIBSVM. Note that, despite producing competitive models, the MT-SVM generates models with fewer SVs. On the other hand, the GPU-SVM implementation generates models with a larger number of SVs.

Table 5.4 Number of Support Vectors (SVs) of the models generated by MT-SVM, GPU-SVM and LIBSVM

Dataset	**MT-SVM**	**GPU-SVM**	**LIBSVM**
Adult	**9,781.76 ± 56.38**	9,801.94 ± 56.43	9,788.40 ± 48.90
Breast Cancer	**113.28 ± 05.18**	115.46 ± 04.85	114.44 ± 04.89
German	**713.70 ± 05.53**	718.94 ± 05.04	718.74 ± 05.14
Haberman	**149.14 ± 04.84**	152.42 ± 04.55	151.20 ± 04.41
Heart	**175.74 ± 03.09**	178.28 ± 02.98	177.18 ± 03.10
Ionosphere	**215.10 ± 03.08**	218.72 ± 03.20	217.74 ± 03.23
Sonar	**151.10 ± 02.57**	154.82 ± 02.39	153.74 ± 02.36
Tic-tac-toe	**548.36 ± 10.08**	553.18 ± 10.58	551.78 ± 10.89
Two-Spiral	**698.80 ± 28.65**	1,053.10 ± 67.49	939.86 ± 58.23
MP3 Steganalysis	**346.16 ± 07.11**	348.82 ± 07.06	348.08 ± 07.16

Tables 5.5 and 5.6 show respectively the amount of time needed for the training and classification tasks and the corresponding speedups obtained by the MT-SVM and the GPU-SVM implementations, relatively to LIBSVM. Note that, only the three heaviest datasets (*Adult*, *Two-Spiral* and *MP3 steganalysis*) are shown, since for the smaller datasets the GPU speedups will be negative (< 1). Both the MT-SVM and the GPU-SVM implementations can boost significantly the training and the classification tasks. This takes particular relevance for the training task as it can considerably reduce the time required to perform a grid search (a fundamental process to obtain good generalization models).

Table 5.5 Training and classification times (in seconds) for the MT-SVM and GPU-SVM and LIBSVM implementations

	Training			**Classification**		
Dataset	**MT-SVM**	**GPU-SVM**	**LIBSVM**	**MT-SVM**	**GPU-SVM**	**LIBSVM**
Adult	14.83±0.42	**2.24±0.22**	30.49±0.72	0.84±0.12	**0.03±0.01**	2.02±0.07
Two-Spiral	21.26±0.19	**0.89±0.01**	146.72±0.42	3.02±0.50	**0.04±0.11**	11.72±0.19
MP3 Steganalysis	**0.34±0.02**	0.68±0.07	2.37±0.05	**0.02±0.01**	0.06±0.01	0.57±0.02

Table 5.6 Speedups ($\times$) obtained for the MT-SVM and GPU-SVM relatively to LIBSVM implementation

	MT-SVM		**GPU-SVM**	
Dataset	**Training**	**Classification**	**Training**	**Classification**
Adult	2.06	2.27	13.61	67.33
Two-Spiral	6.90	3.88	165.15	265.85
MP3 Steganalysis	6.88	29.53	3.48	9.50

Regarding the UKF kernel, Table 5.7 presents the classification results obtained using the MT-SVM implementation. The additional number of parameters greatly increases the complexity of performing a grid search. Having said that, it is possible that the results could be improved by narrowing the search. Nevertheless, the results show the usefulness of the this kernel. Using the Wilcoxon signed ranked test we found no statistical evidence of the UKF kernel performing worse than the RBF kernel and vice-versa.

Table 5.8 presents the average number of SV of the models generated. It is interesting to note that, when the UKF presents a smaller number of SVs than the RBF kernel it generally yields better F-Measure results. This seems to indicate that the UKF classification performance improves when it is able to gather points near to each other, in a higher dimension space, as intended.

Table 5.7 RBF versus UKF kernel classification results

	RBF kernel		UKF kernel	
Accuracy (%)	**F-Meausure (%)**	**Classification**	**Accuracy (%)**	**F-Meausure (%)**
Adult	**84.65±0.39**	**90.13±0.28**	83.36±0.37	89.47±0.26
Breast Cancer	97.48±2.26	96.59±1.90	**98.11±1.00**	**97.39±1.46**
German	**73.61±1.65**	83.44±1.08	71.79±3.15	83.04±2.08
Haberman	71.85±4.42	82.14±3.13	**72.45±4.91**	**82.85±3.58**
Heart	**83.18±4.64**	**85.44±4.09**	82.93±3.59	84.85±3.41
Ionosphere	89.66±3.38	91.16±3.13	**94.28±3.04**	**95.61±2.30**
Sonar	**85.77±4.90**	**87.30±4.39**	85.10±4.81	86.19±4.69
Tic-tac-toe	97.70±1.22	98.28±0.90	**98.17±0.94**	**98.59±0.76**
Two-Spiral	**100.00±0.00**	**100.00±0.00**	**100.00±0.00**	**100.00±0.00**
MP3 Steganalysis	**97.05±0.87**	**97.06±0.88**	93.02±0.78	93.12±0.83

Table 5.8 Average number of SVs of the RBF versus UKF models

	RBF kernel	UKF kernel
Adult	**9,781.76 ± 56.38**	12,543.60 ± 56.63
Breast Cancer	113.28 ± 05.18	**63.08 ± 03.03**
German	**713.70 ± 05.53**	795.60 ± 13.89
Haberman	**149.14 ± 04.84**	231.08 ± 04.03
Heart	**175.74 ± 03.09**	181.52 ± 05.52
Ionosphere	215.10 ± 03.08	**101.40 ± 03.99**
Sonar	151.10 ± 02.57	**129.76 ± 03.84**
Tic-tac-toe	548.36 ± 10.08	**475.64 ± 10.95**
Two-Spiral	698.80 ± 28.65	**324.00 ± 03.66**
MP3 Steganalysis	**346.16 ± 07.11**	1019.44 ± 08.51

5.7 Conclusion

As the amount of data produced grows at an unprecedented rate fast machine learning algorithms that are able to extract relevant information from large repositories have become extremely important. To partly answer to this challenge in this Chapter we presented a multi-threaded parallel MT-SVM which parallelizes the SMO algorithm [74]. This implementation uses the power available on multi-core CPUs and GPUs, and efficiently learns (and classifies) within several domains, exposing good properties in scaling data.

Additionally, the UKF kernel which has good generalization properties in the high-dimensional feature space has been included, although more parameters are needed to fine tune the results. It would be interesting to account for vectorization (SSE or AVX) as well as support for kernel caching which may drastically decrease the amount of computation.

Chapter 6
Incremental Hypersphere Classifier (IHC)

Abstract. In the previous chapters we have presented batch learning algorithms, which are designed under the assumptions that data is static and its volume is small (and manageable). Faced with a myriad of high-throughput data usually presenting uncertainty, high dimensionality and large complexity, the batch methods are no longer useful. Using a different approach, incremental algorithms are designed to rapidly update their models to incorporate new information on a sample-by-sample basis. In this chapter we present a novel incremental instance-based learning algorithm, which presents good properties in terms of multi-class support, complexity, scalability and interpretability. The Incremental Hypersphere Classifier (IHC) is tested in well-known benchmarks yielding good classification performance results. Additionally, it can be used as an instance selection method since it preserves class boundary samples.

6.1 Introduction

Learning from data streams is a pervasive area of increasing importance. Typically, stream learning algorithms run in resource-aware environments, constructing decision models that are continuously evolving and tracking changes in the environment generating the data [66]. This contrasts with traditional ML algorithms that are commonly designed with the emphasis set on effectiveness (e.g. classification performance) rather than on efficiency (e.g. time required to produce a classifier) [263] and predominantly focus on one-shot data analysis from homogeneous and stationary data [66]. Generally, it is assumed that data is static and finite and that its volume is "small" and manageable enough for the algorithms to be successfully applied in a timely manner. However, these two premises rarely hold true for modern databases and although (as we have seen) the GPU implementations can reduce considerably the time necessary to build the models, there are many real-world scenarios for which traditional batch algorithms are inapplicable [69]. Rationally, when dealing with large amounts of data it is conceivable that the memory capacity will be insufficient to store every piece of relevant information

N. Lopes and B. Ribeiro, *Machine Learning for Adaptive Many-Core Machines – A Practical Approach,* Studies in Big Data 7,
DOI: 10.1007/978-3-319-06938-8_6,

during the whole learning process [96]. Moreover, even if the required memory is available, the computational requirements to process such amounts of data in a timely manner can be prohibitive, even when GPU implementations are considered. Additionally, modern databases are dynamic by nature. They are incessantly being fed with new information and it is not uncommon for the original concepts to drift [131, 134]. Therefore, both real-time model adaptation and classification are crucial tasks to extract valuable and up-to-date information from the original data, playing a vital role in many industry segments (e.g. financial sectors, telecommunications) that rely on knowledge discovery in databases and data mining tasks to stay competitive [186]. Reducing both the memory and the computational requirements inherent to ML algorithms can be accomplished by using instance selection methods that select a representative subset of the data. The idea is to identify the relevant instances for the learning process while discarding the superfluous ones. These methods can be divided into two groups (wrapper and filter) according to the strategy used for choosing the instances [167, 105]. Unlike filter methods, wrapper methods use a selection criterion that is based on the accuracy obtained by a classifier (instances that do not contribute to improve the accuracy are discarded). A review of both wrapper and filter methods can be found in López et al. [167]. Although instance selection methods can effectively reduce the volume of data to be processed, their application may be time consuming (in particular for wrapper methods) and in some situations we may find that we are simply transferring the complexity from the learning methods to the instance selection methods. Usually, instance selection methods present scaling problems: for very large datasets the run-times may grow to the point where they become inapplicable [167]. A more desirable approach to deal with the memory and computational limitations consists of using incremental learning algorithms. In this approach, the learner gradually adjusts its model as it receives new data. At each step the algorithm can only access a limited number of new samples (instances) from which a new hypothesis is built upon [96]. Another important consideration when extracting information from data repositories is the interpretability of the resulting models. In some application domains, the comprehensibility of the decisions is as valuable as having accurate models. Moreover, understanding the predictions of the models can improve the users' confidence on them [172, 17].

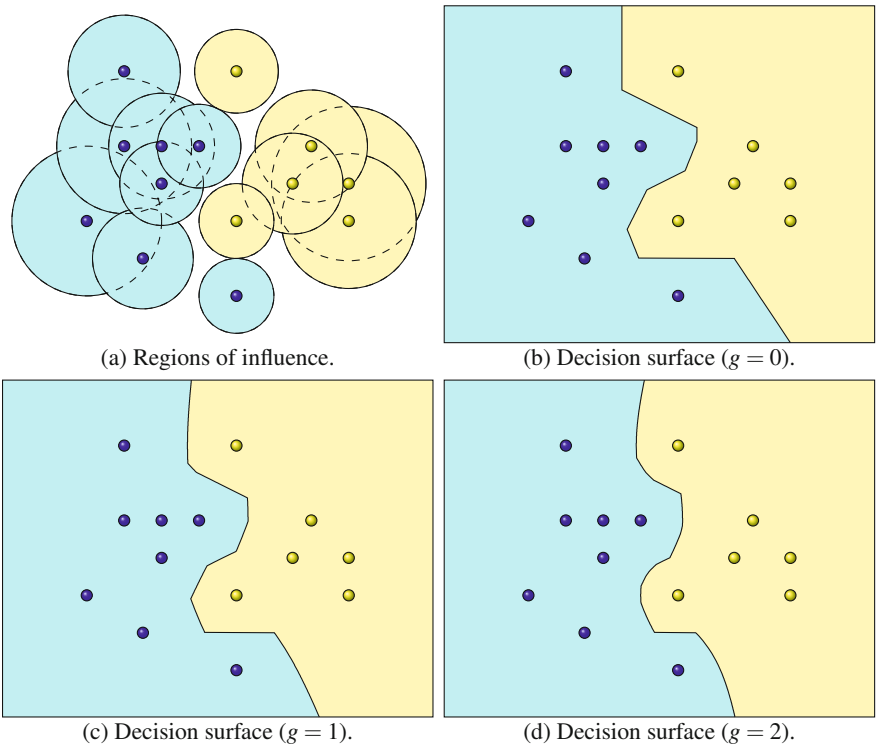
6.2 Proposed Incremental Hypersphere Classifier Algorithm

The IHC is a relatively simple algorithm. Its main task consists of assigning a region (zone) of influence to each sample, by which classification is achieved. The region of influence of a given sample i is then defined by an hypersphere of radius ρ_i, given by (6.1):

$$\rho_i = \frac{\min(\left|\left|\mathbf{x_i} - \mathbf{x_j}\right|\right|)}{2}, \text{ for all } j \text{ where } y_j \neq y_i \tag{6.1}$$

Note that unlike the NNs models, the IHC produces a single discrete output value, $y \in \{1,...,C\}$ that differentiates C classes, i.e. $f : \mathbb{R}^D \longrightarrow \{1,...,C\}$.

The radius is defined so that hypersphere's belonging to different classes do not overlap each other. However, regions of influence belonging to the same class may overlap one another. Given any two samples i and j of different classes ($y_i \neq y_j$), the radiuses (ρ_i and ρ_j) will be at most half of the distance between the two input vectors ($\mathbf{x_i}$ and $\mathbf{x_j}$), which is the maximum value that ρ_i and ρ_j can have without overlapping their hypersphere's. Figure 6.1(a) shows the regions of influence for a chosen toy problem.



(a) Regions of influence. (b) Decision surface ($g = 0$). (c) Decision surface ($g = 1$). (d) Decision surface ($g = 2$).

Fig. 6.1 Application of the IHC algorithm to a toy problem

New data points are classified according to the class of the nearest region of influence (not the nearest sample). Let $\mathbf{x_k}$ represent an input vector whose class y_k is unknown. Then, sample k belongs to class y_i ($y_k = y_i$) provided that:

$$||\mathbf{x_i} - \mathbf{x_k}|| - g\mathrm{a}_i\rho_i \leq \left|\left|\mathbf{x_j} - \mathbf{x_k}\right|\right| - g\mathrm{a}_j\rho_j, \text{ for all } j \neq i \tag{6.2}$$

where g (gravity) controls the extension of the zones of influence, increasing or shrinking them and a_i is the accuracy of sample i when classifying itself and

the forgotten training samples for which i was the nearest sample in memory. A forgotten sample is one that either has been removed from memory or did not qualify to enter the memory in the first place. Hence, the accuracy is only updated when the memory is full. In such a scenario, at each iteration, the accuracy of a single (nearest) sample is updated, while the accuracy of all the others remains unmodified.

The accuracy is the first mechanism of defense against outliers. As it decreases so does the influence of the associated hypersphere. This effectively reduces the damage caused by outliers and by samples with zones of influence that are excessively large.

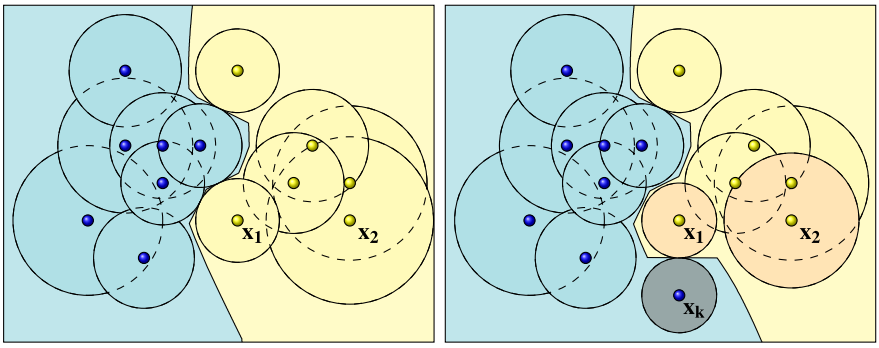
Figures 6.1(b), 6.1(c) and 6.1(d) show the decision surface generated by the IHC algorithm for a toy problem, considering different gravity, g, values. Note that for $g = 0$ the decision rule of the IHC is exactly the same as the one of the 1-nearest neighbor (see (6.2)). A detailed description of the k-nn can be found in Clarke et al. [49]. It is interesting to point out that (for $g > 0$) the IHC algorithm generates smoother decision surfaces (see Figures 6.1(b), 6.1(c) and 6.1(d)).

Note that the farthest from the decision border an hypersphere is, the larger its radius will be (see Figure 6.1(a)). This provides a simple method for determining the relevance of a given sample: samples with smaller radius (ρ) are more important to the classification task than those with bigger radius. Therefore, when the memory is full, the radius of a new sample is compared with the radius of the nearest sample of the same class and the one with the smallest radius is kept in the memory while the other is discarded. By doing so, we are keeping the samples that play the most significant role in the construction of the decision surface (given the available memory) while removing those that have less or no impact in the model. The radius of a new sample is only compared with the one of its nearest sample to prevent the concentration of the memory samples in the same space region.

Unfortunately, outliers will most likely have a small radius and end-up occupying our limited memory resources. Thus, although their impact is diminished by the use of the accuracy in (6.2), it is still important to identify and remove them from memory. To address this problem we mimic the process used by the Instance Based learning (IB3) algorithm, which consists of removing all samples that are believed to be noisy by employing a significance test. A detailed description of the IB3 algorithm can be found in Wilson and Martinez [242] or alternatively in Aha et al. [2].

Accordingly, as in the IB3 algorithm, confidence intervals are determined both for the instance accuracy, not including its own classification, unlike in (6.2), and for the relative frequency of the classes. The instances whose maximum (interval) accuracy is less than the minimum class frequency (for the desired confidence level – typically 70%) are considered outliers and consequently dropped off [242, 2].

A major advantage of the IHC algorithm relies on the possibility of building models incrementally on a sample by sample basis. Figure 6.2 shows the regions of influence and the corresponding decision surfaces generated by IHC for a chosen toy problem, before and after the addition of a new sample, k. Notice that adding a new sample might affect the radius of the samples already in the model (in this particular case the ones with the input vectors $\mathbf{x_1}$ and $\mathbf{x_2}$).



(a) Before adding a new sample. (b) After adding a new sample, k.

Fig. 6.2 Regions of influence and decision surfaces generated by IHC for a toy problem ($g = 1$)

Algorithm 3 describes the main steps required to incorporate a new sample, k, on the IHC model. A working version of IHC, including its source code, can be found at `http://sourceforge.net/projects/ihclassifier/`.

To cope with unbalanced datasets and avoid storing a disproportionate number of samples for each class, the algorithm assumes that the memory is divided into C groups. Considering that the available memory can hold up to n samples, the complexity of this algorithm is $O(2Dn)$.

Another advantage of the algorithm is that it can accommodate the restrictions in terms of memory and computational power, creating the best model possible for the amount of resources given, instead of requiring systems to comply with its own requirements. Since we can control the amount of memory and computational power required by the algorithm (by changing the value of n) and due to its scalability (memory and computational requirements grow linearly with the number of samples stored), it is feasible to create up-to-date models in real-time to extract meaningful information from data.

Moreover, if we limit the number of samples stored, such that $n < N$, then the IHC algorithm can be viewed as an instance selection method, which retains the samples that play the most significant role in the construction of the decision surface while removing those that have less or no impact in the model. In this context, IHC efficiently selects a representative and reduced dataset that can be used to create models using more sophisticated algorithms (e.g. NNs, SVMs), whose application to the original dataset could be impractical.

Algorithm 3 Incremental Hypersphere Classifier (IHC) algorithm.

Input: $\mathbf{x_k}$ ▷ Input vector of the new sample k.
Input: y_k ▷ Class of the new sample k.
$\rho_k \leftarrow \infty$ ▷ Radius of sample k
$d \leftarrow \infty$ ▷ Distance to the nearest sample (using $||\mathbf{x_i} - \mathbf{x_k}|| - ga_i\rho_i$)
$n \leftarrow$ null ▷ Nearest sample (using $||\mathbf{x_i} - \mathbf{x_k}|| - ga_i\rho_i$)
$tp_k \leftarrow 1$ ▷ True positives (classified by sample k)
$fp_k \leftarrow 0$ ▷ False positives ($a_k = \frac{tp_k}{tp_k + fp_k}$)
for $class \leftarrow 1, \ldots, C$ **do**
  for all sample $i \subset memory[class]$ **do**
    if $||\mathbf{x_i} - \mathbf{x_k}|| - ga_i\rho_i < d$ **then**
      $d \leftarrow ||\mathbf{x_i} - \mathbf{x_k}|| - ga_i\rho_i$
      $n \leftarrow i$
    end if
    if $class \neq y_k$ **then**
      if $\frac{||\mathbf{x_i} - \mathbf{x_k}||}{2} < \rho_k$ **then** $\rho_k \leftarrow \frac{||\mathbf{x_i} - \mathbf{x_k}||}{2}$
      if $\frac{||\mathbf{x_i} - \mathbf{x_k}||}{2} < \rho_i$ **then** $\rho_i \leftarrow \frac{||\mathbf{x_i} - \mathbf{x_k}||}{2}$
    end if
  end for
end for
if $memory[y_k]$ **is full and** $n \neq$ null **then**
  if $\rho_n > \rho_k$ **then**
    Remove sample n from $memory[y_k]$
    $d \leftarrow \infty$
    $j \leftarrow$ null ▷ Nearest sample of sample n
    for $class \leftarrow 1, \ldots, C$ **do**
      for all sample $i \subset memory[class]$ **do**
        if $||\mathbf{x_i} - \mathbf{x_n}|| - ga_i\rho_i < d$ **then**
          $d \leftarrow ||\mathbf{x_i} - \mathbf{x_n}|| - ga_i\rho_i$
          $j \leftarrow i$
        end if
      end for
    end for
    if $j \neq$ null **then**
      if $y_n = y_j$ **then** $tp_j \leftarrow tp_j + 1$ **else** $fp_j \leftarrow fp_j + 1$
    end if
  else
    if $y_n = y_k$ **then** $tp_n \leftarrow tp_n + 1$ **else** $fp_n \leftarrow fp_n + 1$
  end if
end if
if $memory[y_k]$ **is not full then** Add sample k to $memory[y_k]$

6.3 Results and Discussion

6.3.1 Experimental Setup

Since the IHC is an instance based classifier, we chose to perform a first comparison of this algorithm with the well-known 1-nn (another instance based classifier),

which has demonstrated good classification performance in a wide range of real-world problems [221]. This experiment is important not only to validate the proposed method but also because as we said before, for $g = 0$ (and assuming sufficient memory to store all the samples) the IHC generates exactly the same decision borders as the 1-nn. Hence, the resulting information will allow us to determine if it is advantageous to use different values of g in order to create smoother decision surfaces. Accordingly, we have carried out experiments on 14 datasets with different characteristics (number of samples, features and classes).

Moreover, we have also conducted further tests, using the same datasets, in order to determine the performance of the algorithm when confronted with memory and computational constraints. To this end, we have compared the IHC algorithm with the IB3, which is also an incremental instance selection algorithm.

For statistical significance, in both cases, each experiment was run using repeated 5-fold stratified cross-validation. Altogether 30 different random cross-validation partitions were created, accounting for a total of 150 runs per benchmark.

The remaining experiments analyze the capacity of the algorithm to deal with large datasets and data streams involving concept drifts. First, we examine the performance of the algorithm on a large dataset (*Knowledge Discovery and Data mining (KDD) Cup 1999*) while varying the amount of memory provided to the algorithm. In this case, the samples are presented to the algorithm in the same order as they appear in the dataset, thus simulating a data stream.

Second, in order to analyze the applicability of the IHC to data streams involving concept drifts, we have conducted experiments on two real-world datasets (*Luxembourg Internet usage* and *Electricity demand*) known to contain concept drifts. To this end, each experiment was run 30 times, varying the order in which the samples were presented. The performance of the IHC was compared with the corresponding one of IB3.

Finally, in a real-world emblematic case study of Protein membership prediction the IHC and SVM algorithms are successfully combined. Albeit the SVMs have been successfully applied in classification of biological data including sub-sequence cellular prediction and protein sequence classification [93, 183, 208] the hybrid approach has shown to be a powerful technique for addressing this problem.

Note that, we specifically choose a relatively small dataset, so that we could optimize the baseline SVM algorithm parameters in order to guarantee the validity of the comparisons between the proposed approach and the baseline method. All the experiments were performed using computer system 2 (see Table A.1, page 202).

6.3.2 Benchmark Results

Table 6.1 reports the macro-average F-measure performance for both the baseline 1-nn and for the IHC using parameters $g = 1$ and $g = 2$ (no memory restrictions were imposed). The IHC algorithm excels the 1-nn in all the experiments except in the *German credit data* (where the 1-nn presents slightly better results). Moreover,

in the case of the *tic-tac-toe* the IHC performs considerably better (with an F-Measure discrepancy of 31.74% for $g = 2$). To validate the referred improvements, we conducted the Wilcoxon signed rank test. The null hypothesis of the 1-nn having an equal or better F-Measure than the IHC algorithm is rejected at a significance level of 0.005 (both for $g = 1$ and $g = 2$). Thus, there is strong evidence that the IHC significantly outperforms the 1-nn. Given the good results obtained, a particular area in which the IHC algorithm may be useful is on the development of ensembles of classifiers.

Table 6.1 IHC and 1-nn classification performance (macro-average F-measure (%)) for the test datasets of the UCI benchmark experiments

Dataset	N	D	C	**1-nn**	**IHC** ($g = 1$)	**IHC** ($g = 2$)
Breast cancer	569	30	2	95.15 ± 0.41	$\mathbf{96.07 \pm 0.30}$	$\mathbf{96.45 \pm 0.36}$
Ecoli	336	7	8	66.04 ± 0.82	$\mathbf{67.51 \pm 0.72}$	$\mathbf{68.03 \pm 0.78}$
German credit data	1000	59	2	$\mathbf{64.38 \pm 0.96}$	63.98 ± 0.95	63.55 ± 0.95
Glass identification	214	9	6	68.77 ± 1.63	$\mathbf{70.30 \pm 2.20}$	$\mathbf{69.81 \pm 2.23}$
Haberman's survival	306	3	2	55.53 ± 2.04	55.26 ± 2.35	$\mathbf{56.36 \pm 1.92}$
Heart - Statlog	270	20	2	75.30 ± 1.60	$\mathbf{75.92 \pm 1.28}$	$\mathbf{76.19 \pm 1.27}$
Ionosphere	351	34	2	85.90 ± 0.69	$\mathbf{90.98 \pm 0.54}$	$\mathbf{92.55 \pm 0.47}$
Iris	150	4	3	95.70 ± 0.69	$\mathbf{95.71 \pm 0.61}$	$\mathbf{96.04 \pm 0.64}$
Pima diabetes	768	8	2	66.95 ± 1.06	$\mathbf{68.41 \pm 1.00}$	$\mathbf{70.09 \pm 0.97}$
Sonar	208	60	2	85.60 ± 1.76	$\mathbf{85.63 \pm 1.79}$	$\mathbf{87.03 \pm 1.50}$
Tic-Tac-Toe	958	9	2	49.47 ± 0.47	$\mathbf{73.43 \pm 0.54}$	$\mathbf{81.21 \pm 0.83}$
Vehicle	946	18	4	69.35 ± 0.76	$\mathbf{69.46 \pm 0.71}$	68.78 ± 0.93
Wine	178	13	3	95.90 ± 0.51	$\mathbf{96.80 \pm 0.44}$	$\mathbf{96.93 \pm 0.64}$
Yeast	1484	8	10	56.32 ± 1.04	$\mathbf{57.73 \pm 1.12}$	$\mathbf{58.75 \pm 0.86}$

While the aforementioned results demonstrate the usefulness of the proposed algorithm, we are particularly interested in its behavior against limited memory and processing power resources. In such scenarios it is up to the algorithm to decide what is relevant and what is accessory (or less relevant). Clearly, there is a trade-off between the amount of information stored and the performance of the resulting models. To exacerbate this problem, the order in which samples are presented may exert a profound impact on the algorithms decisions. Different orders impose distinct biases, affecting the algorithm results.

Table 6.2 compares the performance of the IHC algorithm (for $g = 1$) with the IB3 algorithm. The latter is one of the most successful instance selection and instance-based learning standard algorithms [69], presenting low storage requirements and high accuracy results [242]. Moreover, IB3 is an incremental algorithm, making it the ideal candidate for comparison purposes. For fairness and unbiased comparison, the order in which samples were presented was the same for both algorithms. It is not possible to anticipate the amount of memory that IB3 will require for a given problem. In this aspect the IHC algorithm is advantageous since it respects the memory bounds imposed. Hence, we configured the memory

requirements to match (as closely as possible) those of the IB3 algorithm. On average IB3 presents a storage reduction of 89.69% while the IHC presents a storage reduction of 89.78%. In terms of performance on the test datasets the IHC excels the IB3 in 9 of the 14 benchmarks. On average the IHC algorithm improves the F-Measure by 3.45% relative to the IB3. The performance gap is specially appreciable in the *glass identification*, *Haberman's survival*, *sonar* and *yeast* benchmarks where the F-Measure is improved respectively by 15.80%, 9.25%, 12.36% and 9.69%.

Table 6.2 Classification performance, macro-average F-measure (%), and storage reduction (%) of the IHC and IB3 algorithms for the UCI benchmark experiments

	Storage reduction		**F-Measure (test)**		**F-Measure (overall)**	
Dataset	**IB3**	**IHC**	**IB3**	**IHC**	**IB3**	**IHC**
Breast cancer	93.80 ± 1.18	**93.89 ± 0.07**	93.47 ± 1.02	**93.64 ± 0.97**	94.35 ± 0.66	**94.66 ± 0.70**
Ecoli	**78.98 ± 2.23**	78.73 ± 0.22	63.80 ± 3.41	**65.30 ± 1.88**	60.96 ± 3.28	**83.06 ± 1.58**
German credit data	95.30 ± 1.29	**95.38 ± 0.12**	55.91 ± 2.20	**56.33 ± 2.05**	60.17 ± 1.78	**61.49 ± 0.84**
Glass identification	**86.57 ± 2.40**	86.04 ± 0.22	35.63 ± 2.19	**51.43 ± 3.04**	41.50 ± 3.19	**62.05 ± 1.68**
Haberman's survival	**97.45 ± 1.13**	97.12 ± 0.30	44.85 ± 2.96	**54.10 ± 3.84**	46.21 ± 3.62	**57.33 ± 2.14**
Heart - Statlog	92.44 ± 1.64	**92.76 ± 0.24**	**79.53 ± 1.58**	76.68 ± 2.39	**80.72 ± 0.82**	79.60 ± 1.62
Ionosphere	93.41 ± 2.37	**93.64 ± 0.08**	75.21 ± 4.48	**81.04 ± 2.77**	77.30 ± 3.81	**82.87 ± 2.59**
Iris	81.44 ± 3.28	**82.50 ± 0.00**	**93.87 ± 1.68**	93.60 ± 1.78	94.55 ± 1.10	**95.92 ± 1.20**
Pima diabetes	**94.04 ± 1.41**	94.03 ± 0.15	**66.60 ± 2.33**	64.68 ± 1.81	**69.22 ± 1.78**	68.01 ± 1.10
Sonar	**97.63 ± 1.44**	97.62 ± 0.07	48.26 ± 7.37	**60.62 ± 4.83**	49.71 ± 7.26	**62.05 ± 3.39**
Tic-Tac-Toe	**93.99 ± 2.25**	93.88 ± 0.11	61.56 ± 3.80	**61.99 ± 1.57**	63.81 ± 4.00	**66.67 ± 0.85**
Vehicle	**85.48 ± 1.39**	85.23 ± 0.05	**62.26 ± 1.13**	60.90 ± 1.74	68.15 ± 0.94	**68.20 ± 1.05**
Wine	82.48 ± 1.99	**83.15 ± 0.16**	**94.03 ± 1.28**	93.22 ± 1.43	94.93 ± 0.88	**95.59 ± 0.84**
Yeast	82.68 ± 1.38	**82.90 ± 0.09**	37.52 ± 3.68	**47.21 ± 1.25**	45.00 ± 4.51	**61.70 ± 0.90**

Real-world databases often present a high-degree of redundancy. In some situations similar (or identical) records may be frequent. Therefore it is important to determine the performance of the algorithms on the forgotten data. To this end, Table 6.2 includes the overall performance (train and test data). With respect to the aforementioned, using the Wilcoxon signed rank test, we reject the null hypothesis of IB3 having an equal or better expected F-measure at a significance level of 0.005. Hence, there is strong statistical evidence that the IHC algorithm preserves more (better) information of the forgotten samples than the IB3 algorithm, thus yielding superior results. This is accomplished by using the information of each sample that was ever presented to update the model (i.e. the radius of the samples of different classes in the memory). On average the IHC algorithm improves the F-Measure by 6.62%. Moreover, in the *ecoli*, *glass identification* and *yeast* benchmarks the performance gap is particularly evident with a respective improvement of 22.10%, 20.55%, 16.70%. It is worth mentioning that the IHC results could be enhanced by fine-tuning the value of g.

In order to analyze the behavior of the IHC algorithm on a large database, we have applied it to the *KDD Cup 1999* dataset, described Appendix A.4 (see page 210). In real-world scenarios we cannot control the order in which samples appear, thus they were presented to the algorithm in the same order as they appear on the

dataset. Figures 6.3 and 6.4 show respectively the time required to update the model and the accuracy according to the memory used by the algorithm.

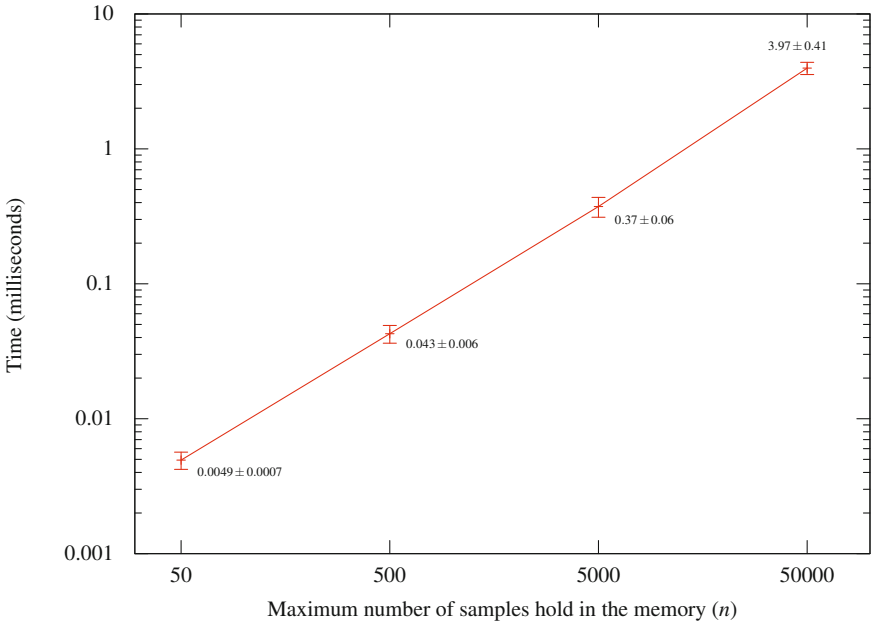


Fig. 6.3 Average time required to update the IHC model (after presenting a new sample) for the *KDD Cup 1999* dataset

As expected the time necessary to update the model grows linearly with the amount of memory used, making the IHC a highly scalable algorithm. To update the model requires approximately 4 milliseconds for $n = 50,000$. This demonstrates real-time model adaptability and knowledge extraction are feasible.

The accuracy depends substantially on the amount of memory supplied to the algorithm. Lower memory bounds imply larger oscillations on the accuracy (see Figure 6.4). These occur when samples conveying information that is not yet covered by the model concepts are presented to the algorithm. In this problem, the first $7,448$ samples belong to the normal class and within the first $75,985$ only 4 samples belong to another class (unauthorized access to local superuser privileges (U2R)). At this point the model concepts do not account for any other classes and samples belonging to them will therefore be misclassified. As a result a sudden decrease in the models accuracy is experienced. Eventually, when the new concepts are integrated in the model, the accuracy recovers. Rationally, if the memory footprint is too small we may find ourselves in a position where there is not enough information to separate useful from accessory information. For example, for $n = 50$ only 10 samples per class can be stored. Given that the first $7,448$ samples belong to the same class, there is no way for the algorithm to make the correct

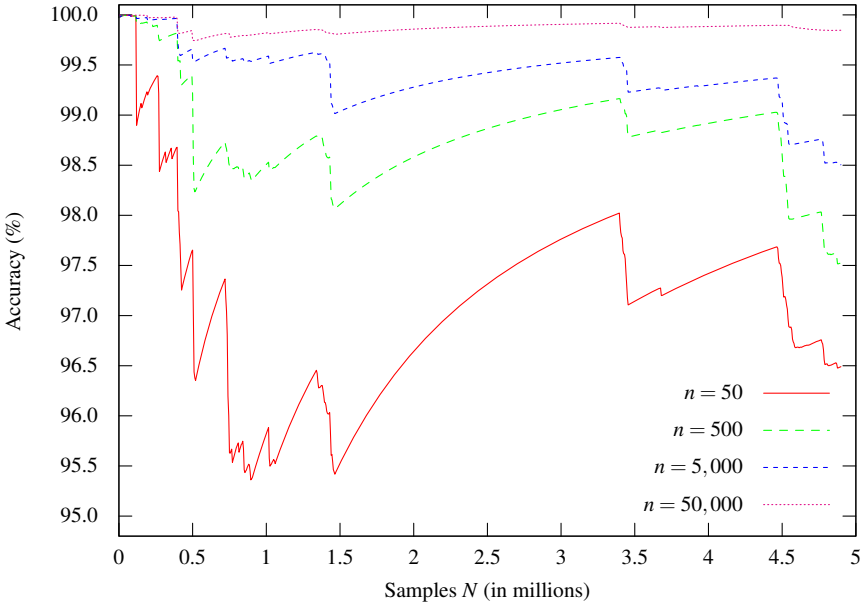


Fig. 6.4 Accuracy of the IHC model for the *KDD Cup 1999* dataset

(informed) decision of which samples to retain in the memory. Of course the larger the number of samples the algorithm is allowed to store, the greater the chances it has to preserve those lying near the decision border. Therefore, lower memory bounds result in accentuated oscillations and in a reduced overall accuracy (as depicted in Figure 6.4). Nevertheless, the algorithm presents a good classification performance even with as little memory as necessary to store 50 samples.

To analyze the performance of the IHC algorithm on data streams, two real-world datasets (*Luxembourg Internet usage* and *Electricity demand*) containing concept drifts were chosen. These are described in Section A.4 (see page 213, 210). Figures 6.5 and 6.6 show the F-Measure evolution, respectively for the (*Luxembourg*) Internet usage and electricity (*elec2*) datasets, both for the IHC and IB3 algorithms (based on 30 runs). Again, for fairness of comparison, the memory settings defined for the IHC algorithm were similar to those used by the IB3. Notice that despite IB3 being an incremental algorithm capable of handling gradual concept drifts [15], in both cases the performance of the IHC excels the performance of IB3 right away from the early phases of the learning process. Moreover, unlike the IHC algorithm the IB3 algorithm presents some degree of randomness (in particular in the early phases of learning when there are no acceptable samples), which leads to a higher variability in the results obtained. This is specially evident in the Internet usage dataset (see Figure 6.5). On the other hand, the variability of IHC is a consequence of using different memory settings. Overall, these results indicate that

IHC is more robust than IB3 excelling the latter both in the ability to handle concept drifts and in classification performance.

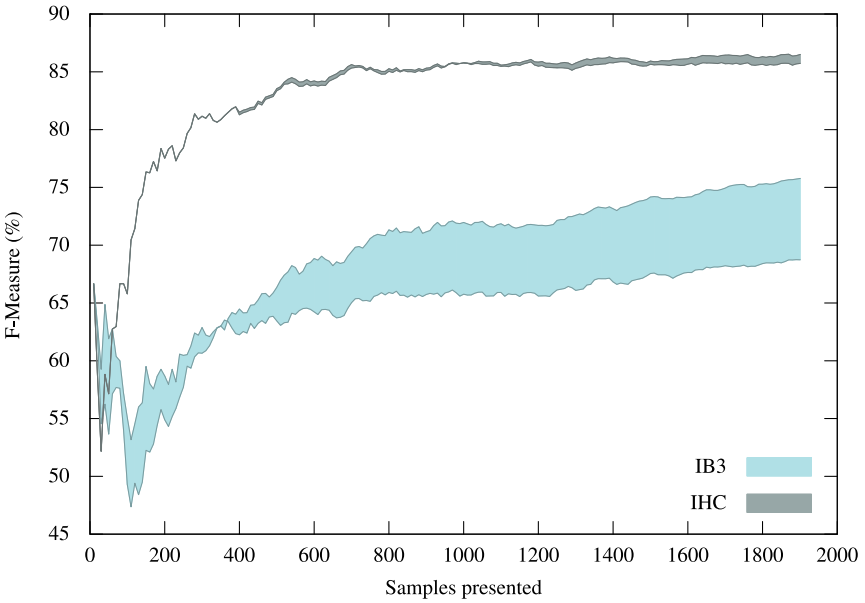


Fig. 6.5 Evolution of the F-Measure for the Internet usage dataset

6.3.3 Case Study: Protein Membership Prediction

Due to the ever increasing biological databases, a fast response for protein classification prompts the need to expedite models' adaptation. In this context, two main approaches can be considered [151]. One approach consists of using a single model that is dynamically updated as new data becomes available (incremental learning). The other is a hybrid batch-incremental approach, that relies on several models built using batch learning techniques. New models are continuously being created by using only a small subset of the data. As more reliable models become available they replace the older and obsolete ones. Some of the hybrid approaches rely on a single model while others use an ensemble of models [151].

Concerning the protein membership prediction case study, described in Appendix A.5, (see page 217), a two-step learning approach, bridging incremental and batch algorithms, is implemented. First IHC is used to select a reduced dataset (and also for immediate prediction); second the SVM algorithm is applied to the resulting dataset in order to build the protein detection model. Figure 6.7 presents the combined learning framework, which works as follows: As new data becomes

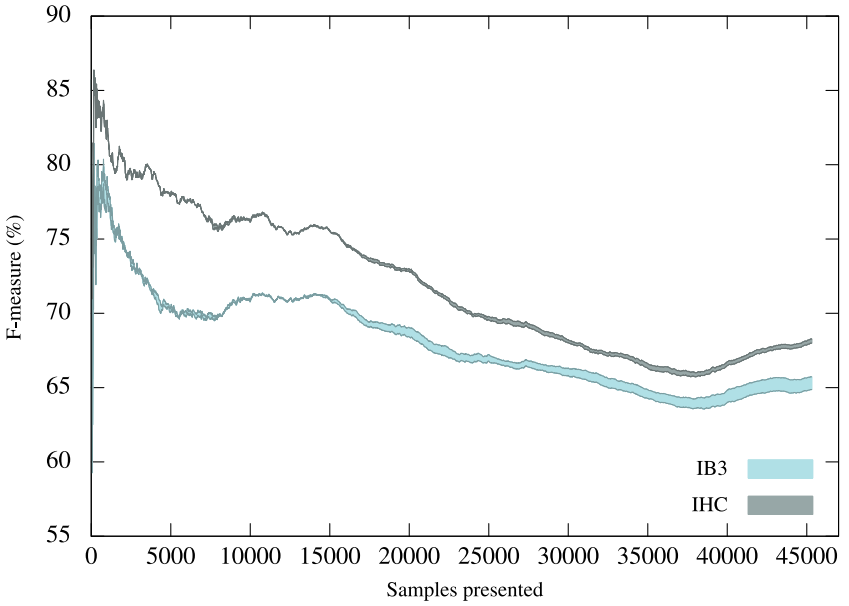


Fig. 6.6 Evolution of the F-Measure for the electricity dataset

available, the IHC is used to create a new model or to update an existing one. As mentioned before, this can be accomplished incrementally on a sample by sample basis. Thus, we can periodically check if the IHC model has changed significantly and use the samples that it encompasses to create more robust models, using state-of-the-art batch algorithms (in this case the SVM) whose application would otherwise be impractical due to processing power and memory constraints.

By combining incremental and batch algorithms in the same framework, we expect to obtain the benefits of both approaches while minimizing their disadvantages. Namely, we expect the proposed framework to be able to cope with extremely large, fast changing datasets while retaining state-of-the-art classification performance.

Figure 6.8 shows the time required to update the IHC model, after the number of samples in the memory is stabilized. Prior to that, the time required is much smaller. Note that the number of samples actually stored is inferior to n because the training dataset is strongly unbalanced. Since IHC divides the available memory by the C classes, the number of samples stored in memory for the non-peptidase class will be at most $1,806$. Thus, for $n = 20,000$ the actual number of samples stored will never exceed $11,806$. Once again, the time necessary to update the model grows linearly with the amount of memory used, demonstrating that real-time model adaptability and knowledge extraction are feasible.

To define a baseline of comparison, we started by computing the performance of the SVM algorithm. For this purpose, several kernels and parameters (using grid

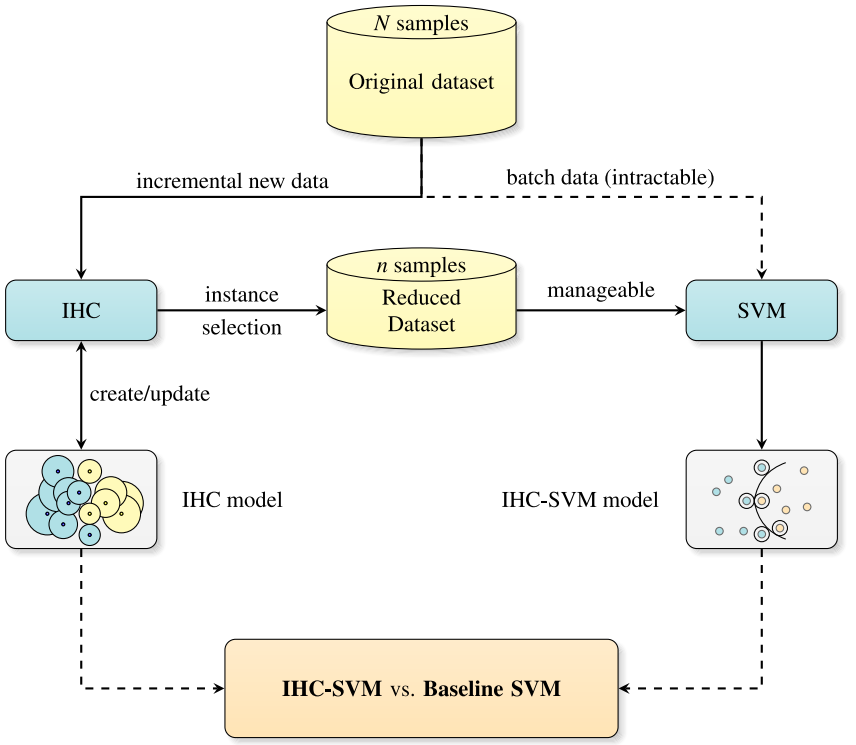


Fig. 6.7 IHC-SVM learning framework

search) were tried using 5-fold cross-validation on the training dataset in order to determine the best possible configuration. Specifically we found that the best configuration to use was a RBF (Gaussian) kernel with parameters $\gamma = 0.4$ and $C = 100$. Adopting the specified configuration, we have obtained an macro-average F-measure for the test dataset of 95.91%. The same configurations were used to train the SVMs in the proposed (IHC-SVM) approach.

We tested both the IHC and the IHC-SVM approach for the concerned problem, using parameters $g = 1$ and $g = 2$ which have demonstrated to yield good results in Lopes and Ribeiro [131]. Based on the information collected, we have set $g = 2$. Figure 6.9 shows the macro-average F-measure for both the IHC algorithm and for IHC-SVM approach, using the specified parameter.

Incremental algorithms, such as the IHC, have no access to previously discarded data and no control on the order in which the data is presented. With these constraints, we cannot expect their performance to match those of batch algorithms. Despite that, the IHC is able to achieve an F-measure of 93.73%. A working version of the IHC algorithm for peptidase detection can be found at `http://193.137.78.18/ihc/`.

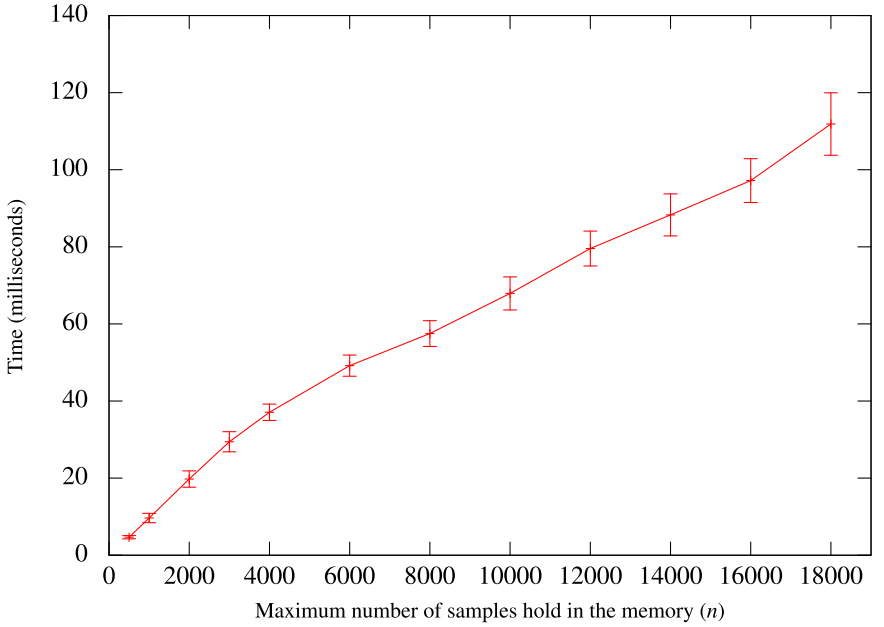


Fig. 6.8 Average time required to update the IHC model (with a new sample) for the protein membership prediction case study

Notice that when the number of samples stored (tied directly to n) is too small, the resulting models will be unable to capture the underlying distribution of the data (under-fitting), since we will only be able to store a fraction of the samples that define the decision frontier (see Figure 6.9). In such a situation the IHC algorithm performs better than the IHC-SVM approach. This might seem strange at first, however the explanation is quite simple: while the SVM algorithm only has access to the instances selected by IHC, the latter had access to the whole dataset (although in a sample by sample basis and not in the desired (optimal) order) and thus it is able to use the additional information to construct better models. As n grows, the number of stored samples gets larger and as a result, the number of forgotten samples declines. Since these are essential to reduce the damage caused by outliers and by samples with zones of influence excessively large, we will end-up with over-fitting models, concerning the IHC algorithm. However, the IHC-SVM approach is not affected in the same manner, since the SVM algorithm is able to create better models with the additional data supplied by IHC. In fact, in this situation the proposed approach (IHC-SVM) even works better than the baseline batch SVM. This provides evidence that the process used by IHC to determine the relevance of each sample, and decide which ones to retain and which ones to discard, is efficient.

Table 6.3 shows the gains of the proposed incremental-batch approach (IHC-SVM) over the baseline batch approach (SVM).

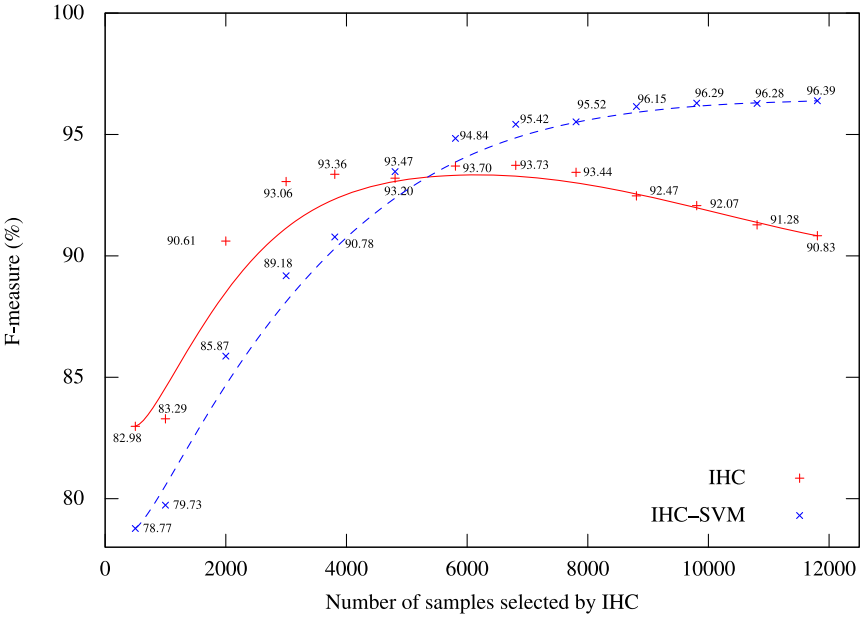


Fig. 6.9 IHC and IHC-SVM macro-average F-measure performance for the protein membership prediction case study

Note that the IHC-SVM approach is able to excel the baseline (SVM) approach using only a subset of the data. With roughly 50% of the original data (8,806 samples

Table 6.3 IHC-SVM storage reduction and classification improvement over the baseline (SVM)

Number of samples selected by IHC	Storage reduction (%)	F-Measure improvement (%)
500	97.09	17.14 ↓
1,000	94.17	16.18 ↓
2,000	88.35	10.04 ↓
3,000	82.52	6.73 ↓
3,806	77.83	5.13 ↓
4,806	72.00	2.44 ↓
5,806	66.17	1.07 ↓
6,806	60.35	0.49 ↓
7,806	54.52	0.39 ↓
8,806	48.69	**0.24** ↑
9,806	42.87	**0.37** ↑
10,806	37.04	**0.37** ↑
11,806	31.22	**0.48** ↑

out of 17,164) it is possible to create improved models. Moreover, it is possible to compact the data even further and still obtain models that match closely the performance of the baseline model.

6.4 Conclusion

This chapter presented an incremental algorithm which is able to deal with high-volumes of data, presenting uncertainty and with high-dimensionality by contrast with batch algorithms. The instance-based selection algorithm although very simple, with almost no parameters to tune, has proved to be a powerful technique specifically to deal with real-time sources of data. It deciphers in real-time in a fast manner the concept drift towards the development of amenable models without too high computational cost. Either in a standalone version or combined with SVM as pre-processing stage of a more complex problem in the field of bioinformatics, a conceivable method, simple and fast, has been provided in this chapter.

Part III
Unsupervised and Semi-supervised Learning

Chapter 7
Non-Negative Matrix Factorization (NMF)

Abstract. In this chapter we introduce the Non-Negative Matrix Factorization (NMF), which is an unsupervised algorithm that projects data into lower dimensional spaces, effectively reducing the number of features while retaining the basis information necessary to reconstruct the original data. Basically, it decomposes a matrix, containing only non-negative coefficients, into the product of two other non-negative matrices with reduced ranks. Since negative coefficients are not allowed, the original data is reconstructed through additive combinations of the parts-based factorized matrix representation. Following, we present the multiplicative and the additive GPU implementations of the NMF algorithm for the Euclidean distance as well as for the divergence cost function. In addition, a new semi-supervised approach that reduces the computational cost while improving the accuracy of NMF-based models is also presented. Finally, we present results for well-known face recognition benchmarks that demonstrate the advantages of both the proposed method and the GPU implementations.

7.1 Introduction

With the means to gather an unprecedented volume of high-dimensional data from a wide diversity of data sources, we tend to collect and store as much information as we can, thus increasing the number of features (D) simultaneously measured in each observation (sample) [234]. The rationale is that by doing so, we improve the chances of collecting data that encompasses the appropriate latent variables needed to extract valuable information. Moreover, usually there is no *a priori* information regarding the usefulness of each variable (feature), thus we are tempted to collect as many as possible [234]. Increasing the number of features places at our disposal additional data for further analysis, out of which relevant and useful information can be extracted, thus adding value to the original data. Furthermore, in general, the features present interdependencies, thus erroneous and noisy values (of certain features) can be compensated by the values of other features. However, despite these beneficial aspects, learning algorithms often present difficulties dealing with high-

N. Lopes and B. Ribeiro, *Machine Learning for Adaptive Many-Core Machines – A Practical Approach,* Studies in Big Data 7,
DOI: 10.1007/978-3-319-06938-8_7,

dimensional data [233]. The number of data samples (N) required to estimate a function of several variables grows exponentially with the number of dimensions (D) [234]. Since, in practice, the number of observations available is limited, high-dimensional spaces are inherently sparse. This fact is responsible for the so-called curse of dimensionality and is often known as the empty space phenomenon [234]. Although most real-world problems involve observations composed of several (many) variables, usually they do not suffer severely from this problem because data is located near a manifold of dimension r smaller than D [233]. Therefore, according to Verleysen, any learning task should begin by an attempt to reduce the dimensionality of the data, since this is a key issue in high-dimensional learning [233]. The rationale is to take advantage of the data redundancies in order to circumvent the curse of dimensionality problem (or at least to attenuate the problems inherent to high-dimensions) [234, 233]. In this context, the NMF algorithm can be used to reduce the data dimensionality, while preserving the information of the most relevant features in order to rebuild accurate approximations of the original data. NMF is a non-linear unsupervised technique for discovering a parts-based representation of objects [264, 111], with applications in image processing, text mining, document clustering, multimedia data, bioinformatics, micro-array data analysis, molecular pattern discovery, physics, air emission control, collaborative filtering and financial analysis among others [73, 115, 188, 264]. Essentially, it decomposes a matrix, containing only non-negative coefficients, into the product of two other matrices (also composed of non-negative coefficients): a parts-based matrix and a matrix containing the fractional combinations of the parts that approximate the original data. Since the factorized matrices are usually created with reduced ranks, NMF can be perceived as a method for reducing the number of features, while preserving the relevant information that allows for the reconstruction of the original data. Reducing the dimensionality of data poses several advantages: First, since noise is usually a random parameter, NMF cannot afford to represent it in lower dimensions of the space. Hence, noise disturbances are simply discarded, because there is no room to represent them. Moreover, redundant (highly correlated) data will be compacted for the same reason. Second, it allows for the circumvention of the curse of dimensionality and the empty space phenomenon problems [233], therefore allowing for the improvement of the accuracy of the models. Third, the computational cost associated with the problem is usually reduced [70] (since less data needs to be processed) and intractable problems may be handled. Moreover, learning methods (including NNs), often present difficulties handling high-dimensional feature vectors [233]. Thus, reducing the data dimensionality assumes particular relevance when dealing with data vectors in high-dimensional spaces and may be crucial in domains such as face recognition or text categorization where the number of available features is typically high.

7.2 NMF Algorithm

Given a matrix $\mathbf{V} \in \mathbb{R}_+^{D \times N}$ containing only non-negative coefficients ($V_{ij} \geq 0$) and a pre-specified positive integer, $0 < r < \min(D,N)$, NMF produces two matrices $\mathbf{W} \in \mathbb{R}_+^{D \times r}$ and $\mathbf{H} \in \mathbb{R}_+^{r \times N}$, also with non-negative coefficients, whose product approximates $\mathbf{V}$ (as closely as possible):

$$\mathbf{V} \approx \mathbf{WH}\,. \tag{7.1}$$

Generally, the value of r is chosen to satisfy $(D+N)r < DN$, so that the approximation, $\mathbf{WH}$, can be viewed as a compressed form of the original data [247].

Assuming that each column of $\mathbf{V}$ contains a sample with D features, then we can consider NMF as a method for extracting a new set of r features from the original data. In this case, the parts-based matrix, $\mathbf{W}$, will contain the basis vectors (one per column), which define a new set of features as an additive function of the original ones. Moreover, $\mathbf{H}$ will contain the fractional combination of basis vectors that is used to create an approximation of the original samples in $\mathbf{V}$ [115]. In other words, each column of $\mathbf{H}$ will contain a sample (mapped to a new r-dimensional space), which results from superposing the (fractional) contribution of each individual basis vector that approximates the original sample data. Figure 7.1 illustrates this idea.

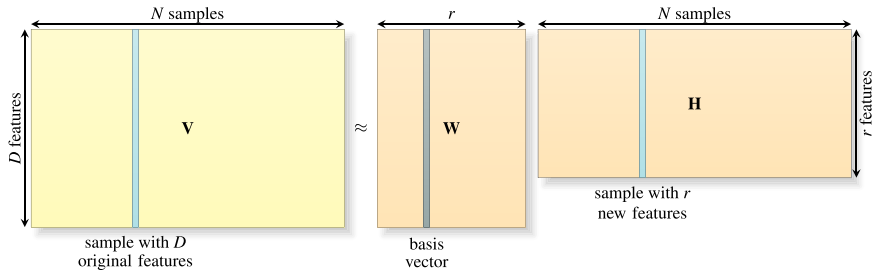


Fig. 7.1 NMF factorization

Since NMF does not allow negative information to be included in the projected spaces, cancellation effects cannot be obtained when combining data. Therefore, the encoding of the data becomes easier to interpret as compared with other methods, such as the PCA or the Independent Component Analysis (ICA) [115]. Moreover, the localized nature of the extracted features is compatible with the intuitive notion of combining parts to form a whole [264]. In other words, the original data is reconstructed through additive combinations of the parts-based factorized matrix representation. This is consistent with psychological and physiological evidence for parts-based representations in the brain [111]. For example, if each column of $\mathbf{V}$ represents a human face, then the basis elements of $\mathbf{W}$, generated by NMF, can be facial features, such as eyes, noses and lips [73].

Despite the requirement of $\mathbf{V}$ not containing negative elements, it is possible to apply NMF to any dataset, by first rescaling it between any two predefined non-negative numbers (typically between 0 and 1). Hence, we can set $\mathbf{V} = \mathbf{X}^\top$, provided that the values of $\mathbf{X}$ have been rescaled such that $\mathbf{X} \in \mathbb{R}_+^{N \times D}$.

7.2.1 Cost Functions

In order to measure the quality of the approximation defined in (7.1) it is necessary to define cost functions, by using proximity metrics, between the original matrix, $\mathbf{V}$, and the resulting approximation, $\mathbf{WH}$. Two common metrics are the Euclidean distance, given by (7.2) and the (generalized) Kullback-Leibler divergence, given by (7.3):

$$\|\mathbf{V} - \mathbf{WH}\|^2 = \sum_{ij} ((\mathbf{V})_{ij} - (\mathbf{WH})_{ij})^2 \,. \tag{7.2}$$

$$D(\mathbf{V}\|\mathbf{WH}) = \sum_{ij} \left((\mathbf{V})_{ij} \log \frac{(\mathbf{V})_{ij}}{(\mathbf{WH})_{ij}} - (\mathbf{V})_{ij} + (\mathbf{WH})_{ij} \right) . \tag{7.3}$$

Analogously to the Euclidean distance, the divergence is also lower bounded by zero and vanishes only when $\mathbf{V} = \mathbf{WH}$. However it cannot be called a "distance", since it is not symmetric in $\mathbf{V}$ and $\mathbf{WH}$. Minimizing (7.2) and (7.3) subject to the constraints $W_{ij} \geq 0$ and $H_{ij} \geq 0$ leads to two different optimization problems that can be solved using either multiplicative or additive update rules [112].

7.2.2 Multiplicative Update Rules

Considering the multiplicative rules and the Euclidean distance metric, the updates specified in (7.4) and (7.5) can be used iteratively, until a good approximation of $\mathbf{V}$ is found:

$$(\mathbf{H})_{a\mu} \leftarrow (\mathbf{H})_{a\mu} \frac{(\mathbf{W}^\top \mathbf{V})_{a\mu}}{(\mathbf{W}^\top \mathbf{WH})_{a\mu}} \,, \tag{7.4}$$

$$(\mathbf{W})_{ia} \leftarrow (\mathbf{W})_{ia} \frac{(\mathbf{VH}^\top)_{ia}}{(\mathbf{WHH}^\top)_{ia}} \,. \tag{7.5}$$

Similarly, (7.6) and (7.7) can be used for the divergence metric and the multiplicative update rules:

$$(\mathbf{H})_{a\mu} \leftarrow (\mathbf{H})_{a\mu} \frac{\sum_i (\mathbf{W})_{ia} (\mathbf{V})_{i\mu} / (\mathbf{WH})_{i\mu}}{\sum_k (\mathbf{W})_{ka}} \,, \tag{7.6}$$

$$(\mathbf{W})_{ia} \leftarrow (\mathbf{W})_{ia} \frac{\sum_\mu (\mathbf{H})_{a\mu} (\mathbf{V})_{i\mu} / (\mathbf{WH})_{i\mu}}{\sum_v (\mathbf{H})_{av}} \,. \tag{7.7}$$

7.2.3 Additive Update Rules

An alternative to the multiplicative update rules can be obtained by using the gradient descent technique. In such a case (7.8) and (7.9) can be applied iteratively, for the Euclidean distance, until a good approximation of $\mathbf{V}$ is found:

$$(\mathbf{H})_{a\mu} \leftarrow \max(0, (\mathbf{H})_{a\mu} + \eta_{au}\left[(\mathbf{W}^\top \mathbf{V})_{au} - (\mathbf{W}^\top \mathbf{W}\mathbf{H})_{au}\right]), \eta_{au} = \frac{(\mathbf{H})_{a\mu}}{(\mathbf{W}^\top \mathbf{W}\mathbf{H})_{au}}, \tag{7.8}$$

$$(\mathbf{W})_{ia} \leftarrow \max(0, (\mathbf{W})_{ia} + \gamma_{ia}\left[(\mathbf{V}\mathbf{H}^\top)_{ia} - (\mathbf{W}\mathbf{H}\mathbf{H}^\top)_{ia}\right]), \gamma_{ia} = \frac{(\mathbf{W})_{ia}}{(\mathbf{W}\mathbf{H}\mathbf{H}^\top)_{ia}}. \tag{7.9}$$

Similarly (7.10) and (7.11) can be used for the divergence:

$$(\mathbf{H})_{a\mu} \leftarrow \max(0, (\mathbf{H})_{a\mu} + \eta_{au}\left[\sum_i (\mathbf{W})_{ia}\frac{(\mathbf{V})_{i\mu}}{(\mathbf{W}\mathbf{H})_{i\mu}} - \sum_i (\mathbf{W})_{ia}\right]), \eta_{au} = \frac{(\mathbf{H})_{a\mu}}{\sum_i (\mathbf{W})_{ia}}, \tag{7.10}$$

$$(\mathbf{W})_{ia} \leftarrow \max(0, (\mathbf{W})_{ia} + \gamma_{ia}\left[\sum_j (\mathbf{H})_{aj}\frac{(\mathbf{V})_{ij}}{(\mathbf{W}\mathbf{H})_{ij}} - \sum_j (\mathbf{H})_{aj}\right]), \gamma_{ia} = \frac{(\mathbf{W})_{ia}}{\sum_j (\mathbf{H})_{aj}}. \tag{7.11}$$

7.3 Combining NMF with Other ML Algorithms

In order to combine NMF with other algorithms we use the procedure described in Ribeiro et al. [188]. Accordingly, Figure 7.2 illustrates the process to combine NMF with other ML (supervised) algorithms. First, the NMF algorithm is applied to the training dataset, with the purpose of reducing the data dimensionality, while obtaining the main discriminative characteristics of the data. By doing so, matrix $\mathbf{W} \in \mathbb{R}_+^{D\times r}$ will hold the r main features extracted from the original training dataset and matrix $\mathbf{H_{train}} \in \mathbb{R}_+^{r\times N}$ the codification of the parts-based characteristics (incorporated in $\mathbf{W}$) that when added will result in the appropriate approximation of the original data. The data contained in $\mathbf{H_{train}}$ (encompassing r inputs instead of the original D inputs) is then used to build a model (with the desired learning algorithm). Finally, the quality of the resulting model can be asserted by using $\mathbf{H_{test}} \in \mathbb{R}_+^{r\times N'}$, which is also computed by NMF, using the same basis features, $\mathbf{W}$, as those obtained for the training data. Hence, in this case, only the $\mathbf{H_{test}}$ matrix gets updated while the $\mathbf{W}$ matrix remains constant. Eventually, the previous steps can be repeated with different configurations, until a classifier that meets the goals expectations is found.

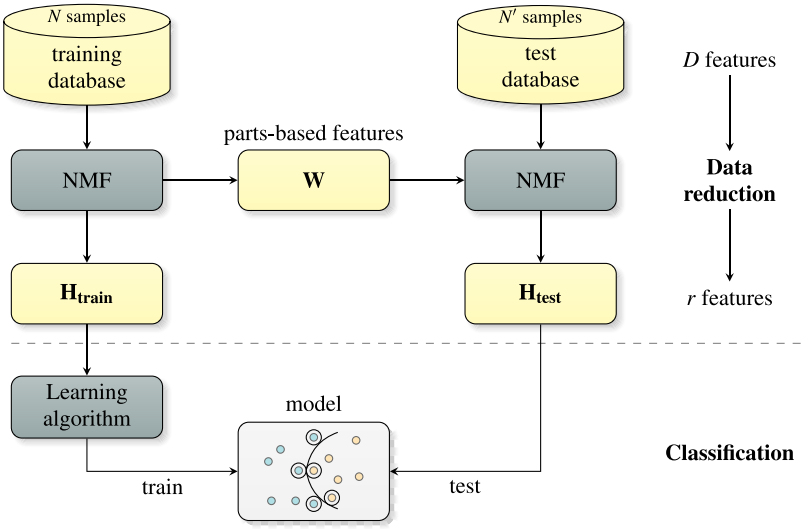


Fig. 7.2 Combining NMF with other learning algorithms

Note that each time new data is gathered to be used by the resulting classifier, a brand-new **H** matrix (containing the codification of the parts-based characteristics that approximate the new data) needs to be computed, using the aforementioned process. Although the parts-based matrix **W** remains invariant, computing **H** is still a time consuming process that can prevent this method from being used in real-world applications. In this context, the GPU parallel implementation, presented later in Section 7.5, is a fundamental step towards softening this problem.

7.4 Semi-Supervised NMF (SSNMF)

Since NMF is an unsupervised algorithm, the extracted features can comprise a combination of characteristics present in objects (samples) of different classes. For some problems, this is not desirable and characteristics of different classes should not be intermixed. In particular for classification tasks, we are interested in the most unique and discriminating characteristics of each class. For example, in the face recognition domain, although some individuals may look alike it is desirable to extract their unique and peculiar characteristics rather than the ones that reflect similar aspects of different individuals. However, by directly applying NMF to the training data, the extracted features, **W**, will most likely be shared by objects of all the classes. To overcome this problem, we propose to partition the matrix, **V**, containing the original inputs (features) of the training data into sub-matrices, $\mathbf{V_1}, \mathbf{V_2}, \ldots, \mathbf{V_C}$, each containing the samples' inputs of one of the C different classes.

The NMF algorithm is then independently applied to each one of the sub-matrices, using a smaller number of basis vectors $r_1, r_2, \ldots, r_C$ such that $r = \sum_{i=1}^{C} r_i$. This results in the creation of C parts-based feature matrices, $\mathbf{W_1}, \mathbf{W_2}, \ldots, \mathbf{W_C}$, and C codification matrices, $\mathbf{H_1}, \mathbf{H_2}, \ldots, \mathbf{H_C}$, which are then combined to create a global $\mathbf{W}$ matrix and a global $\mathbf{H_{train}}$ matrix. Figure 7.3 represents this process, where the white areas of the $\mathbf{H_{train}}$ matrix correspond to zero value elements [132].

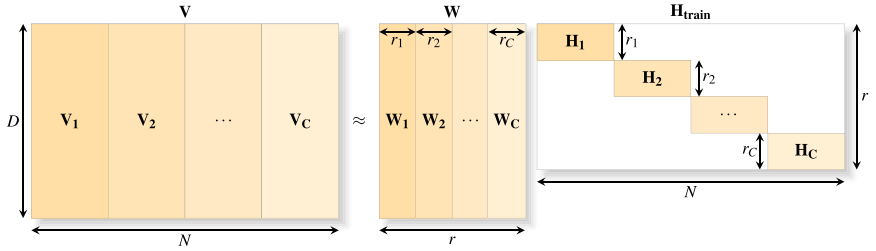


Fig. 7.3 Generation and combination of the individual class matrices. The white areas of the $\mathbf{H_{train}}$ matrix correspond to zero value elements.

Although the resulting method, designated by SSNMF, does not prevent similar characteristics to arise independently for the different classes, it increases the probability of extracting unique class features. This assumes particular relevance for unbalanced datasets, where the particular and representative characteristics of the objects belonging to minority classes may be perceived as noise, whenever NMF is applied directly to all the data. Figure 7.4 shows the typical basis vectors generated by the NMF and SSNMF methods for the Yale face database (described in Appendix A.4, page 215). Note that as expected, it is easier to recognize the individuals from the base features generated by the SSNMF approach (compare the resulting basis vectors with the original faces in Figure A.8, page 216). Finally, it is worth mentioning that the SSNMF method applies only to the computation of $\mathbf{W}$ and $\mathbf{H_{train}}$. The $\mathbf{H_{test}}$ matrix is calculated in the same manner (unsupervised) as described in Section 7.3 [132].

For many real-world problems, obtaining labeled data is a relatively expensive process [41]. Hence, in some situations we may have at our disposal a relatively small number of labeled samples, while having an enormous amount of unlabeled samples. Although, only the data samples for which the class is known can be used by a supervised algorithm to create a classifier model, we can still use all the data (labeled and unlabeled) for the data reduction phase (see Figure 7.2), in order to extract characteristics that will in principle reflect better the actual data distribution. In this case we could still apply the SSNMF algorithm, using the labeled data in order to extract unique class characteristics that could be then fine-tuned by applying NMF to all the data, while using the $\mathbf{W}$ matrix computed by SSNMF as a starting point. Note that it is also possible to use the $\mathbf{H}$ matrix computed by SSNMF as a starting point, if we add additional columns (equal to the number of unlabeled

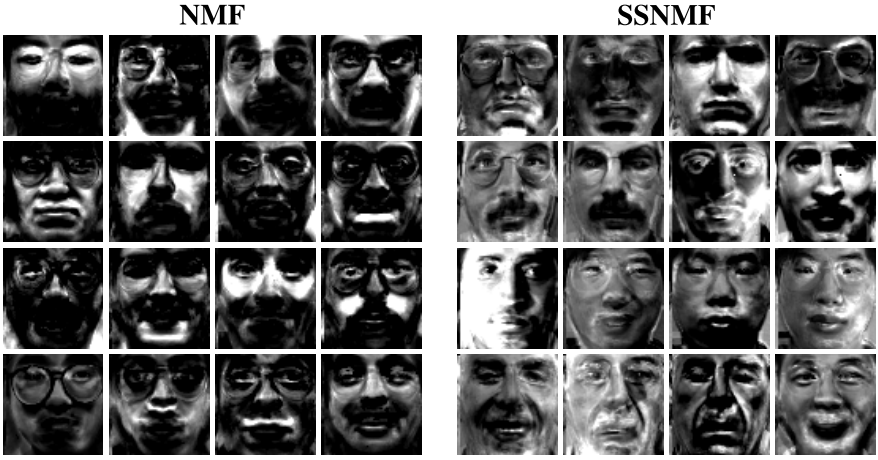


Fig. 7.4 Typical basis vectors (**W** columns) generated by NMF and SSNMF for the Yale face database (using $r_i = 3$ and $r = 45$ (3×15))

samples) to it. To prevent the original (SSNMF) extracted characteristics from being completely overwritten, matrix **W** can remain fixed during a predefined number of iterations or until there is no significant gain in updating **H** alone.

7.5 GPU Parallel Implementation

Our GPU implementation, features both the multiplicative and additive versions of NMF and supports the Euclidean distance and the Kullback-Leibler divergence metrics [136, 126].

7.5.1 Euclidean Distance Implementation

The implementation of the NMF algorithm for the Euclidean distance, relies mainly on matrix multiplications, regardless of the update rules chosen (multiplicative or additive). Hence, we rely on the functionalities provided by the class `DeviceMatrix`, described earlier in Section 2.5 (page 31). Notice however that we avoid the computation of the transpose matrices (see (7.4), (7.5), (7.8) and (7.9)), by changing the order in which the matrices are stored, from column-major to row-major (or vice-versa). This procedure, which reduces significantly the amount of memory and processing required, is depicted in Figure 7.5.

Furthermore, we realize that the order in which we multiply the matrices affects the performance of the resulting implementation (e.g. calculating $\mathbf{W}(\mathbf{H}\mathbf{H}^\top)$ is faster

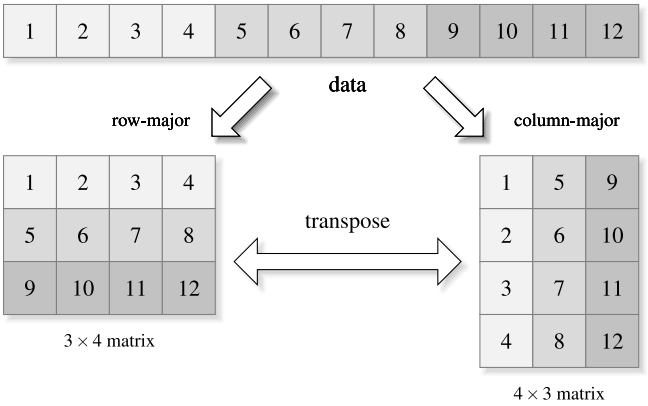


Fig. 7.5 Interpretation of the same data, using either row-major or column-major orders

Listing 7.1 CUDA kernel used to implement the NMF algorithm for the multiplicative update rules, considering the Euclidean distance.

```
__global__ void UpdateMatrix_ME(
    cudafloat * A,
    cudafloat * B,
    cudafloat * X,
    int elements
) {
    int idx = blockIdx.x * blockDim.x + threadIdx.x;
    if (idx < elements) {
        X[idx] *= A[idx] / (B[idx] +
            SMALL_VALUE_TO_ADD_DENOMINATOR);
    }
}
```

than calculating $(\mathbf{WH})\mathbf{H}^\top$). Naturally this was taken into consideration and is reflected in the resulting implementations.

Considering the multiplicative update rule implementation, an additional kernel (`UpdateMatrix_ME`) is required. This kernel, presented in Listing 7.1, updates each element of a given matrix $\mathbf{X}$ according to (7.12):

$$X_{ij} \leftarrow X_{ij} \frac{A_{ij}}{B_{ij}} , \tag{7.12}$$

where $\mathbf{A}$ and $\mathbf{B}$ are matrices of the same size as $\mathbf{X}$ (see (7.4) and (7.5)).

Listing 7.2 shows the code that implements one iteration of the multiplicative update rule for the Euclidean distance, in which the order of matrix multiplications was optimized for maximizing the computational performance.

Listing 7.2 NMF iteration code for the multiplicative update rules, considering the Euclidean distance.

```
void NMF_MultiplicativeEuclidianDistance::DoIteration(bool
    updateW) {
    DetermineQualityImprovement(true);

    // Calculate W^T
    W.ReplaceByTranspose();
    DeviceMatrix<cudafloat> & Wt = W;

    // Calculate W^T V
    DeviceMatrix<cudafloat>::Multiply(Wt, V, WtV);

    // Calculate W^T W
    Wt.MultiplyBySelfTranspose(WtW);

    // Calculate W^T WH
    DeviceMatrix<cudafloat>::Multiply(WtW, H, WtWH);

    // Update H
    UpdateMatrix_ME<<<blocksH, SIZE_BLOCKS_NMF>>>
     (WtV.Pointer(), WtWH.Pointer(), H.Pointer(), H.Elements
         ());

    Wt.ReplaceByTranspose();

    if (!updateW) return;

    // Calculate H^T
    H.ReplaceByTranspose();
    DeviceMatrix<cudafloat> & Ht = H;

    // Calculate VH^T
    DeviceMatrix<cudafloat>::Multiply(V, Ht, VHt);

    // Calculate HH^T
    DeviceMatrix<cudafloat> & HHt = WtW;
    Ht.ReplaceByTranspose();
    H.MultiplyBySelfTranspose(HHt);

    // Calculate WHH^T
    DeviceMatrix<cudafloat>::Multiply(W, HHt, WHHt);

    // Update W
    UpdateMatrix_ME<<<blocksW, SIZE_BLOCKS_NMF>>>
     (VHt.Pointer(), WHHt.Pointer(), W.Pointer(), W.Elements
         ());
}
```

Listing 7.3 CUDA kernel used to implement the NMF algorithm for the additive update rules, considering the Euclidean distance.

```
__global__ void UpdateMatrix_AE(
    cudafloat * X,
    cudafloat * A,
    cudafloat * B,
    int elements
) {
    int idx = blockIdx.x * blockDim.x + threadIdx.x;
    if (idx < elements) {
        cudafloat v = X[idx] + (X[idx] / B[idx]) * (A[idx] -
            B[idx]);
        if (v < CUDA_VALUE(0.0)) v = CUDA_VALUE(0.0);
        X[idx] = v;
    }
}
```

Similarly to the multiplicative rules, the additive update rules also require an additional kernel (`UpdateMatrix_AE`), reproduced in Listing 7.3, which updates all the elements of a given matrix $\mathbf{X}$ according to (7.13):

$$X_{ij} \leftarrow \max(0, X_{ij} + \frac{X_{ij}}{B_{ij}}(A_{ij} - B_{ij})) \,, \tag{7.13}$$

where once again $\mathbf{A}$ and $\mathbf{B}$ are matrices of the same size as $\mathbf{X}$ (see (7.8) and (7.9)).

7.5.2 Kullback-Leibler Divergence Implementation

Unlike the Euclidean distance, the divergence update rules do not depend so heavily on matrix multiplications. Thus, in the case of the multiplicative rules, four kernels are required (`SumW`, `SumH`, `UpdateH_MD` and `UpdateW_MD`).

The `SumW` kernel calculates $\sum_k W_{ka}$ for each column a of $\mathbf{W}$ and puts the result in a vector of dimension r, see (7.6). Similarly, `SumH` calculates $\sum_v H_{av}$ for each row a of $\mathbf{H}$, placing the result in a vector also with dimension r, see (7.7). Listing 7.4 presents the code of the `SumW` kernel, which uses the reduction process, described earlier in Section 2.5 (page 32). Note that for simplification, the code presented here, assumes that $\mathbf{W}$ is stored in column-major order. However, the GPUMLib code available for download can be configured to support both row-major and column-major orders.

The kernels `UpdateH_MD` and `UpdateW_MD` respectively update all the elements of $\mathbf{H}$ and $\mathbf{W}$. Both kernels work in a similar manner, thus we will focus on the inner-working of the `UpdateH_MD` kernel.

Listing 7.4 One of the CUDA kernels (SumW) used to implement the NMF algorithm for the multiplicative update rules, considering the Kullback-Leibler divergence.

```
template <int blockSize>
__global__ void SumW(cudafloat * W, int d, cudafloat * sumW) {
    extern __shared__ cudafloat w[];

    w[threadIdx.x] = CUDA_VALUE(0.0);
    for(int k = threadIdx.x; k < d; k += blockSize) {
        w[threadIdx.x] += W[d * blockIdx.x + k];
    }
    __syncthreads();

    if (blockSize >= 1024) {
        if (threadIdx.x < 512) w[threadIdx.x] += w[threadIdx.x +
            512];
        __syncthreads();
    }

    if (blockSize >= 512) {
        if (threadIdx.x < 256) w[threadIdx.x] += w[threadIdx.x +
            256];
        __syncthreads();
    }

    if (blockSize >= 256) {
        if (threadIdx.x < 128) w[threadIdx.x] += w[threadIdx.x +
            128];
        __syncthreads();
    }

    if (blockSize >= 128) {
        if (threadIdx.x < 64) w[threadIdx.x] += w[threadIdx.x + 64];
        __syncthreads();
    }

    if (threadIdx.x < 32) {
        volatile cudafloat * _w = w;

        if (blockSize >= 64) _w[threadIdx.x] += _w[threadIdx.x + 32];
        if (blockSize >= 32) _w[threadIdx.x] += _w[threadIdx.x + 16];
        if (blockSize >= 16) _w[threadIdx.x] += _w[threadIdx.x + 8];
        if (blockSize >= 8) _w[threadIdx.x] += _w[threadIdx.x + 4];
        if (blockSize >= 4) _w[threadIdx.x] += _w[threadIdx.x + 2];
        if (blockSize >= 2) _w[threadIdx.x] += _w[threadIdx.x + 1];

        if (threadIdx.x == 0) {
            cudafloat sum = w[0];
            if (sum < SMALL_VALUE_TO_ADD_DENOMINATOR) {
                sum = SMALL_VALUE_TO_ADD_DENOMINATOR;
            }
            sumW[blockIdx.x] = sum;
        }
    }
}
```

In order to update a given element $H_{a\mu}$, we need to access all the elements in the column a of $\mathbf{W}$ and all elements in the column μ of both $\mathbf{V}$ and $\mathbf{WH}$, as shown in Figure 7.6. Hence, the CUDA thread assigned to update a given matrix element $H_{a\mu}$ needs to access the same elements of $\mathbf{V}$ and $\mathbf{WH}$ than the threads assigned to process the elements $H_{i\mu}$ ($i \neq a$). Similarly it needs to access the same elements of $\mathbf{W}$ as those required by the threads processing the elements H_{aj} ($j \neq \mu$).

The rationale behind organizing the threads into blocks is to share as much information as possible among the threads within a block. This substantially improves the kernel performance, since (as we said before) accessing the shared memory is significantly faster than accessing the global device memory. Given the amount of shared memory available per block in our devices (see Table A.2, page 202), we found that we were able to store at least 32×32 pieces of the matrices $\mathbf{W}$ and $(\mathbf{V})_{ij}/(\mathbf{WH})_{ij}$. Thus, ideally our kernel should be executed in blocks of $32 \times 32 = 1024$ threads. However, the devices available at the time (a GeForce 8600 GT and a GeForce GTX 280), supported a maximum of 512 threads per block (see Tables A.2 (page 202) and 2.1 (page 22)). To solve this problem and create a kernel that is able to run on any device, while maximizing the amount of information shared, each block contains $32 \times 16 = 512$ (`blockDim.x` = 32, `blockDim.y` = 16) threads. However, each thread gathers two elements of $\mathbf{W}$, $\mathbf{V}$ and $\mathbf{WH}$ instead of one, and updates two elements of $\mathbf{H}$ (observe Figure 7.6). Therefore, although each block contains only 512 threads, 1024 elements are updated. This strategy improves the speedup gains.

The additive update rules, for the divergence, require only two kernels (`UpdateH_AD` and `UpdateW_AD`) which are similar to `UpdateH_MD`.

7.6 Results and Discussion

7.6.1 Experimental Setup

We have conducted all the NMF related experiments in the face recognition domain. Face recognition has many potential applications in various distinct areas, such as military, law-enforcement, anti-terrorism, commercial and human-computer interaction [238]. Over the past decades, face recognition has become an increasingly important area, attracting researchers from pattern recognition, neural networks, image processing, computer vision, machine learning and psychology among others [238, 260]. However, this is still a very challenging and complex problem, because the appearance of individuals is affected by numerous factors (e.g. illumination conditions, facial expressions, usage of glasses) and current systems are still no match for the human perception system [260]. A detailed survey on existing techniques and methods for face recognition can be found in Zhao et al. [260]. Typically, solving this problem involves several phases: (*i*) segmentation of the faces, (*ii*) extraction of relevant features from the face regions, (*iii*) recognition and

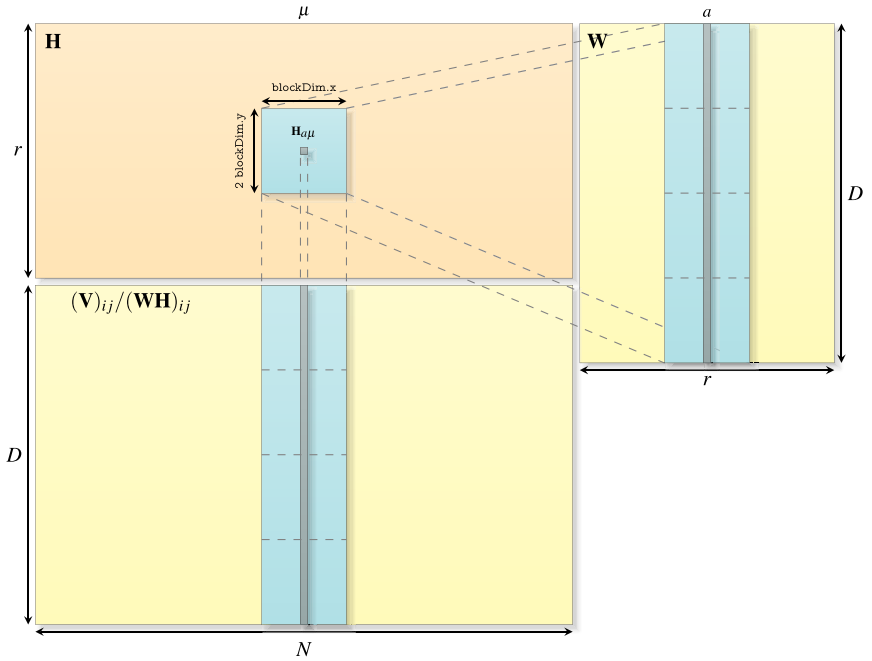


Fig. 7.6 Processing carried out, for each element $\mathbf{H}_{a\mu}$, by the `UpdateH_MD` kernel

(*iv*) verification [260]. However, in this work, we concentrate on the last phases, leaving out the segmentation phase. Accordingly, instead of relying on handcrafted features, we use the NMF algorithm to (*ii*) extract features directly from the raw images' data. These are then used to (*iii*) create and (*iv*) validate a face recognition model, using the process described in Section 7.3.

Altogether, in our testbed experiments we have used three different databases: the Center for Biological and Computational Learning (CBCL), Yale and AT&T databases. These were described in Section A.4 (see page 210, 215, 207).

The CBCL face database #1 was used specifically to test and validate the GPU parallel implementations of the NMF algorithm. The tests were conducted using the 2,429 face images of the training dataset. The matrix containing the face images was created by placing one image per column. Thus, in this case, matrix $\mathbf{V}$ is composed by 361 rows (19×19 pixels) and 2,429 columns (samples).

Aside from testing and validating the GPU parallel implementations of the NMF algorithm, the remaining two databases (the Yale and AT&T) were also used to evaluate the effectiveness of the classification method presented in Section 7.3 as well as the performance of the SSNMF method described earlier (see Section 7.4). To this end, the leave-one-out-per-class cross-validation method was used. Thus, in the case of the Yale database, the training matrix, $\mathbf{V}_{\mathbf{train}}$, is composed of 4,096 rows (64×64 pixels) and 150 columns (face images), while the test matrix, $\mathbf{V}_{\mathbf{test}}$, is

composed of $4,096$ rows and 15 columns. Similarly, in the case of the AT&T (ORL) database, $\mathbf{V_{train}}$ is composed of $10,304$ (112×92) rows and 360 columns and $\mathbf{V_{test}}$ is composed of $10,304$ rows and 40 columns.

In addition, the Yale face database was also used to further test and validate the ATS (described in Section 3.4). Accordingly, we have used the process, described in Section 7.3, to combine the NMF algorithm with the MBP algorithm. Moreover, in this particular experiment, we decided to use the hold-out validation instead of the leave-one-out-per-class cross-validation method, so that we could train more networks using the ATS. Hence, in order to build the training dataset, we randomly select 8 images of each person (corresponding to approximately $3/4$ of the database images). Consequently, the remaining 3 images per person, encompassing approximately $1/4$ of the images, were used to create the test dataset. Thus, in this case the training matrix, $\mathbf{V_{train}}$, is composed of $4,096$ rows and 120 columns, while the test matrix, $\mathbf{V_{test}}$, is composed of $4,096$ rows and 45 columns.

With the exception of the experiments conducted in order to determine the GPU implementations' speedups, the Euclidean distance implementation with the multiplicative update rules of the NMF algorithm was used, since as we shall see in the next Section this is the fastest implementation.

Before running the experiments, a histogram equalization was applied to the datasets' images, in order to reduce the influence of the surrounding illumination (see Section A.6). Moreover, all the tests were performed using the computer system 2 (see Table A.1, page 202).

7.6.2 Benchmarks Results

Concerning the CBCL face database, we have carried out several tests, in order to determine the speedups provided by the GPU implementations relative to the CPU. The tests were performed for 1,000 iterations of the algorithm, using different values of r. Figures 7.7 and 7.8 present the time required to run the NMF method using respectively the multiplicative and the additive update rules.

These results, clearly show that the GPU is able to reduce significantly the amount of time required by the NMF algorithms. Moreover, the Euclidean distance is faster than the Kullback-Leibler divergence and typically the multiplicative rules perform slightly faster than the additive rules.

In addition, the GPU parallel implementations have proven to scale better when facing larger volumes of data, due to the high number of cores present in the GPU. For example, considering the multiplicative update rules and the Euclidean cost function, when r is set to 50, the GPU needs approximately 2.5 seconds to run the iterations while the CPU requires approximately 6 minutes, which is translated into a speedup of $137.55\times$. However when r is set to 300, the GPU now requires about 12 seconds while the CPU needs over an hour, which corresponds to a speedup of $311.86\times$. This is better emphasized in Figure 7.9 which exhibits the speedups provided by the GPU over the CPU.

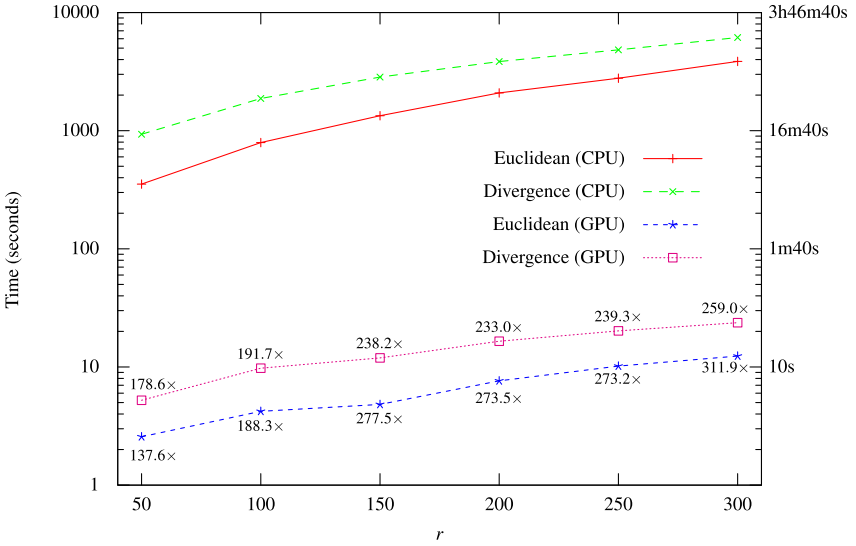


Fig. 7.7 Time required to run the NMF algorithms on the CBCL face database, during 1,000 iterations, using the multiplicative update rules. The speedups (×) provided by the GPU are shown in the respective lines.

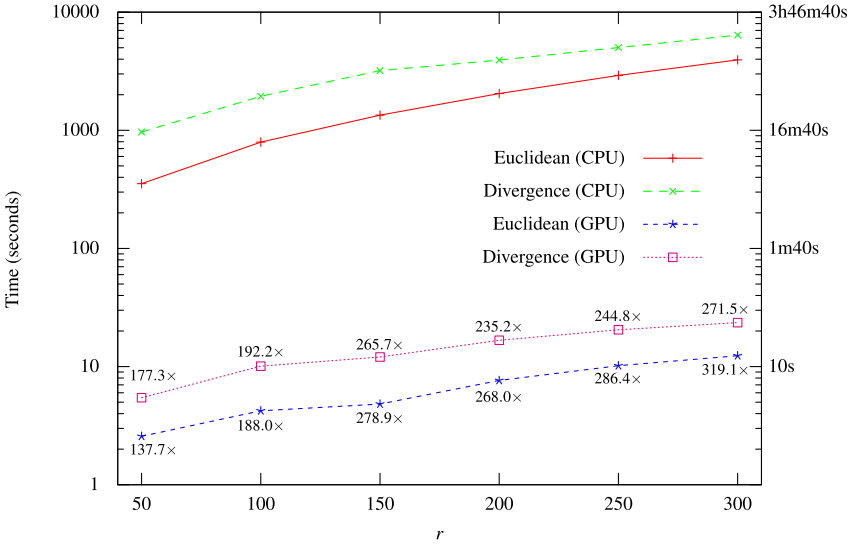


Fig. 7.8 Time required to run the NMF algorithms on the CBCL face database, during 1,000 iterations, using the additive update rules. The speedups (×) provided by the GPU are shown in the respective lines.

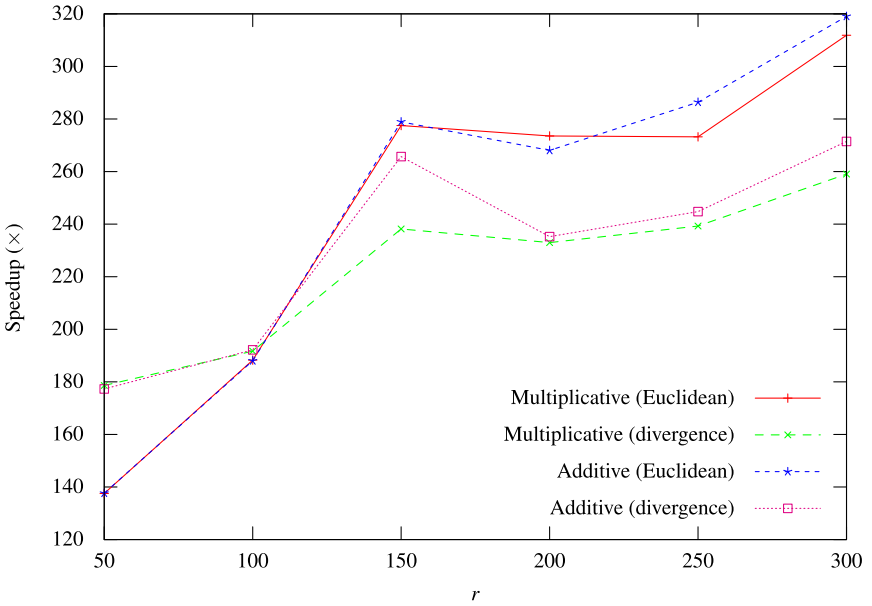


Fig. 7.9 NMF GPU Speedups for the CBCL face database

Figure 7.10 shows (a) five of the original CBCL face images, (b) the face images after applying the histogram equalization technique, (c)(d)(e)(f) the approximations generated by NMF after 1,000 iterations and (c')(d')(e')(f') some of the resulting parts. In this case, r was set to 49, which corresponds to using approximately one part image per 50 images in the original dataset. The results obtained, demonstrate that the GPU parallel implementations of the NMF algorithm are working as intended.

To further validate the GPU parallel implementation of the NMF algorithm as well as the approach (described in Section 7.3) to combine NMF with other algorithms, we have integrated this method with the MBP algorithm. Moreover, in order to train the networks, we have used the ATS (described in Section 3.4). This allowed us to perform an additional test on the capabilities of the ATS, to find the adequate topology of the networks, in a practical situation. Therefore, we started by applying the NMF algorithm to the Yale training dataset (containing 120 samples), in order to determine the parts-based matrix, $\mathbf{W}$, representation of the faces and the matrix $\mathbf{H}_{\mathbf{train}}$ that will later be used to create (train) the classifiers. The number of parts-based images (r) was chosen to be 45, so that each individual could potentially have three part-based images. In practice, because NMF is an unsupervised algorithm there is no guarantee that each individual will have three parts-based images associated or that the parts-based images will not end up being sharing by several individuals. Figure 7.11 shows the first 40 images of $\mathbf{W}$, obtained with the specified configuration.

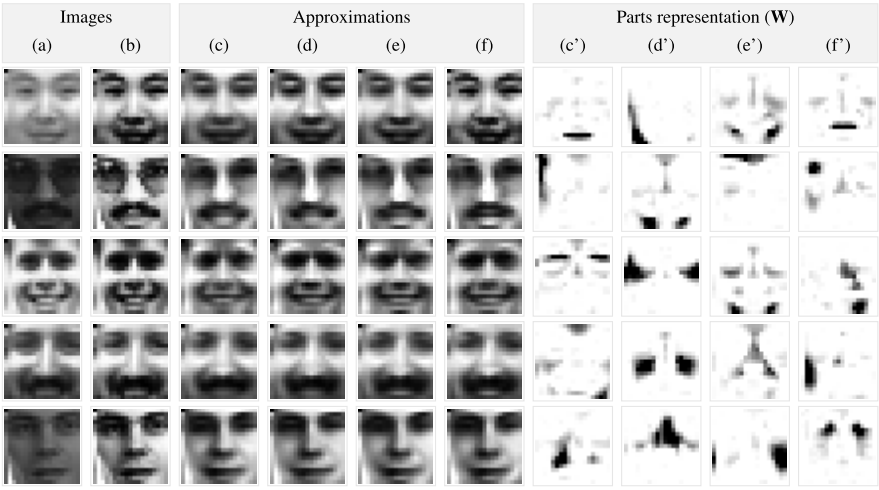


Fig. 7.10 Approximations and parts representation generated by the NMF algorithms

Fig. 7.11 Parts-based faces representations, **W**, generated by NMF for the Yale dataset

Following the computation of **W** and $\mathbf{H}_{\mathbf{train}}$, we use the NMF algorithm again, this time on the test dataset (containing 45 samples) in order to obtain the $\mathbf{H}_{\mathbf{test}}$ matrix, necessary to assert the quality of the resulting NN classifiers (see Figure 7.2).

The $\mathbf{H}_{\mathbf{train}}$ and $\mathbf{H}_{\mathbf{test}}$ matrices containing the codification of the new features extracted from the original data by NMF (in an unsupervised manner) were then used by the ATS to build a suitable classification model. To this end, the ATS actively searches for better network topology configurations in order to improve the classification models (see Section 3.4). For the problem at hand, the ATS took less than 16 hours to train a total of 100,000 networks. Figure 7.12 shows the number

of networks trained by the ATS according to number of hidden neurons. The best network (with 12 hidden neurons) presents an accuracy of 93.33% on the test dataset and of 100% on the training dataset. Only three images (of different persons) on the test dataset were misclassified and among those one had 46.11% probability of belonging to the correct individual. Thus, the results obtained demonstrate once again the potential of the ATS, which was able to find high-quality solutions without any human-intervention (aside from the initial configuration). Moreover, most of the networks trained have identical or similar topologies to the best found (over 95% of the networks trained had between 11 and 14 hidden neurons).

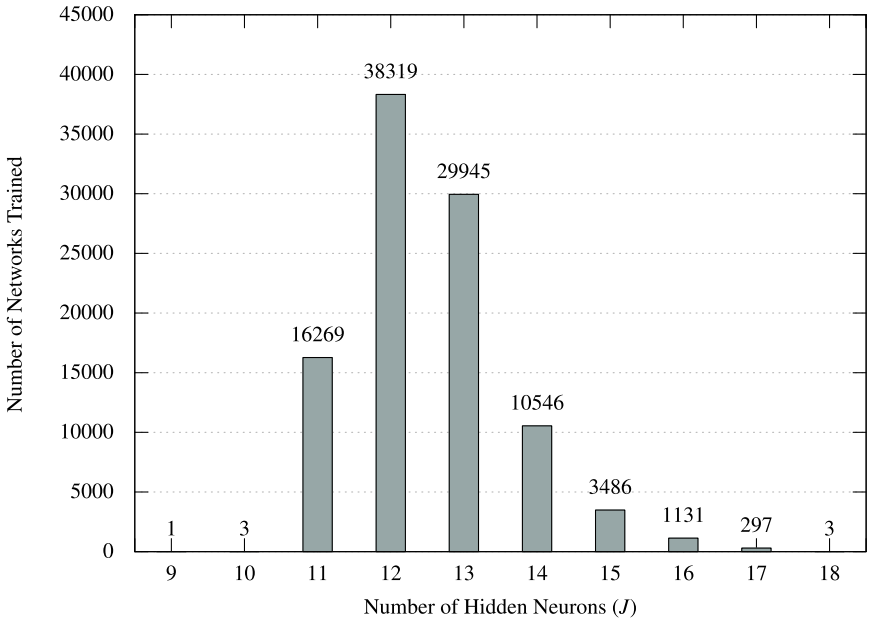


Fig. 7.12 Number of networks trained by the ATS, according to the number of neurons

In the remaining experiments, the leave-one-out-per-class cross-validation method was used. Moreover, as before, in order to build the classification models, we have chosen a value of r, so that each individual could potentially have three part-based images. Thus in the Yale dataset, r was set to 45 (3×15), while in the AT&T dataset, r was set to 120 (3×40).

Figure 7.13 shows the time required to perform 10,000 iterations of the NMF algorithm on the training and test datasets of the Yale database, depending on the hardware used. In the case of the test dataset only the $\mathbf{H}_{\mathbf{test}}$ matrix gets updated. Once again, it is evident that the GPU scales better than the CPU as the volume of data to process increases (increasing values of r correspond to bigger speedups).

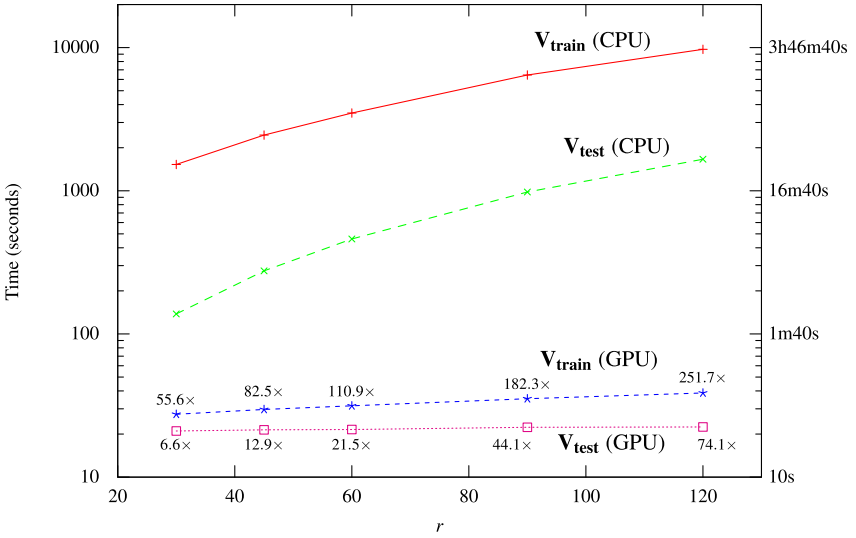


Fig. 7.13 Time to perform 10,000 NMF iterations on the Yale database. The speedups (×) provided by the GPU are shown in the respective lines.

Computing the $\mathbf{W}$ and $\mathbf{H}_{\mathbf{train}}$ matrices (for $r = 45$) with the desired accuracy, requires from 10,000 to 20,000 iterations, taking between 30 to 60 seconds on the GPU and from 40 to 80 minutes on the CPU. As for the $\mathbf{H}_{\mathbf{test}}$ matrix, roughly 10,000 iterations are required. Those can be carried out in around 20 seconds by the GPU, but take almost 5 minutes on the CPU (posing a bottleneck to the scalability of such models).

Since the matrix, $\mathbf{V}$, generated for the AT&T (ORL) database has larger dimensions, it is expected that for this problem these aspects turn out to be more relevant. Figure 7.14 shows the time required to perform 10,000 iterations of the NMF algorithm on the training and test datasets of the AT&T database for both platforms. As before, it is observed that the GPU scales better than the CPU with larger processing requirements. The GPU can yield a speedup of over 700 times (for $r = 300$) which means that each minute of processing on the GPU is roughly equivalent to 12 hours of processing on the CPU.

For the problem at hand, computing the $\mathbf{W}$ and $\mathbf{H}_{\mathbf{train}}$ matrices (for $r = 120$) with the desired accuracy, requires around 20,000 iterations. The GPU takes approximately 5 minutes to perform this task, while the CPU requires over 40 hours to do the same job. Likewise, computing $\mathbf{H}_{\mathbf{test}}$ requires roughly 10,000 iterations (for $r = 120$) which can be performed by the GPU in approximately 1 minute but would take approximately 1 hour and 25 minutes on the CPU. Clearly, the GPU accounts for an exceptional boost in the performance of the NMF algorithm and is undoubtedly connected to the success of an NMF-based classification method. Moreover, while the time required to compute the matrix on the GPU remains

mostly the same, regardless of the number of parts-based images (r), the time on the CPU escalates as the value of r increases. In fact, for $r = 300$, the CPU requires approximately 28 minutes to compute $\mathbf{H}_{\mathbf{test}}$, making the resulting model useless in many application scenarios.

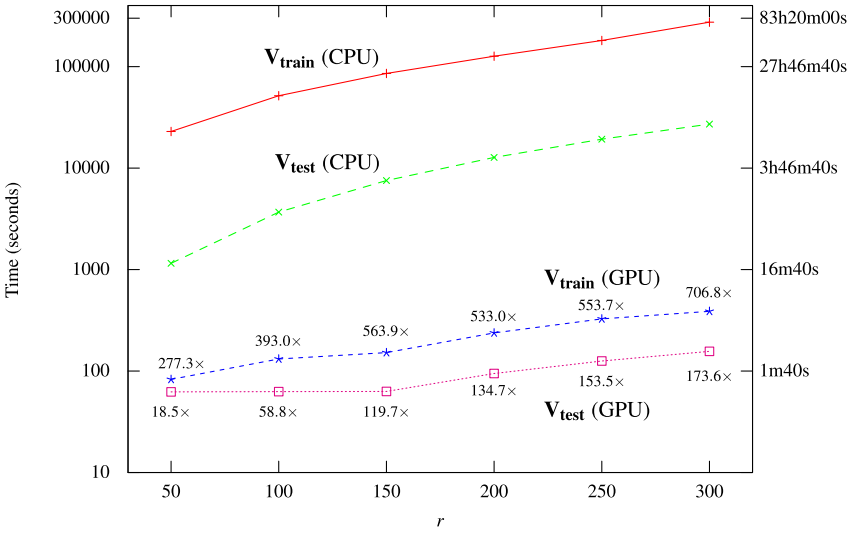


Fig. 7.14 Time to perform 10,000 NMF iterations on the AT&T (ORL) database. The speedups (×) provided by the GPU are shown in the respective lines.

After computing $\mathbf{H}_{\mathbf{train}}$ and $\mathbf{H}_{\mathbf{test}}$ with the NMF algorithm we can use the former to create a classifier and the latter to assert the quality of the resulting model. In this context, we have combined the NMF and MBP algorithms (NMF-MBP), using the ATS to create the NNs models.

For the Yale problem, we have trained 100 networks per fold and selected the one that presented the best results. The NNs had a single hidden layer, on average with 8 selective input neurons. Each network took (on average) less than 1 second to train on the GPU and we estimate that the GPU provided a speedup of 68.5 times (relatively to the CPU).

In the case of the AT&T problem, we have trained 50 networks per fold. Again, the network that presented the best results was chosen. Moreover, the NNs had a single hidden layer, on average with 11 selective input neurons. In this case, each network took on average approximately 2 minutes to train on the GPU. Furthermore, we estimate that the GPU provided a speedup of 120 times, which means that those networks would take around 4 hours to train on the CPU.

Table 7.1 exhibits the results of the NMF-MBP method as compared with other methods (Eigenface, Fisherface, Elastic Graph Matching (EGM), SVMs, NNs (BP with the Fisher projection used as the feature vector) and Face Recognition

Committee Machine (FRCM)) reported in Tang et al. [222]. The results show that our method (NMF-MBP) performs considerably better than the others for two of the image sets (left-light and right-light), demonstrating a greater robustness when dealing with different lighting conditions. Overall, on average the proposed approach excels all the others and even if we exclude the left-light and the right-light image sets (see the no light row of Table 7.1) it still yields excellent results being only surpassed by the FRCM.

Table 7.1 Accuracy (%) results for the Yale dataset

Image Set	NMF-MBP	Eigenface	Fisherface	EGM	EGM-SVM	NN	FRCM
center-light	**93.3**	53.3	**93.3**	66.7	86.7	73.3	**93.3**
glasses	**100.0**	80.0	**100.0**	53.3	86.7	86.7	**100.0**
happy	93.3	93.3	**100.0**	80.0	**100.0**	93.3	**100.0**
left-light	**60.0**	26.7	26.7	33.3	26.7	26.7	33.3
no glasses	**100.0**	**100.0**	**100.0**	80.0	**100.0**	**100.0**	**100.0**
normal	**100.0**	86.7	**100.0**	86.7	**100.0**	93.3	**100.0**
right-light	**53.3**	26.7	40.0	40.0	13.3	26.7	33.3
sad	**100.0**	86.7	93.3	93.3	**100.0**	93.3	**100.0**
sleepy	**100.0**	86.7	**100.0**	73.3	**100.0**	**100.0**	**100.0**
surprised	**93.3**	86.7	66.7	33.3	73.3	66.7	86.7
wink	93.3	**100.0**	**100.0**	66.7	93.3	93.3	**100.0**
No light	97.0	85.9	94.8	70.4	93.3	88.9	**97.8**
Average	**89.7**	75.2	83.6	64.2	80.0	77.6	86.1

The results for the AT&T database are presented in Table 7.2. In this case, three of the methods (Fisherface, SVM and FRCM) perform better than the NMF-MBP approach. Nevertheless, our method presents competitive results and it would be a valuable asset in the design of systems such as the FRCM, especially due to its robustness to different illumination conditions. The idea behind this particular committee machine (FRCM) consists of combining the results of several individual classifiers, using an ensemble of mixture of experts. The rationale is to take advantage of the specific nature of each algorithm (expert), which may present distinct performance rates for different input regions, in order to build a system with improved performance [222].

The last experiment focused on comparing the NMF with the SSNMF algorithm. To this end, we have combined those methods with the SVM algorithm.

Naturally, since the semi-supervised method works with smaller matrices it can compute and combine all the individual matrices, significantly faster than the time required for the NMF method to obtain the corresponding matrices ($\mathbf{W}$ and $\mathbf{H}_{\mathbf{train}}$). Figure 7.15 presents the time required to compute the matrices using our GPU parallel implementation of the NMF algorithm. On average SSNMF was over 6 times faster than NMF. However, the speedups yielded by the semi-supervised

Table 7.2 Accuracy (%) results for the AT&T (ORL) dataset

Image Set	NMF-MBP	Eigenface	Fisherface	EGM	EGM-SVM	NN	FRCM
1	95.0	92.5	**100.0**	90.0	95.0	92.5	95.0
2	95.0	85.0	**100.0**	72.5	**100.0**	95.0	**100.0**
3	97.5	87.5	**100.0**	85.0	**100.0**	95.0	**100.0**
4	92.5	90.0	97.5	70.0	**100.0**	92.5	**100.0**
5	97.5	85.0	**100.0**	82.5	**100.0**	95.0	**100.0**
6	95.0	87.5	**97.5**	70.0	**97.5**	92.5	**97.5**
7	92.5	82.5	95.0	75.0	95.0	95.0	**100.0**
8	92.5	92.5	95.0	80.0	**97.5**	90.0	**97.5**
9	87.5	90.0	**100.0**	72.5	97.5	90.0	**100.0**
10	87.5	85.0	**97.5**	80.0	95.0	92.5	**97.5**
Average	93.3	87.8	98.3	77.8	97.8	93.0	**98.8**

method should be greater in the CPU because the GPU scales better than the CPU when dealing with larger volumes of data (bigger matrices) [128].

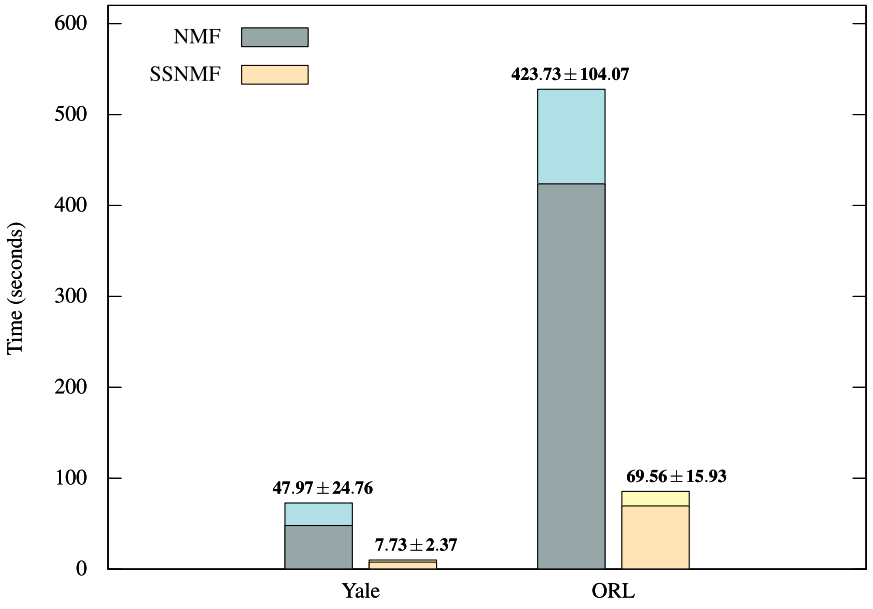


Fig. 7.15 Time required to compute the **W** and $\mathbf{H}_{\mathbf{train}}$ matrices

In terms of sparsity, it is obvious that the $\mathbf{H}_{\mathbf{train}}$ matrices generated by the SSNMF method will be sparser (by design) than the corresponding matrices produced by the unsupervised method. As for the $\mathbf{H}_{\mathbf{test}}$ matrices (generated in

the same manner regardless of the method), the matrices associated to the semi-supervised method are significantly sparser than the matrices associated to NMF (observe Table 7.3). This is important because in addition to reducing the storage memory requirements, sparsity improves the interpretation of the factors, especially when dealing with classification and clustering problems in domains like text mining and computational biology [73].

Table 7.3 Percentage of zero values present in the $\mathbf{H}_{\mathbf{test}}$ matrix

Benchmark	NMF	SSNMF
Yale	36.70 ± 8.39	63.70 ± 4.35
AT&T	45.20 ± 2.14	72.25 ± 7.48

For the SVM algorithm the RBF (Gaussian) kernel was used and the C and γ parameters were chosen by cross-validation (using grid search). Figures 7.16 and 7.17 show the average accuracy respectively for the Yale and for the AT&T datasets. Clearly, the chances of selecting a good set of parameters (C and γ) are greater for the SSNMF method. This is better quantified in Table 7.4, which presents the average accuracy of the grid search test folds. While model selection by cross-validation is a good practice, the best performance may not always be obtained on unseen data. Therefore, the SSNMF method reduces the risk of creating unfitted/inadequate models.

Table 7.4 Grid search average accuracy on the test folds

	NMF		SSNMF	
Dataset	**Mean**	**Median**	**Mean**	**Median**
Yale	82.74%	80.61%	83.34%	84.24%
AT&T	81.92%	75.00%	92.65%	93.00%

Tables 7.5 and 7.6 show the accuracy obtained by the NMF-SVM and the SSNMF-SVM approaches as compared with the NMF-MBP and the aforementioned methods reported in Tang et al. [222], respectively for the Yale and for the AT&T databases.

Considering the Yale database, the NMF based approaches (NMF-SVM, SSNMF-SVM and NMF-MBP) excel all the other methods in terms of accuracy. Overall, the best results were obtained by the NMF-MBP approach. As in the case of the NMF-MBP, the NMF-SVM and SSNMF-SVM methods perform considerably better than the others for two of the image sets (left-light and right-light), demonstrating higher robustness when dealing with different lighting conditions. This is consistent with the idea that parts-based representations can naturally deal with partial occlusion and some illumination problems and therefore are considered

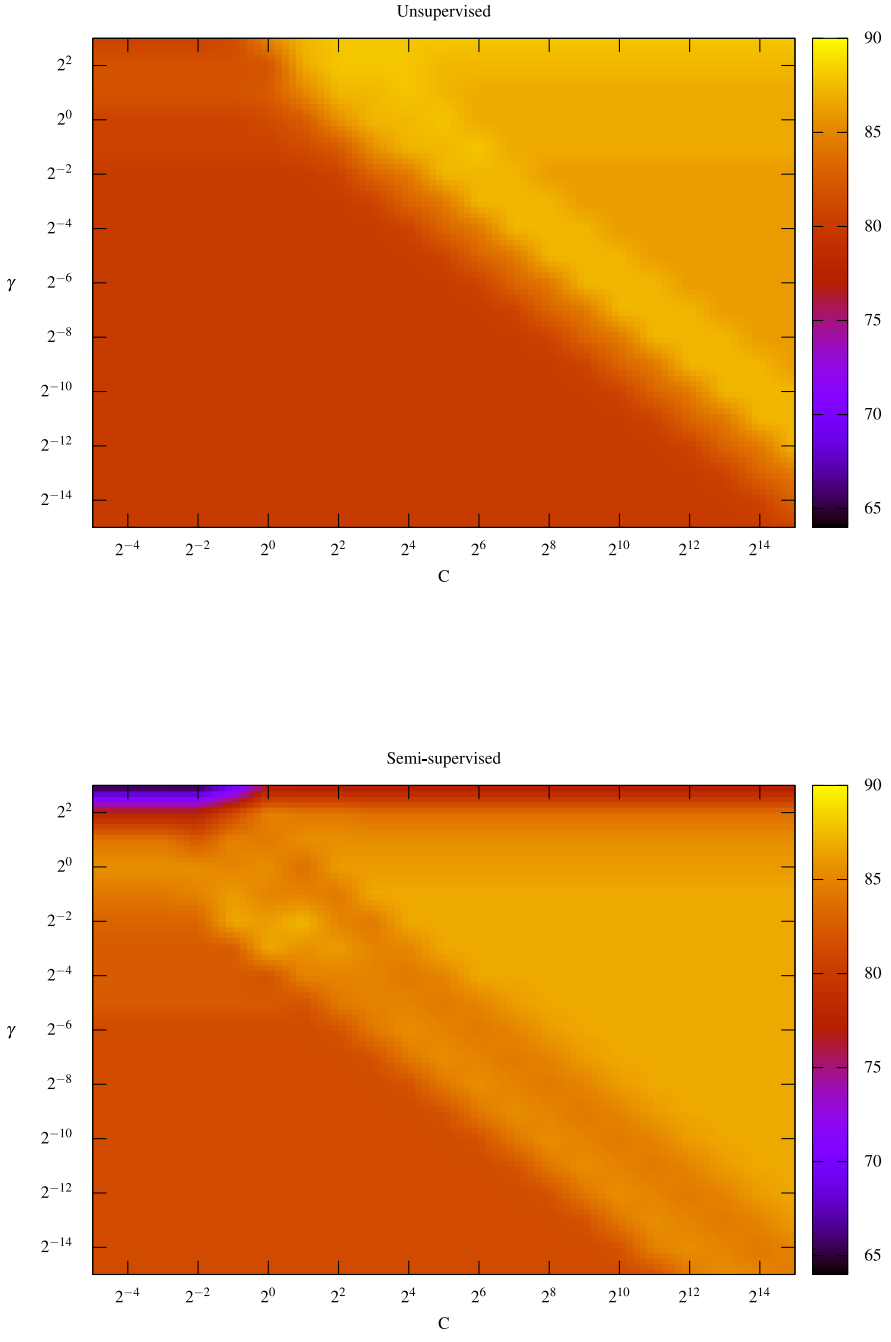


Fig. 7.16 Average accuracy yielded by the SVM algorithm for the Yale dataset

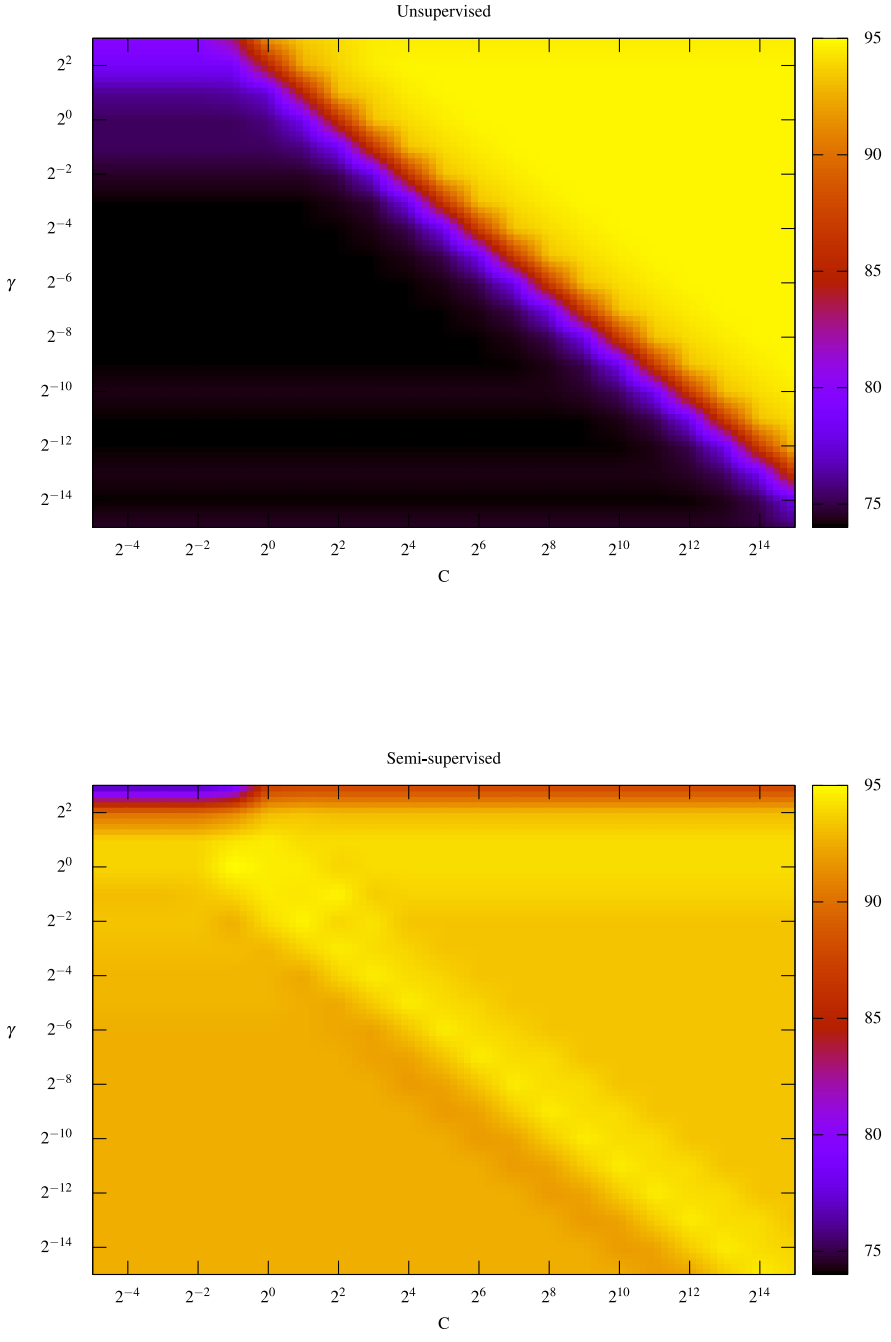


Fig. 7.17 Average accuracy yielded by the SVM algorithm for the AT&T dataset

Table 7.5 Accuracy (%) results for the Yale dataset

Image Set	NMF-SVM	SSNMF-SVM	NMF-MBP	Eigenface	Fisherface	EGM	EGM-SVM	NN	FRCM
center-light	**93.3**	**93.3**	**93.3**	53.3	**93.3**	66.7	86.7	73.3	**93.3**
glasses	**100.0**	93.3	**100.0**	80.0	**100.0**	53.3	86.7	86.7	**100.0**
happy	**100.0**	93.3	93.3	93.3	**100.0**	80.0	**100.0**	93.3	**100.0**
left-light	**60.0**	**60.0**	**60.0**	26.7	26.7	33.3	26.7	26.7	33.3
no glasses	93.3	**100.0**	**100.0**	**100.0**	**100.0**	80.0	**100.0**	**100.0**	**100.0**
normal	93.3	**100.0**	**100.0**	86.7	**100.0**	86.7	**100.0**	93.3	**100.0**
right-light	46.7	**53.3**	**53.3**	26.7	40.0	40.0	13.3	26.7	33.3
sad	**100.0**	**100.0**	**100.0**	86.7	93.3	93.3	**100.0**	93.3	**100.0**
sleepy	93.3	93.3	**100.0**	86.7	**100.0**	73.3	**100.0**	**100.0**	**100.0**
surprised	**93.3**	86.7	**93.3**	86.7	66.7	33.3	73.3	66.7	86.7
wink	93.3	86.7	93.3	**100.0**	**100.0**	66.7	93.3	93.3	**100.0**
Nolight	95.5	94.1	97.0	85.9	94.8	70.4	93.3	88.9	**97.8**
Average	87.9	87.3	**89.7**	75.2	83.6	64.2	80.0	77.6	86.1

Table 7.6 Accuracy results for the AT&T (ORL) dataset

Image Set	NMF-SVM	SSNMF-SVM	NMF-MBP	Eigenface	Fisherface	EGM	EGM-SVM	NN	FRCM
1	97.5	97.5	95.0	92.5	**100.0**	90.0	95.0	92.5	95.0
2	97.5	92.5	95.0	85.0	**100.0**	72.5	**100.0**	95.0	**100.0**
3	97.5	**100.0**	97.5	87.5	**100.0**	85.0	**100.0**	95.0	**100.0**
4	97.5	95.0	92.5	90.0	97.5	70.0	**100.0**	92.5	**100.0**
5	95.0	**100.0**	97.5	85.0	**100.0**	82.5	**100.0**	95.0	**100.0**
6	**97.5**	**97.5**	95.0	87.5	**97.5**	70.0	**97.5**	92.5	**97.5**
7	90.0	95.0	92.5	82.5	95.0	75.0	95.0	95.0	**100.0**
8	90.0	95.0	92.5	92.5	95.0	80.0	**97.5**	90.0	**97.5**
9	92.5	90.0	87.5	90.0	**100.0**	72.5	97.5	90.0	**100.0**
10	92.5	87.5	87.5	85.0	**97.5**	80.0	95.0	92.5	**97.5**
Average	94.8	95.0	93.3	87.8	98.3	77.8	97.8	93.0	**98.8**

to perform better for facial image processing [261]. Moreover, although on average the NMF-SVM performs better than the SSNMF-SVM method, the SSNMF-SVM approach presents better results on the two aforementioned (left-light and right-light) image sets.

Concerning the AT&T database, the SSNMF-SVM method outperforms the other NMF approaches (NMF-SVM and NMF-MBP). However, for this dataset there are other methods that yield higher accuracies. Still, the proposed approach demonstrates competitive results and as before, there is no doubt that it would be a valuable asset in the design of systems such as the FRCM.

7.7 Conclusion

This chapter presented the NMF algorithm, which is a non-linear unsupervised technique for discovering a parts-based representation of objects with applications in many domains ranging from image processing, text mining, bioinformatics, collaborative filtering to air emission control. We started by giving the motivation for matrix decomposition methods considering their salient characteristics in capturing the discriminating motifs in many business and industry problems. Then the algorithm was described taken into account the essential details of the decomposition of the overall matrix, containing only non-negative coefficients, into the product of two matrices whose elements are also non-negative. A wide range of studies in the above areas have shown that for each problem the original data can be decomposed into a parts-based matrix and a matrix containing the fractional combinations of the parts that approximate the data. The chapter includes some examples of an interesting problem of widely used face recognition databases where both the additive and the multiplicative updates are obtained by minimizing a cost function. A parallel GPU implementation using both the multiplicative and additive versions of NMF and based on the Euclidean distance and the Kullback-Leibler divergence metrics is thoroughly described and presented with promising results.

Chapter 8
Deep Belief Networks (DBNs)

Abstract. This chapter covers successful applications in deep learning with remarkable capability to generate sophisticated and invariant features from raw input signal data. New insights of the visual cortex and studies in the relations between the connectivity found in the brain and mechanisms for mind inference have enlightened the development of deep neural networks. In this chapter the motivation for the design of these architectures points out towards models with many layers exhibiting complex behavior enhanced by thousands of neurons in each layer. By contrast, shallow neural networks have admittedly less capability with respect to inference mechanisms since no feature detectors are found in their hidden layer. Then the chapter formalizes Restricted Boltzmann Machines (RBMs) and Deep Belief Networks (DBNs), which are generative models that along with an unsupervised greedy learning algorithm CD–k are able to attain deep learning of objects. A robust learning adaptive size method is presented. Moreover, a GPU parallel implementation yields high speedups as compared to conventional implementations. Results in benchmark data sets are presented as well as further discussion of these models.

8.1 Introduction

Recent empirical and theoretical advances in deep learning methods have led to a widespread enthusiasm in the pattern recognition and ML areas [148, 110]. Inspired by the depth structure of the brain, deep learning architectures encompass the promise of revolutionizing and widening the range of tasks performed by computers [148]. In recent months deep learning applications have been growing both in number and accuracy [148]. State-of-the-art technologies such as the Apple's Siri personal assistant or Google's Street View already integrate deep NNs into their systems, asserting their potential to increase the specter of automated systems capable of performing tasks that would otherwise require humans [148]. Moreover, just a few months ago, a team of graduate students of Geoffrey E. Hinton won the top prize in a contest aimed at finding molecules that might lead to new drugs. This

N. Lopes and B. Ribeiro, *Machine Learning for Adaptive Many-Core Machines – A Practical Approach,* Studies in Big Data 7,
DOI: 10.1007/978-3-319-06938-8_8,

was a particularly impressive achievement because never before had a deep learning architecture based-system won a similar competition and the software was designed with no prior knowledge on how the molecules bind to their targets, using only a relatively small dataset [148].

Deep architecture models reflect the results of many levels of composition of non-linear operations in their outputs [110, 192, 14]. The idea is to have feature detector units at each layer (level) that gradually extract and refine more sophisticated and invariant features from the original raw input signals. Lower layers aim at extracting simple features that are then clamped into higher layers, which in turn detect more complex features [113]. In contrast, shallow models (e.g. linear models, one hidden layer NNs, SVMs) present very few layers of composition that basically map the original input features into a problem-specific feature space [110, 251]. Figure 8.1 illustrates the main architectural differences between deep and shallow models.

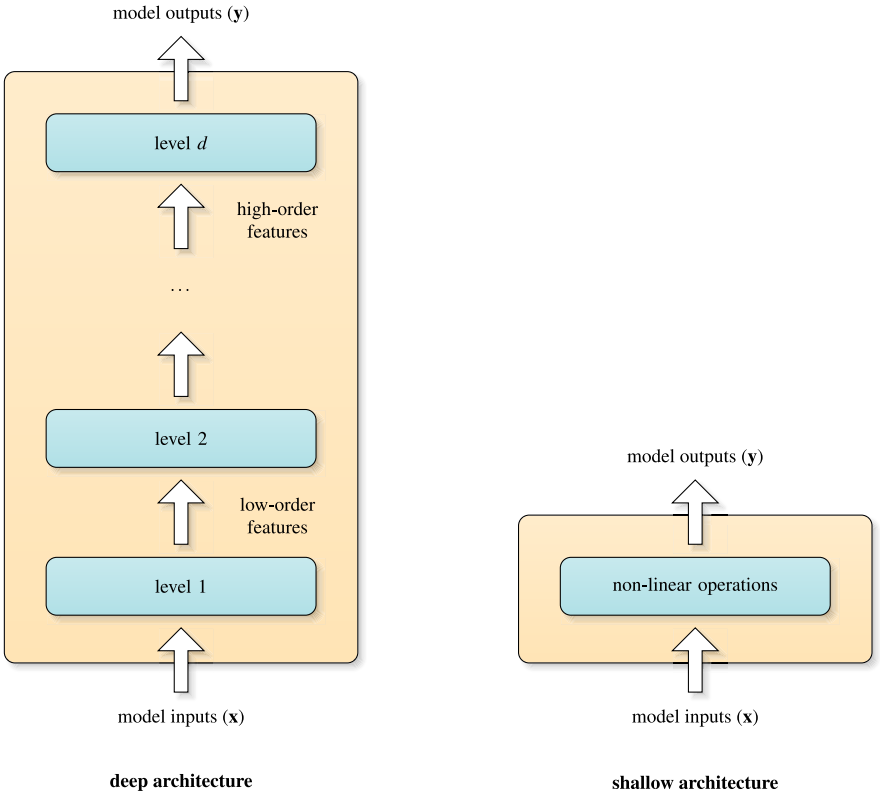


Fig. 8.1 Deep architectures versus shallow ones

Deep architectures can be exponentially more efficient than shallow ones [193]. For example, some functions can be compactly represented with an NN of depth l, while requiring an exponential number of computational elements and parameters for a network with depth $l-1$ [14]. Shallow architectures may require a huge number of elements and, consequently, of training samples to represent highly varying functions [192, 110, 14]. A function is highly varying when a large number of pieces are required in order to create its piecewise approximation [14]. On the other hand deep architectures can represent these functions efficiently, in particular when their Kolmogorov complexity is small [110].

Moreover, since each element of the architecture is learned using examples, the number of computational elements one can afford is limited by the number of training samples available [14]. Thus, the depth of architecture can be very important from the point of view of statistical efficiency and using shallow architectures may result in poor generalization models [14]. As a result, deep models tend to outperform shallow models such as SVMs [110]. Additionally, theoretical results suggest that deep architectures are fundamental to learn the kind of complex functions that can represent high-level abstractions (e.g. vision, language) [14], characterized by many factors of variation that interact in non-linear ways, making the learning process difficult [110].

However, the challenge of training deep multi-layer NNs remained elusive for a long time [14], since traditional gradient-based optimization strategies are ineffective when propagated across multiple levels of non-linearities [110]. This changed with the development of DBNs [88], which were subsequently successfully applied to several domains including classification, regression, dimensionality reduction, object segmentation, information retrieval, language processing, robotics, speech, audio, and collaborative filtering [14, 192, 110, 219, 251], thus demonstrating its ability to outperform state-of-the-art algorithms [14].

The DBNs infrastructure is supported by several layers of RBMs that are stacked on top of each other, thus forming a network that is able to capture the underlying regularities and invariances directly from the original raw data. Each RBM, within a given layer, receives the inputs of the previous layer and feeds the RBM in the next layer, thereby allowing the network as a whole to progressively extract and refine higher-level dependencies [182].

Building a DBN consists of independently training each RBM that encompasses it, starting by the lower-level layer and progressively moving up in the hierarchy, until the top layer RBM is trained.

8.2 Restricted Boltzmann Machines (RBMs)

An RBM is an energy-based generative model that consists of a layer of I binary visible units (observed variables), $\mathbf{v} = [v_1, v_2, \ldots, v_I]$ where $v_i \in \{0,1\}$, and a layer of J binary hidden units (explanatory factors), $\mathbf{h} = [h_1, h_2, \ldots, h_J]$ where $h_j \in \{0,1\}$, with bidirectional weighted connections [87], as depicted in Figure 8.2. RBMs

follow the encoder-decoder paradigm. In this paradigm an encoder transforms the input into a feature vector representation from which a decoder can reconstruct the original input [182]. In the case of RBMs both the encoded representation and the (decoded) reconstruction are stochastic by nature. The encoder-decoder architecture is appealing because: (*i*) after training, the feature vector can be computed in an expedited manner and (*ii*) by reconstructing the input we can assess how well the model captured the relevant information from the data [182].

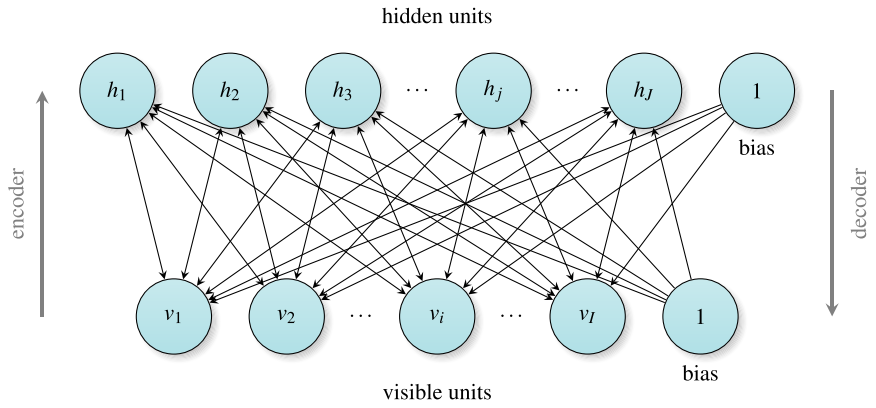


Fig. 8.2 Schematic representation of a Restricted Boltzmann Machine (RBM)

Given an observed state, the energy of the joint configuration of the visible and hidden units $(\mathbf{v},\mathbf{h})$ is given by (8.1):

$$\begin{aligned} E(\mathbf{v},\mathbf{h}) &= -\mathbf{c}\mathbf{v}^\top - \mathbf{b}\mathbf{h}^\top - \mathbf{h}\mathbf{W}\mathbf{v}^\top \\ &= -\sum_{i=1}^{I} c_i v_i - \sum_{j=1}^{J} b_j h_j - \sum_{j=1}^{J}\sum_{i=1}^{I} W_{ji} v_i h_j \,, \end{aligned} \tag{8.1}$$

where $\mathbf{W} \in \mathbb{R}^{J\times I}$ is a matrix containing the RBM connection weights, $\mathbf{c} = [c_1, c_2, \ldots, c_I] \in \mathbb{R}^I$ is the bias of the visible units and $\mathbf{b} = [b_1, b_2, \ldots, b_J] \in \mathbb{R}^J$ the bias of the hidden units. In order to break symmetry, typically the weights are initialized with small random values (e.g. between -0.01 and 0.01) [87]. The hidden bias, b_j, can be initialized with a large negative value (e.g. -4) in order to encourage sparsity and the visible units bias, c_i, to $\log(\frac{\hat{p}_i}{1-\hat{p}_i})$, where $\hat{p}_i$ is the proportion of training vectors in which $v_i = 1$ [87]. Failure to do so will require the learning procedure to adjust (in the early training stages) the probability of a given visible unit i being turned on, so that it gradually converges to $\hat{p}_i$ [87]. Figure 8.3 shows the advantages of initializing c_i in this manner, as compared to initializing the

visible bias in a random manner (between -0.01 and 0.01). Note that this simple initialization technique allows the model to capture the main characteristics of the training data and avoids unnecessary learning steps.

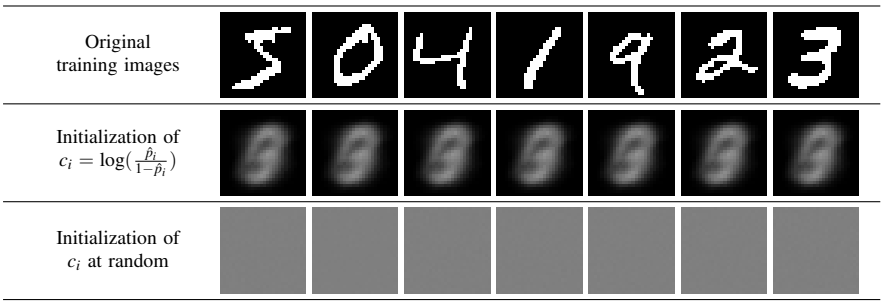


Fig. 8.3 Reconstruction of the MNIST digits made by a newly initialized Restricted Boltzmann Machine (RBM) ($\hat{p}_i$ is the proportion of training vectors in which the pixel i is on).

The RBM assigns a probability for each configuration $(\mathbf{v}, \mathbf{h})$, using (8.2):

$$p(\mathbf{v}, \mathbf{h}) = \frac{e^{-E(\mathbf{v}, \mathbf{h})}}{Z}\,, \tag{8.2}$$

where Z is a normalization constant called *partition function* by analogy with physical systems, which is obtained by summing up the energy of all possible $(\mathbf{v}, \mathbf{h})$ configurations [14, 87, 34]:

$$Z = \sum_{\mathbf{v}, \mathbf{h}} e^{-E(\mathbf{v}, \mathbf{h})}\,. \tag{8.3}$$

Since there are no connections between any two units within the same layer, given a particular random input configuration, $\mathbf{v}$, all the hidden units are independent of each other and the probability of $\mathbf{h}$ given $\mathbf{v}$ becomes:

$$p(\mathbf{h} \mid \mathbf{v}) = \prod_j p(h_j = 1 \mid \mathbf{v})\,, \tag{8.4}$$

where

$$p(h_j = 1 \mid \mathbf{v}) = \sigma(b_j + \sum_{i=1}^{I} v_i W_{ji})\,. \tag{8.5}$$

For implementation purposes, h_j is set to 1 when $p(h_j = 1 \mid \mathbf{v})$ is greater than a given random number (uniformly distributed between 0 and 1) and 0 otherwise.

Similarly given a specific hidden state, $\mathbf{h}$, the probability of $\mathbf{v}$ given $\mathbf{h}$ is obtained by (8.6):

$$p(\mathbf{v} \mid \mathbf{h}) = \prod_i p(v_i = 1 \mid \mathbf{h}) \,, \tag{8.6}$$

where:

$$p(v_i = 1 \mid \mathbf{h}) = \sigma(c_i + \sum_{j=1}^{J} h_j W_{ji}) \,. \tag{8.7}$$

When using (8.7) in order to reconstruct the input vector, it is vital to force the hidden states to be binary. Using the actual probabilities would seriously violate the information bottleneck, which acts as a strong regularizer and is imposed by forcing the hidden units to convey at most one bit of information [87].

The marginal probability assigned to a visible vector, $\mathbf{v}$, is given by (8.8):

$$p(\mathbf{v}) = \sum_{\mathbf{h}} p(\mathbf{v}, \mathbf{h}) = \frac{1}{Z} \sum_{\mathbf{h}} e^{-E(\mathbf{v},\mathbf{h})} \,. \tag{8.8}$$

Hence, given a specific training vector $\mathbf{v}$ its probability can be raised by adjusting the weights and the biases of the network in order to lower the energy of that particular vector while raising the energy of all the others. To this end, we can perform a stochastic gradient ascent on the log-likelihood manifold obtained from the training data vectors[1], by computing the derivative of the log probability with respect to the network parameters $\theta \in \{b_j, c_i, W_{ji}\}$, which is given by (8.9):

$$\begin{aligned} \frac{\partial \log p(\mathbf{v})}{\partial \theta} &= \frac{\partial \log \left(\frac{1}{Z} \sum_{\mathbf{h}} e^{-E(\mathbf{v},\mathbf{h})} \right)}{\partial \theta} = \overbrace{\frac{\partial \log \left(\sum_{\mathbf{h}} e^{-E(\mathbf{v},\mathbf{h})} \right)}{\partial \theta}}^{positive\ phase} \overbrace{- \frac{\partial \log Z}{\partial \theta}}^{negative\ phase} \\ &= \frac{\partial \log \left(\sum_{\mathbf{h}} e^{-E(\mathbf{v},\mathbf{h})} \right)}{\partial \theta} - \frac{\partial \log \left(\sum_{\mathbf{v},\mathbf{h}} e^{-E(\mathbf{v},\mathbf{h})} \right)}{\partial \theta} \\ &= - \frac{\sum_{\mathbf{h}} e^{-E(\mathbf{v},\mathbf{h})} \frac{\partial E(\mathbf{v},\mathbf{h})}{\partial \theta}}{\sum_{\mathbf{h}} e^{-E(\mathbf{v},\mathbf{h})}} + \frac{\sum_{\mathbf{v},\mathbf{h}} e^{-E(\mathbf{v},\mathbf{h})} \partial \frac{E(\mathbf{v},\mathbf{h})}{\partial \theta}}{\sum_{\mathbf{v},\mathbf{h}} e^{-E(\mathbf{v},\mathbf{h})}} \end{aligned} \tag{8.9}$$

Using (8.2) we can write $p(\mathbf{h} \mid \mathbf{v})$ as (8.10):

[1] Maximizing the logarithm of a function is equivalent to maximizing the function itself, since the logarithm is a monotonically increasing function of its argument. However, the former provides two advantages: (*i*) it simplifies the subsequent mathematical analysis and (*ii*) it helps numerically since the product of many small probabilities can cause the processor to underflow due to numerical precision issues and this is avoided when the sum of the log probabilities is used instead [18].

$$
\begin{aligned}
p(\mathbf{h} \mid \mathbf{v}) &= \frac{p(\mathbf{h}, \mathbf{v})}{p(\mathbf{v})} \\
&= \frac{p(\mathbf{h}, \mathbf{v})}{\sum_{\mathbf{h}} p(\mathbf{h}, \mathbf{v})} \\
&= \frac{1}{\sum_{\mathbf{h}} \frac{e^{-E(\mathbf{v},\mathbf{h})}}{Z}} \frac{e^{-E(\mathbf{v},\mathbf{h})}}{Z} \cdot \\
&= \frac{e^{-E(\mathbf{v},\mathbf{h})}}{\sum_{\mathbf{h}} e^{-E(\mathbf{v},\mathbf{h})}}
\end{aligned}
\tag{8.10}
$$

Hence, we can rewrite (8.9) as (8.11):

$$
\frac{\partial \log p(\mathbf{v})}{\partial \theta} = \underbrace{-\sum_{\mathbf{h}} p(\mathbf{h} \mid \mathbf{v}) \partial \frac{E(\mathbf{v}, \mathbf{h})}{\partial \theta}}_{positive\ phase} + \underbrace{\sum_{\mathbf{v},\mathbf{h}} p(\mathbf{v}, \mathbf{h}) \frac{\partial E(\mathbf{v}, \mathbf{h})}{\partial \theta}}_{negative\ phase} \tag{8.11}
$$

As in the maximum likelihood learning procedure, we aim at finding the set of network parameters for which the probability of the (observed) training dataset is maximized. Computing $\partial \frac{E(\mathbf{v},\mathbf{h})}{\partial \theta}$ is straightforward. Thus, in order to obtain an unbiased stochastic estimator of the log-likelihood gradient, we need a procedure to sample from $p(\mathbf{h} \mid \mathbf{v})$ and another to sample from $p(\mathbf{v}, \mathbf{h})$ [14]. In the so-called *positive phase*, $\mathbf{v}$ is clamped to the observed input vector, $\mathbf{x}$, and $\mathbf{h}$ is sampled from $\mathbf{v}$, while in the *negative phase*, both $\mathbf{v}$ and $\mathbf{h}$ are sampled ideally from the model [14].

Sampling can be accomplished by setting up a Markov Chain Monte Carlo (MCMC) using alternating Gibbs sampling [87, 14]. Each iteration of Gibbs sampling consists of updating all of the hidden units in parallel using (8.5) followed by updating all of the visible units in parallel using (8.7) [87]. This process is represented in Figure 8.4.

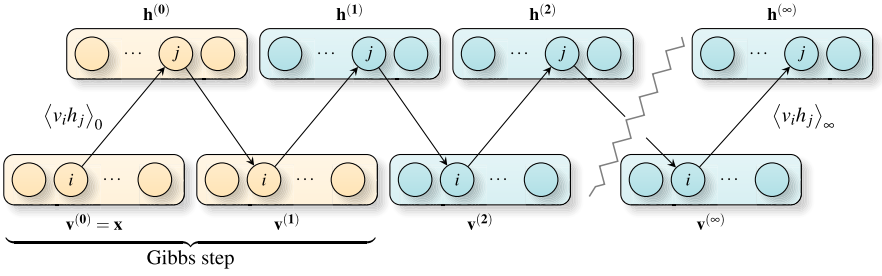


Fig. 8.4 Markov Chain Monte Carlo using alternating Gibbs sampling in a Restricted Boltzmann Machine (RBM). The chain is initialized with the data input vector, $\mathbf{x}$. The blocks in yellow correspond to a Gibbs step.

Using this procedure we can rewrite (8.11) as (8.12):

$$\frac{\partial \log p(\mathbf{v})}{\partial \theta} = \underbrace{-\left\langle \partial \frac{E(\mathbf{v},\mathbf{h})}{\partial \theta} \right\rangle_0}_{positive\ phase} + \underbrace{\left\langle \partial \frac{E(\mathbf{v},\mathbf{h})}{\partial \theta} \right\rangle_\infty}_{negative\ phase} \tag{8.12}$$

where $\langle \cdot \rangle_0$ denotes the expectations for the data distribution ($p_0 = p(\mathbf{h} \mid \mathbf{v}) = p(\mathbf{h} \mid \mathbf{x})$) and $\langle \cdot \rangle_\infty$ denotes the expectations under the model distribution ($p_\infty(\mathbf{v},\mathbf{h}) = p(\mathbf{v},\mathbf{h})$) [192, 34]. It makes sense to start the chain with a training sample, $\mathbf{x}$, because eventually the model distribution, p_∞, and the data distribution, p_0, will become similar as the model gradually captures the statistical structure embedded in the training data [14].

Unfortunately, computing $\langle v_i h_j \rangle_\infty$ is intractable as it requires performing alternating Gibbs sampling for a very long time [87, 14] in order to draw unbiased samples from the model distribution to generate a good gradient approximation [219]. This was a fundamental issue that caused the BP algorithm to replace the Boltzmann machines as the dominant learning approach for training multi-layer NNs, in the late 1980s [14].

To solve this problem, Hinton proposed a much faster learning procedure: the Contrastive Divergence (CD–k) algorithm [86, 87], whereby $\langle . \rangle_\infty$ is replaced by $\langle \cdot \rangle_k$ for small values of k [192]. This is a simple and effective alternative to the maximum likelihood algorithm that eliminates most of the computation required to obtain samples from the equilibrium distribution and significantly reduces the variance (which results from the sampling noise) that masks the gradient signal [86, 34]. Changing (8.12) accordingly, we obtain the following update rule for the weights of the network:

$$\Delta W_{ji} = \eta(\overbrace{\langle v_i h_j \rangle_0}^{positive\ phase} - \underbrace{\langle v_i h_j \rangle_k}_{negative\ phase}) \tag{8.13}$$

where η represents the learning rate. Similarly, the following rules allow us to update the bias of the network:

$$\Delta b_j = \eta(\langle h_j \rangle_0 - \langle h_j \rangle_k) \tag{8.14}$$

$$\Delta c_i = \eta(\langle v_i \rangle_0 - \langle v_i \rangle_k) \tag{8.15}$$

Algorithm 4 describes the main steps of the CD–k algorithm. Note that in the last Gibbs step, it is preferable to use the probabilities associated to the visible units (see (8.6)) for computing the state of the hidden units, instead of using the corresponding stochastic binary states which would cause unnecessary sampling noise [87].

CD–k provides a rough approximation of the log-likelihood gradient for the training data, nevertheless it has been successfully applied to many significant applications, demonstrating its ability to create good generative models from the training dataset [87]. In fact, its learning rule is actually (approximately) following

Algorithm 4 CD–*k* algorithm.

1: $\mathbf{v^{(0)}} \leftarrow \mathbf{x}$ ▷ $\mathbf{x}$ is an input vector of the training dataset.
2: Compute the binary states of the hidden units, $\mathbf{h^{(0)}}$, using $\mathbf{v^{(0)}}$ and eq. 8.5
3: **for** $n \leftarrow 1$ **to** k **do**
4: Compute the "reconstruction" states for the visible units, $\mathbf{v^{(n)}}$, using $\mathbf{h^{(n-1)}}$ and eq. 8.7
5: Compute the binary features (states) for the hidden units, $\mathbf{h^{(n)}}$, using $\mathbf{v^{(n)}}$ and eq. 8.5
6: **end for**
7: Update the weights and biases, using eq. 8.13, 8.14 and 8.15

an objective function which is called Contrastive Divergence and is given by the difference of two KullbackLeibler divergences [87, 86]. Notwithstanding the magnitude of the CD–*k* gradient estimate may not be correct, its direction tends to be accurate and there is empirical evidence that the model parameters are still moved in the same quadrant as in the log-likelihood gradient [225, 14]. In particular, CD–1 provides a low variance, fast and reasonable approximation of the log-likelihood gradient [225], which has been empirically demonstrated to yield good results [14].

Although CD–1 does not provide a very good estimate of the maximum-likelihood, this is not a issue when the features learned by the RBM will serve as inputs to another higher-level RBM [87]. In fact, for RBMs that integrate a DBN, it is not necessarily a good idea to use another form of CD–*k* that may provide closer approximations to the maximum-likelihood, but does not ensure that the hidden features retain most of the information contained in the input data vectors [87]. This is consistent with the results obtained by Swersky et al. for the experiments performed on the MNIST dataset, where the best DBN results were obtained for CD–1 [219].

An RBM by itself is limited in what it can represent and its true potential emerges when several RBMs are stacked together to form a DBN [113].

8.3 Deep Belief Networks Architecture

DBNs were recently proposed by Hinton et al., along with an unsupervised greedy learning algorithm for constructing the network one layer at a time [88]. As described earlier, the subjacent idea consists of using a RBM for each layer, which is trained independently to encode the statistical dependencies of the units within the previous layer [113].

Since a DBN aims to maximize the likelihood of the training data, the training process starts by the lower-level RBM that receives the DBN inputs, and progressively moves up in the hierarchy, until finally the RBM in top layer, containing the DBN outputs, is trained. This approach represents an efficient way

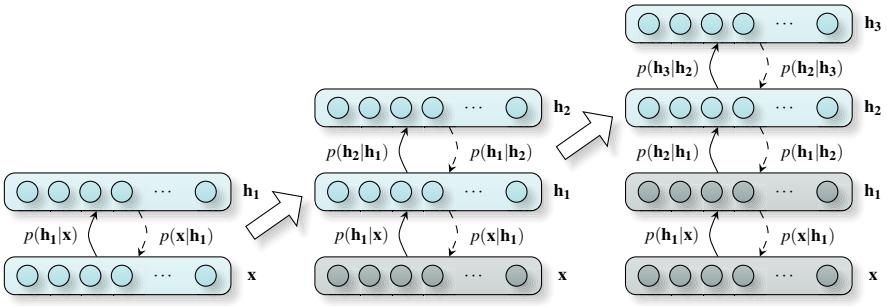


Fig. 8.5 Training process of a Deep Belief Network (DBN) with one input layer, $\mathbf{x}$, and three hidden layers $\mathbf{h_1}$, $\mathbf{h_2}$, $\mathbf{h_3}$. From left to right, layers already trained are represented by darker colors, while the Restricted Boltzmann Machine (RBM) being trained by lighter colors.

of learning (an otherwise complicated model) by combining multiple and simpler (RBM) models, learned sequentially [88]. Figure 8.5 represents this process.

The number of layers of a DBN can be increased in a greedy manner [110]. Each new layer that is stacked on top of the DBN will model the output of the previous layer [110] and aims at extracting higher-level dependencies between the original inputs variables, thereby improving the ability of the network to capture the underlying regularities in the data [182, 219]. The bottom layers are intended to extract low-level features from the input data, while the upper layers are expected to gradually refine previously learned concepts, therefore producing more abstract concepts that explain the original input observations [192, 219, 193].

The training process, also called pre-training [110], is unsupervised by nature, allowing the system to learn non-linear complex mapping functions directly from data, without depending on human-crafted features [14]. However, the output of the top layer can easily be fed to a conventional supervised classifier [87, 182]. Alternatively, it is also possible to create a classification model, by adding an additional layer to the unsupervised pre-trained DBN upon which the resulting network is fine-tuned using the BP algorithm. In this scenario the resulting network is also called a DBN [219]. Moreover, it has been shown that the BP algorithm will barely change the weights learned in the greedy stage and therefore most of the performance gains are actually obtained during the unsupervised pre-training phase [219].

8.4 Adaptive Step Size Technique

The proper choice of the learning parameters is a fundamental aspect of the training procedure that affects considerably the networks convergence [135]. In particular the learning rate is highly correlated with the training speed and convergence [202].

However, finding an adequate set of parameters is not always an easy task and usually involves a trial and error methodology, thus increasing the time and effort associated with the already expensive process of creating an effective model.

In order to mitigate this problem and improve the convergence of the networks, we adapted the adaptive step size technique, described earlier in Section 3.1 (see page 45), to the RBM networks. Hence, at each CD–k iteration, the step sizes are adjusted according to the sign changes:

$$\eta_{ji} = \begin{cases} u\eta_{ji}^{(\text{old})} & \text{if } (\langle v_i h_j \rangle_0 - \langle v_i h_j \rangle_k)(\langle v_i h_j \rangle_0^{(\text{old})} - \langle v_i h_j \rangle_k^{(\text{old})}) > 0 \\ d\eta_{ji}^{(\text{old})} & \text{if } (\langle v_i h_j \rangle_0 - \langle v_i h_j \rangle_k)(\langle v_i h_j \rangle_0^{(\text{old})} - \langle v_i h_j \rangle_k^{(\text{old})}) < 0 \end{cases} \tag{8.16}$$

where, as before, $u > 1$ (up) represents the increment factor for the step size and $d < 1$ (down) the decrement factor. When two consecutive updates have the same direction the step size of that particular weight is increased. For updates with opposite directions the step size is decreased, thus avoiding oscillations in the learning process due to excessive learning rates [135]. The underlying idea of this procedure consists of finding near-optimal step sizes that would allow bypassing ravines on the error surface. This technique is especially effective for ravines that are parallel (or almost parallel) to some axis [5].

In addition, it makes sense to use a different momentum term for each connection, $\alpha_{ji} = \eta_{ji}\alpha$, proportional to a global momentum configuration, α, and to the step sizes, in order to decrease further the oscillations in the training process. According to our tests, it is advantageous to clamp α_{ji}, such that $0.1 \leq \alpha_{ji} \leq 0.9$.

8.5 GPU Parallel Implementation

The RBM weights are not updated after each sample is presented, but rather in a batch or mini-batch process. Hence, we shall assume that the visible units vectors, $\{\mathbf{v_1}, \mathbf{v_2}, \ldots, \mathbf{v_N}\}$, form a matrix $\mathbf{V} \in \mathbb{R}^{N \times I}$, where each row contains a visible units vector, $\mathbf{v_i}$. Similarly, we shall assume that the hidden units vectors, $\{\mathbf{h_1}, \mathbf{h_2}, \ldots, \mathbf{h_N}\}$, form a matrix $\mathbf{H} \in \mathbb{R}^{N \times J}$, where each row contains a hidden units vector, $\mathbf{h_i}$. Hence, for the bottom most RBM within a DBN, $\mathbf{V}$ will be equal to $\mathbf{X}$ and I equal to D (assuming that $\mathbf{X}$ has been binarized).

In order to implement the Algorithm 4 (CD–k) we devised three CUDA kernels: a kernel to compute the binary states of the hidden units, named `ComputeStatusHiddenUnits`, which is used to implement steps 2 and 5; a kernel to compute the "reconstruction" states for the visible units, named `ComputeStatusVisibleUnits`, which is used to implement step 4; and finally a kernel to update the weights and biases, named `CorrectWeights`, which is used to implement step 7 of the referred algorithm. The latter also adjusts the step sizes of each connection. Figure 8.6 shows the sequence of kernel calls (per epoch) needed to implement the CD–k algorithm.

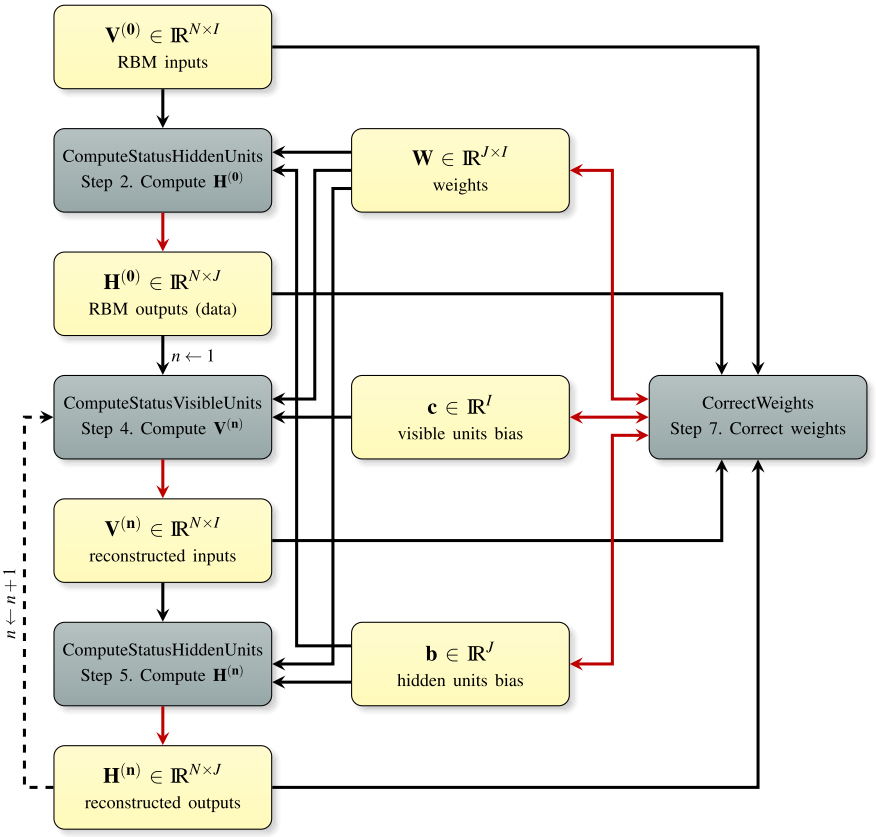


Fig. 8.6 Sequence of GPU kernel calls, per epoch, that implement the CD–*k* algorithm

As in the GPU parallel implementation of the MBP algorithm, we have opted to use a connection between two (one visible and one hidden) neurons instead of using a neuron as the smallest unit of computation (see Section 3.3, page 53). This decision, which applies both for the `ComputeStatusHiddenUnits` and `ComputeStatusVisibleUnits` kernels, allows to consider a much larger number of threads and blocks, thereby improving the scalability of the resulting kernels, allowing them to take full advantage of the GPU high number of cores. Once again, the rationale is to think of a connection as performing a simple function that multiplies the clamped input by its weight. As before, each block represents a neuron and we can take advantage of the fast shared memory to sum up the values computed by each thread, using a reduction process and then computing the output of the neuron for the active sample (defined by its position within the grid). Naturally, each block will compute the neuron output for a single specific sample. The resulting kernel grid and block structure are shown in Figure 8.7. Due to the

limits imposed for each dimension of the grid, for datasets with more than 65535 samples, the kernel must be called multiple times, processing a maximum of 65535 samples per call.

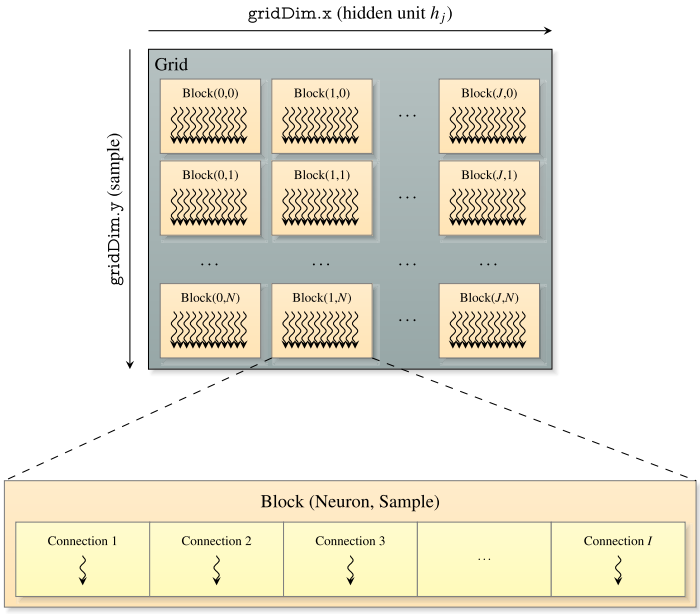


Fig. 8.7 `ComputeStatusHiddenUnits` kernel grid and block structure

In practice, the number of connections of each neuron (defined either by the number of inputs or by the number of hidden units of a RBM, respectively for the `ComputeStatusHiddenUnits` and `ComputeStatusVisibleUnits` kernels) can easily surpass the limits of the maximum number of threads per block imposed by CUDA. As a consequence each thread may actually need to compute the output of several connections [141]. Thus, the actual number of threads being executed in each call of `ComputeStatusHiddenUnits` will be equal to $\min(N,65535) \times J \times \min(I,1024)$, for devices with a compute capability of 2.x or higher. Similarly, the actual number of threads being executed in each call of `ComputeStatusVisibleUnits` will be equal to $\min(N,65535) \times I \times \min(J,1024)$. To better understand the impact of the decision of creating a thread per connection, instead of a thread per neuron, let us consider the following example: suppose that the training dataset is composed of $1,000$ images ($N = 1,000$) of 28×28 pixels ($I = 784$) and that the RBM being trained contains $1,000$ hidden units ($J = 1,000$), using our approach, both kernels (`ComputeStatusHiddenUnits` and `ComputeStatusVisibleUnits`) will execute $784,000,000$ threads. If alternatively, we had used a neuron per thread, in the same scenario, we would have at most 1 million threads.

The order in which the weights of matrix **W** are stored in the memory (row-major or column-major) affects both the `ComputeStatusHiddenUnits` and `ComputeStatusVisibleUnits` kernels. Essentially, one of the kernels will be able to access the weights in a coalesced manner, thus speeding up its execution, while the other will not. Since the kernel `ComputeStatusHiddenUnits` needs to be called more times (see Figure 8.6 and/or Algorithm 4), we decided to store **W** in a row-major order, thus improving its performance in detriment of the `ComputeStatusVisibleUnits` kernel [141]. Figure 8.8 shows the effects of this decision. When storing the weights in row-major order, the `ComputeStatusHiddenUnits` kernel will be able to access the weights in a coalesced manner, but the `ComputeStatusVisibleUnits` kernel must access them in a non-coalesced manner. On the other hand, if the weights are stored in column-major order then the `ComputeStatusVisibleUnits` kernel will be able to access them in a coalesced manner, however the `ComputeStatusHiddenUnits` must access the weights in a non-coalesced manner.

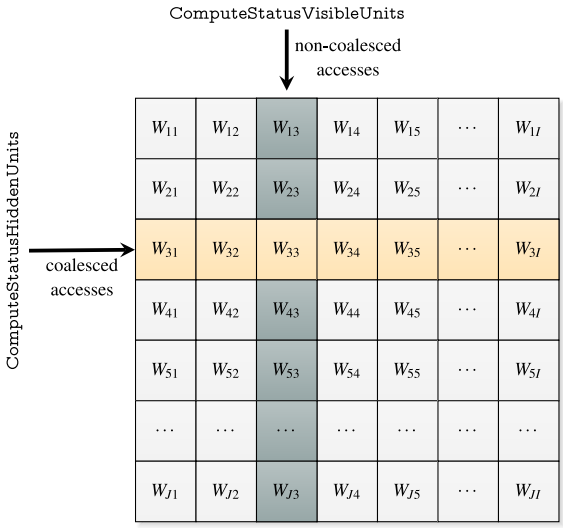


Fig. 8.8 Implications of storing the connection weights using row-major order

The bulk work to be carried out by the `CorrectWeights` kernel consists of aggregating the values for ΔW_{ji}, Δb_j and Δc_i (see respectively (8.13), (8.14) and (8.15)), needed to update the weights. Our first approach to implement this kernel consisted of creating a block for each connection, in which each thread will gather and sum the values of one or more samples, depending on the actual number of samples (N). Then a reduction process takes place in order to calculate the deltas upon which the weights and bias are updated. Figure 8.9 illustrates the resulting grid and block structure.

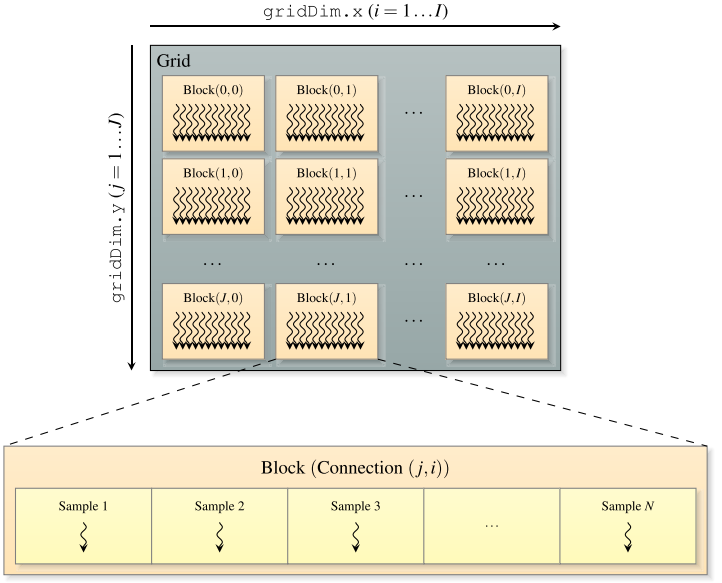


Fig. 8.9 Grid and block structure used by the first approach of the kernel `CorrectWeights`

In order to evaluate this first GPU implementation, we conducted preliminary tests using the MNIST dataset (described in Section A.4, page 213). The left column of Figure 8.10 shows the proportion of time spent in each kernel for $I = 784$, $J = 400$ and $N = 1{,}000$ (these values correspond to the worst GPU speedup, according to the test results presented later in Section 8.6). Note that despite `ComputeStatusHiddenUnits` being called twice (see Figure 8.6 and/or Algorithm 4) the overall time consumed by this kernel is still inferior to the time consumed by the `ComputeStatusVisibleUnits` kernel. This is due to the advantage of accessing the memory in a coalesced manner. Moreover, in this approach, the `CorrectWeights` kernel consumes almost $3/4$ of the total training time. Nevertheless, the preliminary tests show that overall, the GPU implementation presented speedups of one order of magnitude relative to the CPU version.

We identify two main problems in the first approach of the kernel `CorrectWeights`, both related to memory accesses to the $\mathbf{V}^{(\mathbf{0})}$, $\mathbf{H}^{(\mathbf{0})}$, $\mathbf{V}^{(\mathbf{n})}$ and $\mathbf{H}^{(\mathbf{n})}$ matrices: first the accesses were not being done in a coalesced manner and secondly many blocks were trying to access the same memory addresses, which could potentially lead to memory conflicts [141]. Figure 8.11 illustrates the latter problem: for any given hidden unit, h_j, there are $I+1$ connections, which need to access the h_j value in order to update their weights. Hence, they all need to access the same elements of $\mathbf{H}^{(\mathbf{0})}$ and $\mathbf{H}^{(\mathbf{n})}$. Similarly, for any given visible unit, v_i, there are $J+1$ connections that need to access the v_i value in order to update their weights (see Figure 8.12). Thus, they all need to access the same elements of $\mathbf{V}^{(\mathbf{0})}$ and $\mathbf{V}^{(\mathbf{n})}$.

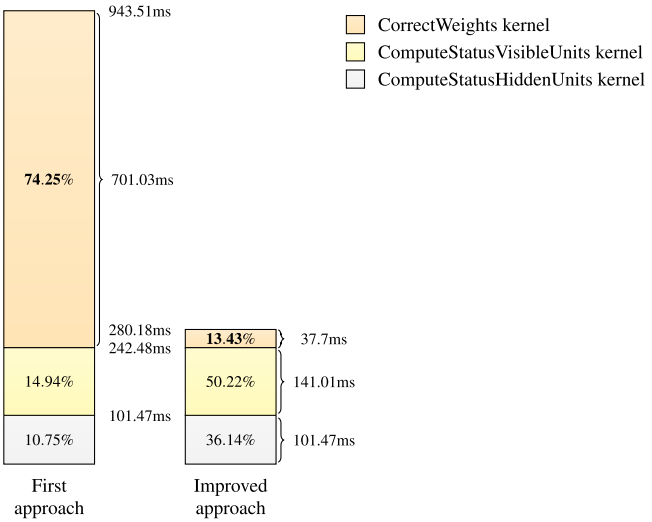


Fig. 8.10 Proportion of time spent, per epoch, in each kernel, as measured in the computer System 2 (with a GTX 280)

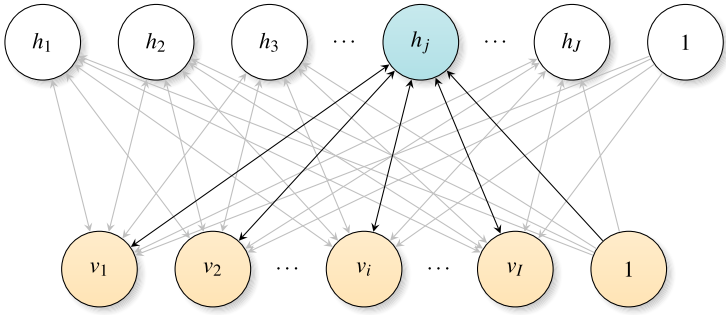


Fig. 8.11 Connections to the hidden unit j

To avoid these problems we decided to use a different approach and rewrite the referred kernel from scratch. The rationale consists of avoiding memory conflicts and uncoalesced accesses, while taking advantage of the shared memory to reduce global memory accesses. To this end, in our new and improved approach, each block processes several adjacent connections that require, to some degree, accessing the same elements of $\mathbf{V}^{(\mathbf{0})}$, $\mathbf{H}^{(\mathbf{0})}$, $\mathbf{V}^{(\mathbf{n})}$ and $\mathbf{H}^{(\mathbf{n})}$. Figure 8.13 shows the new block structure of the kernel `CorrectWeights`. The number of threads per block was defined to be $16 \times 16 = 256$, since it consistently yielded the best results among several configurations tested in the MNIST dataset, using different values of J and N. Each thread within a block must now process all the samples. For each sample,

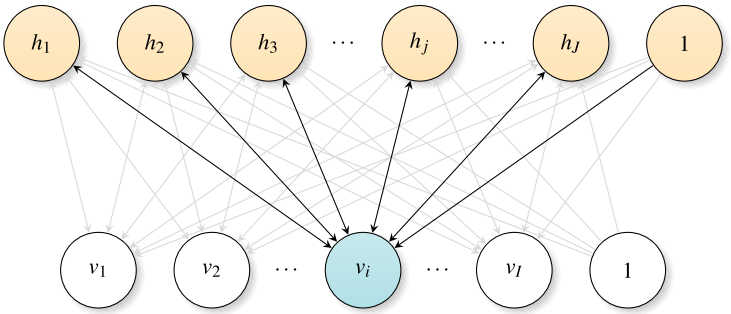


Fig. 8.12 Connections to the visible unit i

the block starts by copying the portions of $\mathbf{V}^{(\mathbf{0})}$, $\mathbf{H}^{(\mathbf{0})}$, $\mathbf{V}^{(\mathbf{n})}$ and $\mathbf{H}^{(\mathbf{n})}$, required by all the threads within the block, to the shared memory which is much faster than the global memory and can be used simultaneously by several threads within the block. Note that for threads with the same index i there will be 16 threads (each with a different j) that use the same values of $\mathbf{V}^{(\mathbf{0})}$ and $\mathbf{V}^{(\mathbf{n})}$. Similarly, for threads with the same index of j there will be 16 threads (each with a different i) that use the same values of $\mathbf{H}^{(\mathbf{0})}$ and $\mathbf{H}^{(\mathbf{n})}$. Moreover, since the required portions of the matrices $\mathbf{V}^{(\mathbf{0})}$, $\mathbf{H}^{(\mathbf{0})}$, $\mathbf{V}^{(\mathbf{n})}$ and $\mathbf{H}^{(\mathbf{n})}$ are gathered for the same sample, the global memory accesses are now coalesced.

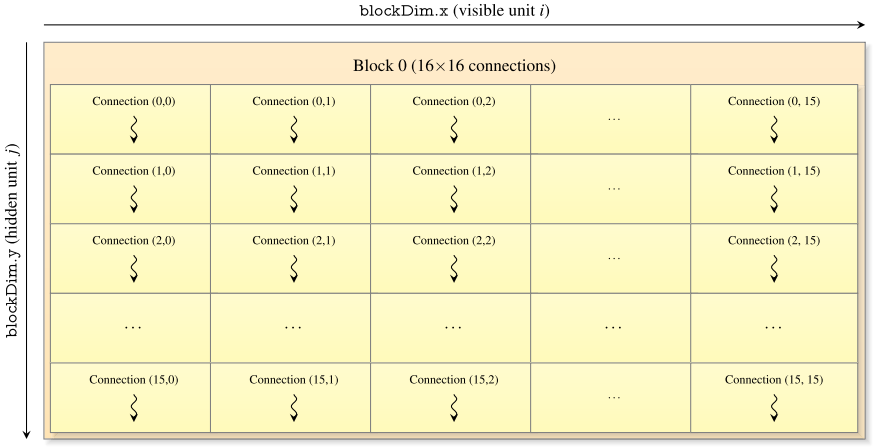


Fig. 8.13 Block structure of the improved approach of the `CorrectWeights` kernel

Although the new approach has a much smaller number of blocks and threads, due to the coalesced memory accesses and the improved use of the shared memory

it is over 18 times faster than the original one (see Figure 8.10). Note that the discrepancy is even bigger for greater values of N and J. Moreover, with this change, correcting the weights and bias is now the fastest task of the training process.

In terms of computation accuracy, the differences between the GPU and the CPU are irrelevant due to the stochastic nature of the CD–k algorithm.

8.6 Results and Discussion

8.6.1 Experimental Setup

In our testbed experiments we have used two datasets: the MNIST database of handwritten digits (see Section A.4, page 213) and the HHreco multi-stroke symbol database (see Section A.4, page 210). Altogether, three different experiments were conducted.

First, an experiment for evaluating the performance of the many-core GPU parallel implementation was carried out. Considering that both the resulting datasets have an equal number of inputs, the tests for evaluating the many-core GPU parallel implementation were carried out exclusively for the MNIST dataset. Moreover, since DBNs are composed by stacked RBMs, which are individually trained, we concentrate our efforts on testing the algorithms' performance for training RBMs. To this end, we have trained several RBMs, varying the number of training samples, N, and the number of hidden neurons, J, using both the GPU and CPU versions of the CD–k algorithm. Furthermore, the performance of the CUDA parallel implementation was benchmarked against the counterpart CPU version, using the computer system 3 (see Table A.1, page 202).

Second, we have conducted an experiment to evaluate the convergence performance of the adaptive step size method. To this end, the proposed method was compared against several typical fixed learning rate and momentum settings. In this case, the experiment was carried out using the computer system 2 (see Table A.1, page 202). Moreover, the study was also confined to the MNIST dataset.

Finally, in the last experiment, the main objective consisted of analyzing the effects of varying the number of layers and neurons of a DBN in terms of classification performance. To this end, we have trained hundreds of networks on both datasets, varying both the number of layers and the number of neurons in each hidden layer. As in the previous experiment, this study was carried out using the computer system 2 (see Table A.1, page 202). Moreover, the final training step of the DBNs was made using the GPU implementation of the BP and MBP algorithms, described earlier in Section 3.3, page 52).

8.6.2 Benchmarks Results

With the purpose of comparing the RBM GPU implementation with the corresponding CPU implementation, we have varied the number of hidden units, J, and the number of training samples, N. For statistical significance we have performed 30 tests per configuration. Figure 8.14 presents the average time required to train a RBM for one epoch, as well as the GPU speedups, depending on the hardware and according to the two aforementioned factors.

The GPU speedups obtained range from approximately 22 to 46 times. With such speedups we can transform a day of work into an hour or less and an hour of work into two or three minutes of work. It is noteworthy to say that for $N = 60,000$ and $J = 800$ the CPU version takes over 40 minutes to train a single epoch, while the GPU version takes approximately 53 seconds [141]. Moreover, there seems to be a direct correlation between the speedup and the number of samples. This was anticipated since, as we said before, the GPU scales better than the CPU when facing large-volumes of data that can be processed in parallel, due to its high-number of cores. Although not so pronounced, we can observe a similar trend correlating the speedup and the number of hidden units.

In order to evaluate the impact of the adaptive step size technique, we have compared it with three different fixed learning rate settings ($\eta = 0.1$, $\eta = 0.4$ and $\eta = 0.7$), while using three distinct momentum terms ($\alpha = 0.1$, $\alpha = 0.4$ and $\alpha = 0.7$). For the adaptive step size technique we have set the initial step sizes to 0.1. Moreover, the increment, u, and decrement, d, factors were set respectively to 1.2 and 0.8. Altogether, twelve configuration settings (three adaptive step size and nine fixed learning rate) were used. For statistical significance, we conducted 30 tests per configuration, using a RBM with 784 inputs and 100 outputs. Each test starts with a different set of weights, but for fairness all the configurations use the same weight settings, according to the test being performed. Due to the high number of tests, we decided to limit the size of the training dataset to 1,000 samples. Hence, only the first 1,000 samples of the MNIST database were used to train the networks.

Figure 8.15 shows the evolution of the RMSE of the reconstruction, according to the learning rate, η, and momentum, α, settings. Independently of the learning rate used, the best results were obtained for a momentum $\alpha = 0.1$, while the worst solutions were obtained for $\alpha = 0.7$. As expected the adaptive step size technique excels all the fixed learning rate configurations, regardless of the momentum term used. The discrepancy is quite significant (2.51%, 9.39% and 14.20% relative to the best fixed learning rate solution, respectively for $\alpha = 0.1$, $\alpha = 0.4$ and $\alpha = 0.7$) and demonstrates the usefulness of the proposed technique and its robustness to an inadequate choice of the momentum [135]. Moreover, in order to achieve better results than those obtained after 1,000 epochs, using a fixed learning rate, we would only require 207, 68 and 48 epochs, respectively for $\alpha = 0.1$, $\alpha = 0.4$ and $\alpha = 0.7$ [135]. Figure 8.16 shows the quality of the reconstruction of the original images in the database, for both the best network trained with a fixed learning rate ($\eta = 0.4$, $\alpha = 0.1$) and the best network trained with the step size technique. Furthermore,

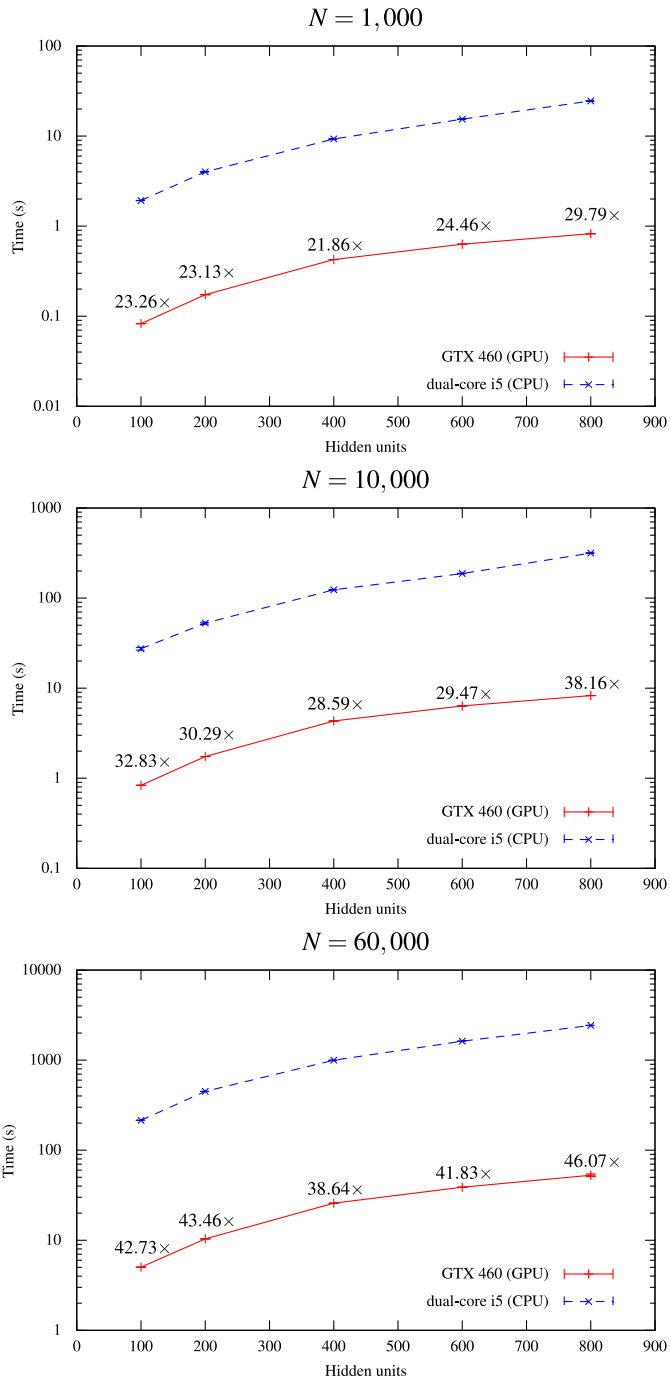


Fig. 8.14 MNIST average training time per epoch (GPU speedups are indicated)

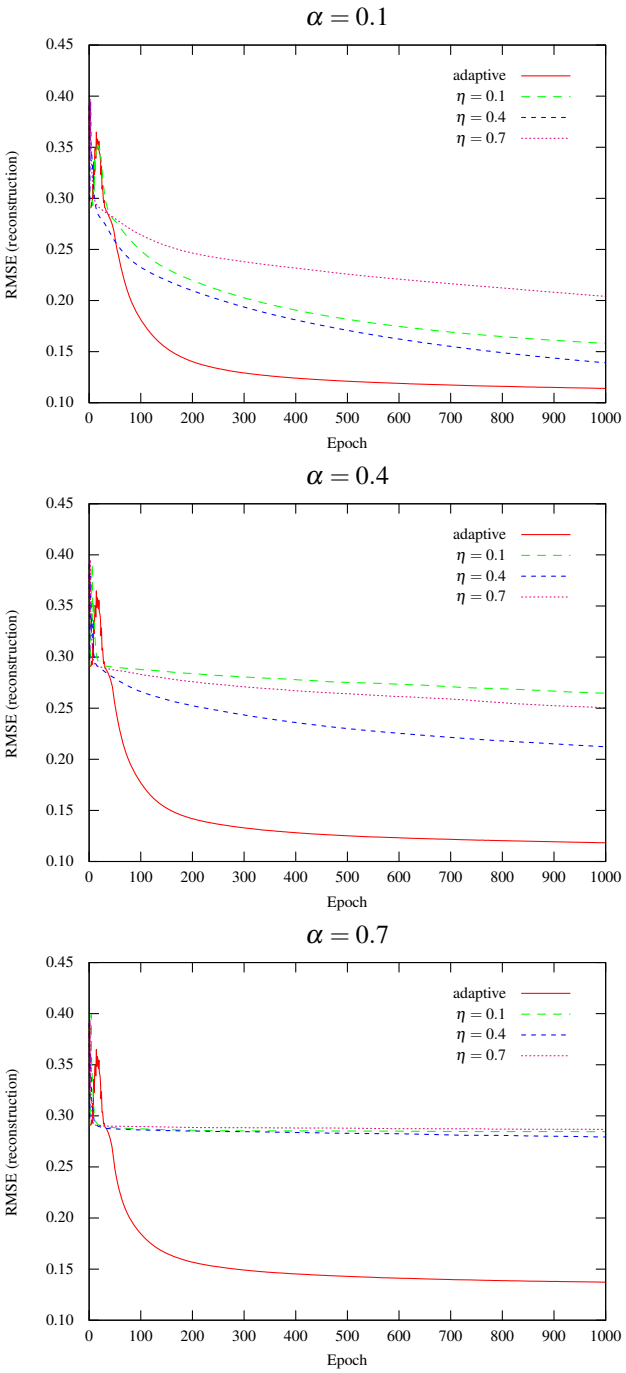


Fig. 8.15 Average reconstruction error (RMSE) according to the learning parameters, η and α

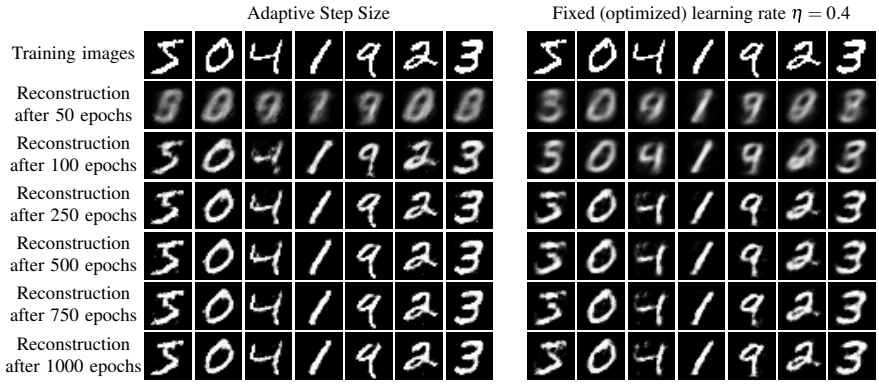


Fig. 8.16 Impact of the step size technique on the convergence of a RBM ($\alpha = 0.1$)

Figure 8.17 shows the receptive fields of the aforementioned networks and Figure 8.18 their excitatory and inhibitory response zones.

Training a network for 1,000 epochs using the adaptive step size took on average 76.63 ± 0.09 seconds, while training the same network with a fixed learning rate took on average 76.12 ± 0.05 seconds. Thus, the overhead of this method is not significant, while the convergence of the network is considerably enhanced. Additionally, the adaptive step size technique solves the difficulty of searching and choosing an adequate learning rate, η, and momentum, α terms. Moreover, the step size method can easily recover from a bad choice of the initial learning rate [5] and the parameters u and d are easily tuned (the values chosen for this problem will most likely yield good results for other problems) [135].

In order to analyze the effects of varying the number of layers and neurons, we have pre-trained several three-layer DBNs using combinations of 100, 500 and 1,000 neurons in each layer. Hence, for each dataset (MNIST and HHreco) a total of $3^3 = 27$ DBNs were trained. Since the DBNs have a modular architecture, i.e. they are built by stacking RBMs on top of each other, we have also included the networks obtained by considering only the first and the first two hidden layers. Thus, we end up with a total of $27 \times 3 = 81$ networks per dataset.

Furthermore, we have decided to test not only the "traditional" approach of adding an additional layer to the unsupervised pre-trained DBN, but also to test the effects of adding two-layers (one hidden layer with 30 neurons and one output layer). Moreover, we have also tested the effects of adding MBP layers (see Section 3.2), instead of the standard BP ones.

Overall, for each one of the original pre-trained DBNs four different classifier models were constructed. Thus, a total of $81 \times 4 = 324$ networks were trained for each dataset. Given the large number of networks to be trained and since that, as we said before, the BP algorithm hardly changes the weights learned in the greedy stage, we have decided to freeze the weights of the pre-trained networks, changing

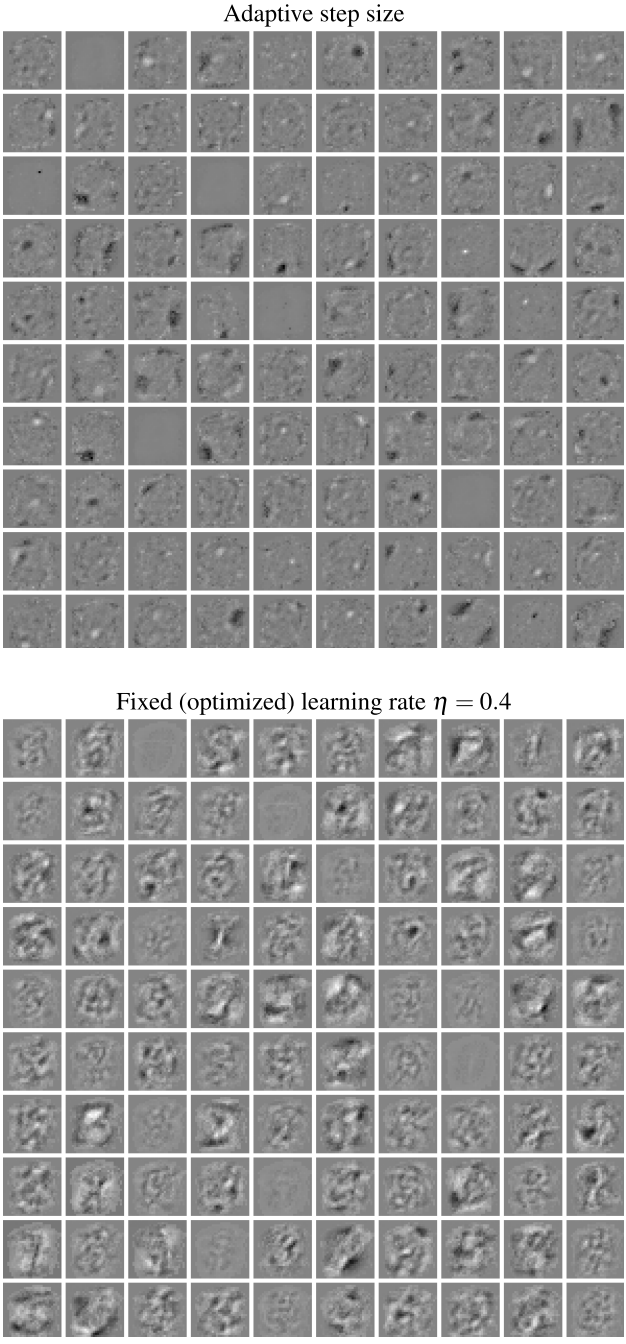


Fig. 8.17 Receptive fields of the best networks trained either with the adaptive step size or with a fixed learning rate

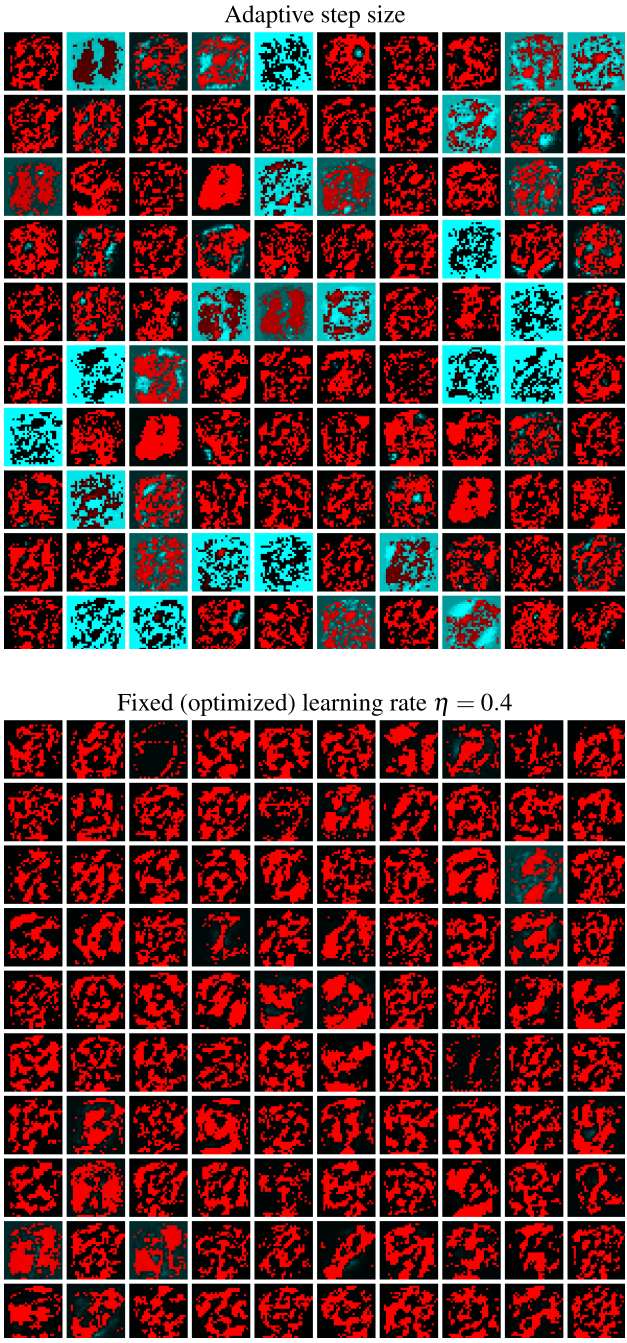


Fig. 8.18 Receptive fields excitatory (red) and inhibitory (blue) response zones for the best networks trained either with the adaptive step size or with a fixed learning rate

only the weights of the appended classification layers. Additionally, we have also decided to use a small number of training samples for each dataset. Hence, in the case of the MNIST database, we have used 1,000 samples (100 of each digit) for the training dataset and the remaining 69,000 samples for the test dataset. Similarly, for the HHreco database, we have used 650 samples (50 per symbol) for the training dataset and the remaining 7,141 for the test dataset.

During the pre-training phase, the RBMs encompassing the DBNs were trained for a maximum of 1,000 epochs. Moreover, in the discriminative phase the resulting networks were trained for a maximum of 100,000 epochs.

Figure 8.19 shows the classification performance of the resulting models, according to the number of layers of the pre-trained DBNs. Moreover, Tables 8.1 and 8.2 present the top 10 best networks achieved respectively for the MNIST and HHreco datasets. Surprisingly, the average F-Measure is inversely proportional to the number of layers (see Figure 8.19). Nevertheless, in the case of the MNIST dataset, most of the best DBNs contain four layers, not including the input layer (see Table 8.1). However, in the case of HHreco dataset, six networks out of ten contain only two layers while the remaining four contain three layers (see Table 8.2).

Table 8.1 Top 10 DBNs with the best classification performance for the MNIST dataset. The topology column refers to the topology of the added classification layers.

Topology	Pre-trained DBN layers	DBN Layers	F-Measure
MBP	784-1000-1000	784-1000-1000-30-10	82.92
MBP	784-500	784-500-30-10	82.77
MBP	784-500-1000	784-500-1000-30-10	82.72
MBP	784-500	784-500-30-10	82.49
MBP	784-500-1000	784-500-1000-30-10	82.38
MBP	784-500	784-500-30-10	82.37
MBP	784-1000-1000	784-1000-1000-30-10	82.36
MBP	784-1000-500	784-1000-500-30-10	82.19
MBP	784-500-100	784-500-100-30-10	82.11
MBP	784-500-1000	784-500-1000-30-10	82.04

Although these results could probably be improved by fine-tuning all the weights of the network, we believe that the reduced number of samples that were used prevents the higher-order layers of the DBNs from extracting useful features providing real discriminative gains, even though they may present reduced error rates. Intuitively, for these layers to be able to capture the underlying regularities of the data, the universe of training samples need to contain evidence of such regularities [138]. Naturally, the more samples we have the more likely (probable) it is for the training data to exhibit evidences of more and more complex regularities. Hence, in order to create models that can actually extract complex and useful features from the raw data, the depth of the network must take into consideration not only the number of training samples but also their diversity. In practice, however, since a DBN is

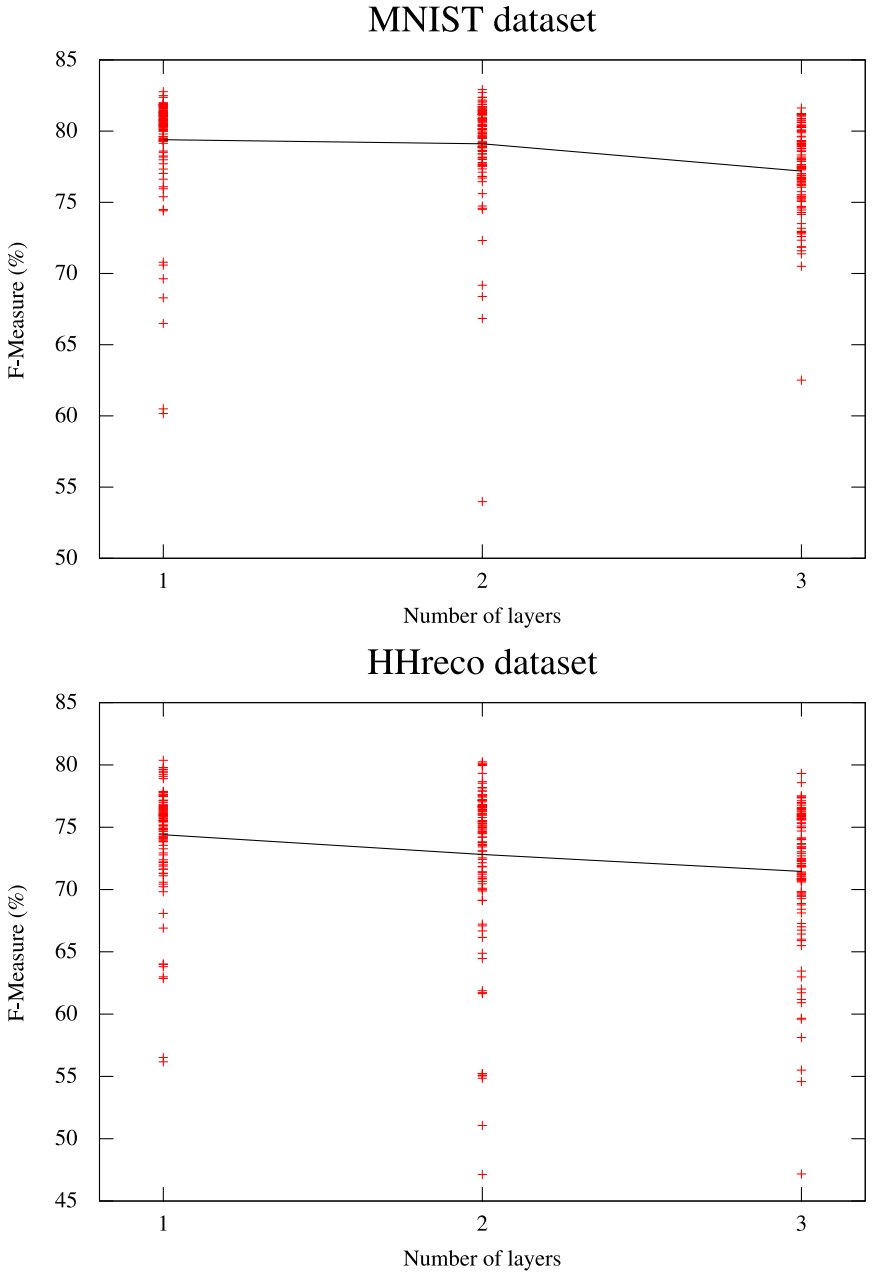


Fig. 8.19 DBNs classification performance, according to the number of pre-training layers

Table 8.2 Top 10 DBNs with the best classification performance for the HHreco dataset. The topology column refers to the topology of the added classification layers.

Topology	Pre-trained DBN layers	DBN Layers	F-Measure
BP	784-1000	784-1000-13	80.37
MBP	784-1000-500	784-1000-500-13	80.25
MBP	784-500-500	784-500-500-13	80.13
MBP	784-1000-500	784-1000-500-13	80.04
MBP	784-1000-500	784-1000-500-13	79.95
MBP	784-1000	784-1000-13	79.79
MBP	784-1000	784-1000-13	79.78
MBP	784-1000	784-1000-13	79.63
BP	784-500	784-500-13	79.61
BP	784-500	784-500-13	79.44

a modular system, it is possible to add new layers, increasing the network depth and test whether the new features improve the overall system [138]. To corroborate this idea we have performed some preliminary tests, using all the 60,000 training samples of the MNIST database. The amount of time required for training a DBN model using such volume of samples is substantially large, involving several hours of training for both the pre-training and the training phases, thus making it difficult to carry out more exhaustive tests. Nevertheless, we were able to achieve far better results than the ones presented in Table 8.1 for all of the DBN models constructed. The best DBN, which presented an F-Measure of 95.01%, is a four layer network (784-600-400-20-10). The pre-training of the original 784-600-400 DBN (each RBM was trained for 300 epochs) took approximately 3:34 hours. Then two MBP layers were added to the network and the resulting network was trained during 10,000 epochs for approximately 3:41 hours. Note that the pre-trained weights were frozen. Table 8.3 presents the confusion matrix of this network.

It is important to point out that the classification performance of the networks presented in Table 8.1 (measured over the 69,000 samples in the test dataset) is actually better than the corresponding performance measured over the standard test dataset (with 10,000 samples), making the results obtained with the full 60,000 training samples even better. Nevertheless, we are confident that these can be improved, namely through the fine-tuning of all the network weights and through the execution of additional experiments with different model configurations.

Figure 8.20 shows the classification performance of the resulting models, according to the number of neurons in the first layer of the pre-trained DBNs. Note that in this case, the average classification performance of the networks improves as the number of units in the hidden layer grows. Moreover, all of the best networks, presented in Tables 8.1 and 8.2, have at least 500 neurons in the first hidden layer. Note also that there is an expressive discrepancy, in both datasets, between the best networks containing 100 neurons in the first hidden layer (which presents an F-Measure of 81.01% and 76.74% respectively for the MNIST and HHreco datasets)

Table 8.3 Confusion matrix of the best MNIST DBN (trained with 60,000 samples)

	Predicted class									
actual	**0**	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**
0	959	0	1	2	1	3	6	2	3	3
1	0	1120	5	3	0	2	1	2	2	0
2	6	2	979	9	7	1	8	10	9	1
3	1	1	10	947	2	18	2	9	12	8
4	2	4	6	0	927	1	8	8	6	20
5	4	2	0	23	6	830	10	1	13	3
6	8	4	3	0	7	5	926	1	3	1
7	0	5	18	10	4	0	1	964	3	23
8	5	3	7	12	10	7	3	4	918	5
9	6	4	0	7	20	6	1	13	16	936

and the remaining networks presented in Tables 8.1 and 8.2. Overall, these results indicate that it is fundamental to extract a significant number of characteristics from the original data right away in the lower-level layer, because these are the key for the next layers to extract additional refined features. The results obtained suggest that the more features (neurons) the first hidden layer comprises the better, although we would need additional tests with more hidden units to confirm this trend [138].

Figure 8.21 presents the DBNs classification performance depending on the topology (BP or MBP) of the additional layers. On average, in the case of the MNIST dataset both topologies perform similarly, with slightly advantage to the BP topology. However, it is important to point out that all of the top 10 best networks, with no exception, have the MBP topology. In the case of the HHreco dataset, on average the MBP networks perform much better than the BP ones and most of the networks presented in Table 8.2 have the MBP topology. Overall, these results confirm that it is possible to enhance the performance of DBNs by including MBP layers in their architecture [138].

Figure 8.22 exhibits DBN classification performance, depending on whether an additional hidden layer with 30 neurons was added to the pre-trained networks. On average, in the HHreco dataset, having such an additional layer with randomly initialized weights turns out to be beneficial, since the classification performance is greatly enhanced. However, in the case of the MNIST dataset, the networks with the additional layer yielded slightly worse results. Nevertheless, as weird as it may seem, all of the top 10 best networks in the MNIST dataset have this additional layer and none of the HHreco have it. These results show that the DBNs performance can be improved by adding additional hidden layers with randomly initialized weights to the pre-training DBNs [138].

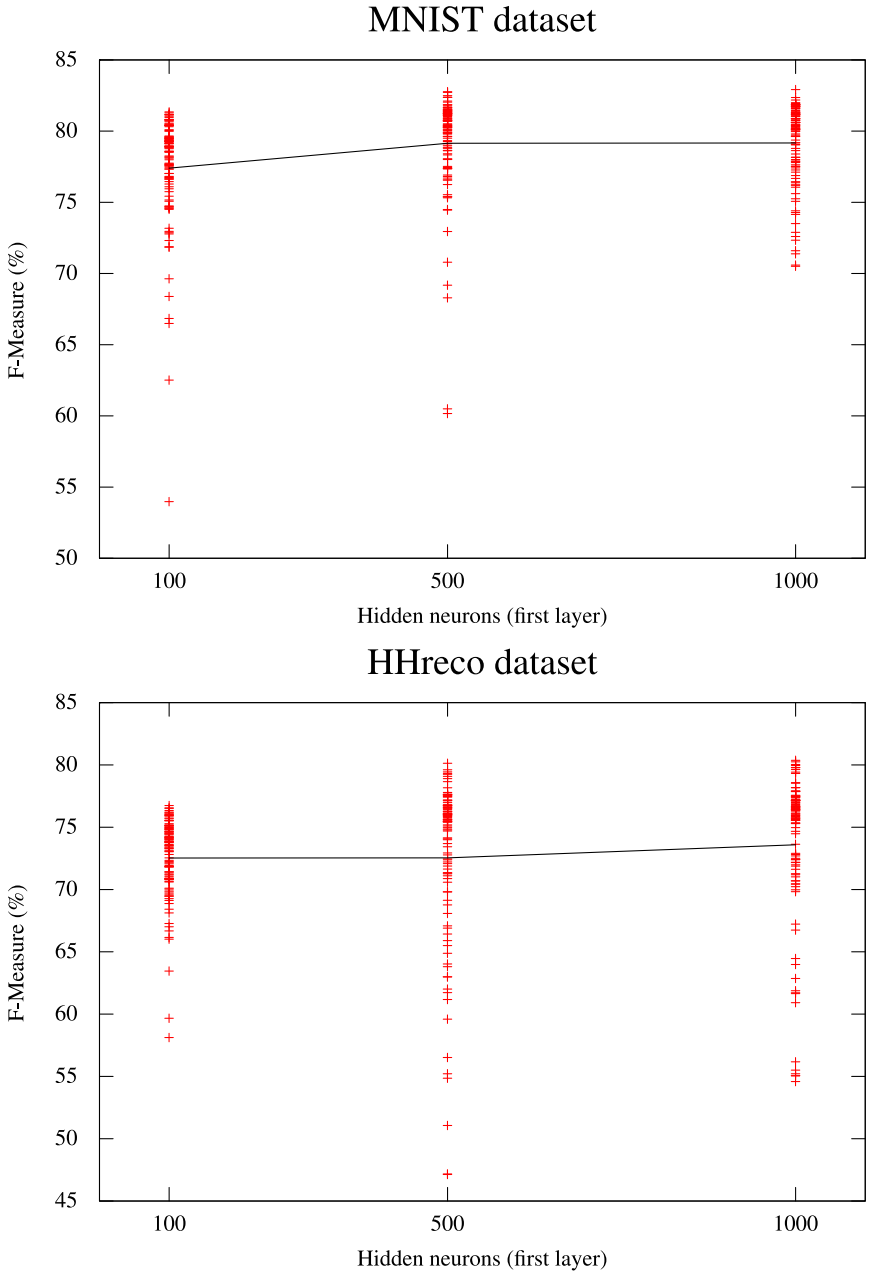


Fig. 8.20 DBNs classification performance, according to the number of neurons in the first hidden layer

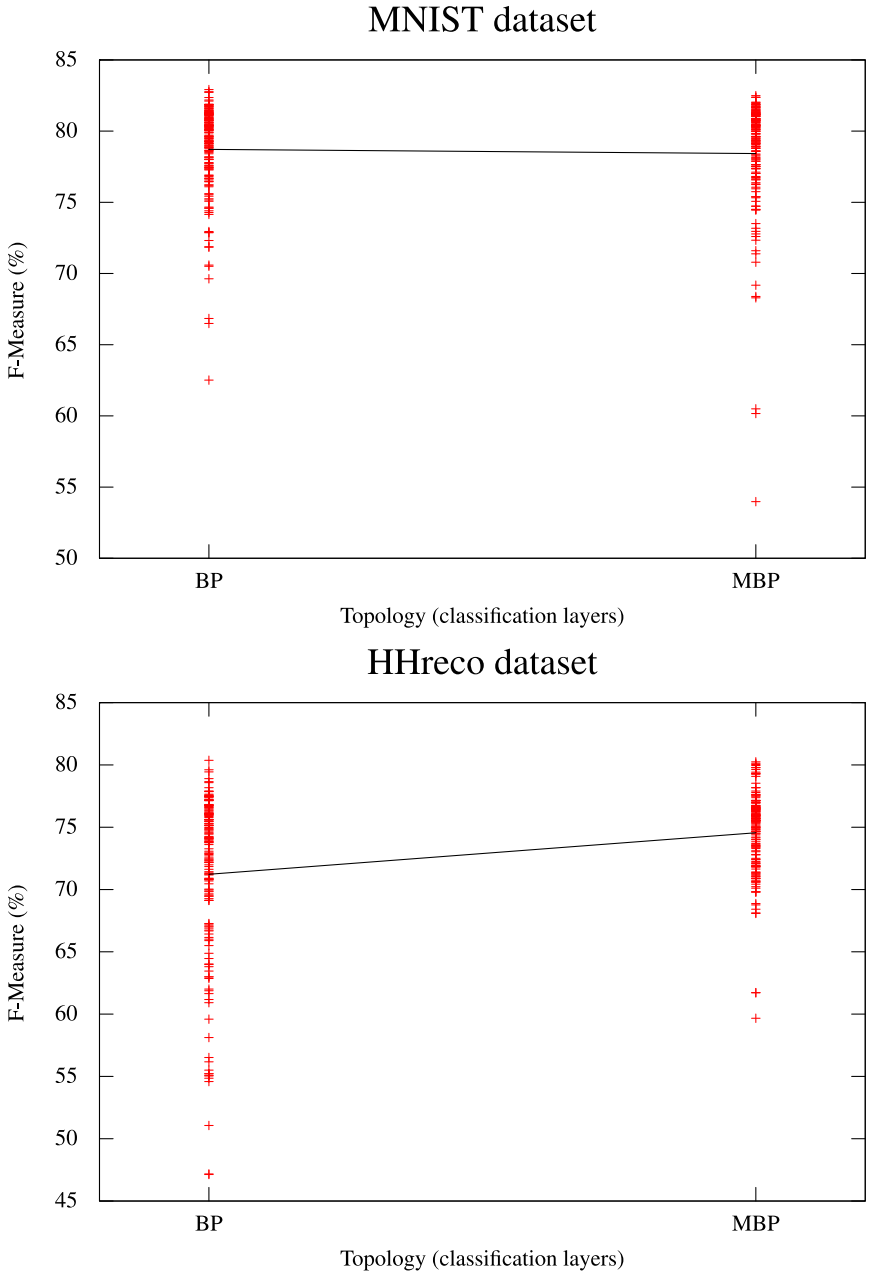


Fig. 8.21 DBNs classification performance, according to the topology of the additional layers (added to the pre-trained networks)

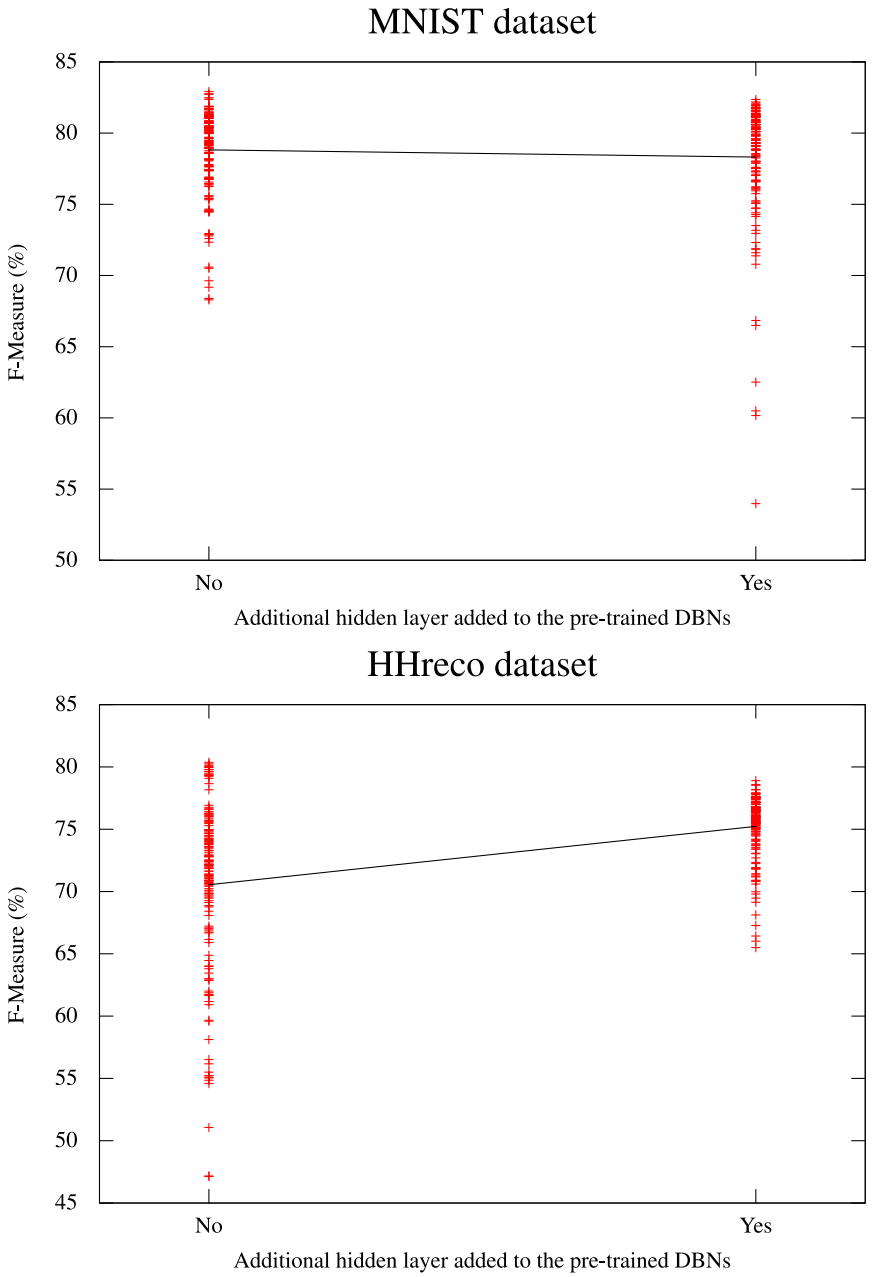


Fig. 8.22 DBNs classification performance, depending on whether or not an additional hidden layer was added to the pre-trained networks

8.7 Conclusion

This chapter introduced deep neural networks motivated by the ever since interest in understanding how the brain works. It describes recent methodologies with grounds on the probabilistic generative models composed by layers of hidden and visible stochastic units. Then the training process is explained in terms of the bidirectional flow of signal between layers in an attempt to minimize an energy function. The aim is to extracting higher-level dependencies between the original inputs variables, thereby improving the ability of the network to capture the underlying regularities in the data. Following the GPU parallel implementation, the sequence of GPU kernel calls, per epoch, that implement the CD–k algorithm was schematically described. Results in benchmarks have shown the capability of such sophisticated devices, which with such high-intensive computation clearly benefit from their implementation as many-core machine devices.

Part IV
Large-Scale Machine Learning

Chapter 9
Adaptive Many-Core Machines

Abstract. The previous chapters presented a number of novel Machine Learning algorithms and high-performance implementations of existing ones with data scalability in mind. The rationale is to increase their practical applicability to large-scale ML problems. The common underlying thread has been the recent progress in usability, cost effectiveness and diversity of parallel computing platforms, specifically, Graphics Processing Units (GPUs), tailored for a broad set of data analysis and machine learning tasks. In this chapter, we provide the main outcomes of the book through a unified view of the rationale behind adaptive many-core machines enlightened by the practical approach taken in this volume. The awareness that big data has sparked large-scale machine learning has put forward a new understanding and thinking into Big Learning. The machine learning community has to take on these challenges by parallelizing the models for the development of successful applications for a class of problems lying on the crossroads of several research topics including data sensing, data mining and data visualization. Thus we include a few promising research directions in large-scale machine learning that are likely to expand in the future.

9.1 Summary of Many-Core ML Algorithms

In this section we give a summary of the main topics covered in the book. We have in mind the technicalities covered in the framework of the many-core machines for large-scale adaptive machine learning tasks. However, we will also review other issues that in a first glance may convey to non less important ones. It is worth mentioning that all the proposed methods and tools were validated by extensive experiments and statistical evidence, using both well-known benchmarks and real-world case studies.

N. Lopes and B. Ribeiro, *Machine Learning for Adaptive Many-Core Machines – A Practical Approach,* Studies in Big Data 7,
DOI: 10.1007/978-3-319-06938-8_9,

Deep Learning Machines with Adaptive Learning

Recent work on deep belief network (DBNs) has shown that applying large-scale unsupervised feature learning the performance can greatly improve. Training billions of parameters in these models such as RBM appears to be computational challenging for modern CPUs. The GPU has been employed in many large-scale deep learning models for performance enhancement due to its massively parallel computing capability. Moreover, a new adaptive step size technique that enhances the convergence of Restricted Boltzmann Machines (RBMs), thereby effectively decreasing the training time of Deep Belief Networks (DBNs), was presented in Section 8.4. The results obtained for the MNIST database of handwritten digits demonstrate that the proposed technique is capable of decreasing substantially the number of epochs necessary to train the RBMs that support the DBN infrastructure. In addition, the technique solves the problem of finding an adequate set of learning parameters.

Semi-Supervised Non-Negative Matrix Factorization (NMF)

A novel semi-supervised algorithm, based on the Non-Negative Matrix Factorization (NMF), was presented in Section 7.4. The algorithm, Semi-Supervised NMF (SSNMF), attempts to extract unique class features (instead of "global" characteristics that are shared by several classes), assuming particular relevance for unbalanced datasets where the distinct characteristics of minority classes may be interpreted as noise by traditional NMF approaches. The experimental results demonstrate that the SSNMF reduces considerably the risk of creating poor models when compared to the original NMF method, providing a better foundation for building classification models. Additionally, the SSNMF generates sparser matrices and is over 6 times faster than the original method.

Instance-Based Learning Incremental Hypersphere Classifier (IHC)

A novel incremental supervised learning algorithm with built-in multi-class support was presented in Chapter 6. The IHC algorithm is extremely versatile and highly-scalable, being able to accommodate memory and computational restrictions, while creating the best possible model with the amount of given resources. Since the algorithm's execution time grows linearly with the amount of samples stored in the memory (which we can control), creating adaptive models and extracting information in real-time from large-scale datasets and data streams is practicable. Moreover, the experiments results, using well-known datasets (*KDD Cup 1999*,

Luxembourg Internet usage and *Electricity demand*), demonstrated that the IHC is able to handle concept drifts scenarios, while maintaining superior classification performance. Additionally, the resulting models are interpretable, making this algorithm useful even in domains where interpretability is a key factor. Finally, since the IHC keeps the samples that are at the odds of lying on the decision boundary while removing the noisy and less relevant ones, it represents a good choice for selecting a representative subset of the data for applying more sophisticated algorithms in a fraction of the time required for the complete dataset.

Hybrid Learning Framework (IHC-SVM)

The hybrid learning framework (IHC-SVM) comprises the IHC algorithm and SVMs. It was conceived for application in a protein membership prediction real-world case study. The framework uses the IHC for immediate prediction and selection of a subset of the training samples, which is subsequently used to build an SVM model. Moreover, the incremental nature of IHC allows prompt detection of significant changes which require the SVM model to be updated. Using the IHC the testing F-Measure was 93.73%, while storing less than 40% of the original samples. As anticipated, this value is smaller than the baseline yielded by the SVM model (95.91%), which was designed using the whole training set. However, the IHC-SVM approach is able to excel the baseline SVM using only a subset of the data. Using roughly half of the original data, it is possible to create improved models (with an F-measure up to 96.39%) and the data can even be further compressed without a significant change in the performance. Thus, there is strong evidence that the procedure used by the IHC to decide which samples to keep and which to discard is efficient. Overall the proposed IHC-SVM approach demonstrated capability to handle dynamic changes of real-world biological databases.

Handling Missing Values in Large-Scale Data

A novel solution, Neural Selective Input Model (NSIM), which empowers Neural Networks (NNs) with the ability to handle Missing Values (MVs), was proposed in Chapter 4. To our best knowledge this is the first method that allows Back-Propagation (BP) and Multiple Back-Propagation (MBP) networks to cope directly with this ubiquitous problem without requiring data to be preprocessed, thereby positioning these networks as an excellent alternative to other algorithms, such as decision trees, capable of dealing directly with the Missing Values Problem (MVP). Through the use of selective inputs the proposed approach accounts for the creation of different conceptual models, while maintaining a unique physical model. The NSIM excels single imputation methods while offering better or similar classification performance than state-of-the-art multiple imputation methods,

especially when the proportion of MVs is significant (more than 5% in our tests) or the prevalence of MVs affects a large number of features. Moreover, multiple imputation methods are computationally demanding and therefore impractical for large-scale datasets. In addition, the NSIM solution presents several other advantages as compared to traditional methods for handling MVs: (*i*) it reduces the burden and the amount of time associated with the preprocessing task by avoiding the estimation of MVs; (*ii*) it preserves the uncertainty inherently associated to the MVP, allowing the algorithms to differentiate between missing and real data; (*iii*) it does not require Missing At Random (MAR) or Missing Completely At Random (MCAR) assumptions to hold, since only the known data is used actively to adjust the models; (*iv*) unlike preprocessing methods which may inject outliers into the data, causing undesirable bias, the NSIM uses the best conceptual model depending exclusively on the available data; (*v*) the NSIM can take advantage of any informative knowledge associated with the MVs; (*vi*) it embodies the best solution in terms of system integration, in particular for hardware realization as it does not require the inclusion of additional and most likely complex systems; (*vii*) NSIM shows a high degree of robustness, since it is prepared to deal with faulty sensors.

The NSIM was successfully applied to a case study involving the prediction of French bankruptcy companies. The results obtained (with an F-measure of 95.70%) excel by far those previously obtained using imputation techniques and demonstrate the validity and usefulness of the proposed approach in a real-world setting.

Practical Approach: The GPU ML Library

A new open-source GPU ML library, designated by GPUMLib, was presented in Chapter 2. GPUMLib aims at providing the building blocks for the development of high-performance GPU parallel ML software, promote cooperation within the field and contribute to the development of innovative applications. Currently, GPUMLib includes several GPU parallel implementations of relevant ML algorithms, upholding considerable speedups. Since its release, GPUMLib (now with approximately 5,000 downloads) has attracted the interest of numerous people, benefiting researchers worldwide.

The GPU parallel implementations of the Back-Propagation (BP) and Multiple Back-Propagation (MBP) algorithms were presented in Section 3.3. Still within the neural network field, the GPU deployment of the Neural Selective Input Model (NSIM) for handling missing values extends the BP and MBP CUDA parallel implementation by adding three kernels.

These integrate the NSIM for handling MVs. The experiments conducted demonstrate that the GPU scales better than the CPU, reducing considerably the networks training time. The speedups obtained, ranging from 5× to 180× (on a GTX 280 device), are directly correlated to the complexity of the problem.

Using the above GPU parallel implementations, a real-world case study involving the detection of Ventricular Arrhythmias (VAs) was successfully addressed. The

results obtained presenting a sensitivity of 98.07% that improve previous work in the field, would not have been possible without the GPU speedups, which accounts for reducing the work of weeks to a matter of hours.

An Autonomous Training System (ATS) that is capable of automatically finding high-quality NNs-based solutions was presented in Section 3.4. The proposed system takes full advantage of the GPU power, searching actively for better solutions without human intervention aside from the initial configuration. The experiments demonstrate that the ATS benefits topologies with improved classification performance, saving time and effort.

A total of four distinct GPU parallel implementations of the NMF algorithm were presented in Section 7.5. These yielded speedups ranging from 55× to 706× (on a GTX 280 device) and assume particular relevance in real-world scenarios, where they may hold the key for the success of many applications.

A hybrid NMF-based face recognition approach was delineated. The rationale consists of using NMF (or SSNMF) to extract a set of parts-based features from the original images, which are subsequently used to build a classification model using a supervised learning algorithm. The proposed approach was tested on the Yale and AT&T (ORL) facial images databases, yielding competitive accuracy and evidencing superior robustness regarding different lighting conditions, thereby demonstrating its usefulness. Specifically, in the Yale dataset an average accuracy of 89.7% was obtained by the NMF-MBP approach, while for the AT&T dataset an average accuracy of 95.0% was obtained using the SSNMF-SVM approach.

A GPU parallel implementation of the CD–k algorithm, which boosts considerably the RBMs training speed was presented in Section 8.5. The implementation includes the previously mentioned adaptive step size technique to further decrease the RBMs and DBNs training time. In experiments performed on MNIST database, the GPU parallel implementation yielded speedups between 23× and 46× (on a GTX 280 device), depending on the complexity of the problem. Moreover, these demonstrate the effectiveness of coupling both approaches, which results in significant time savings.

The resulting tool was used to carry out an extensive study for analyzing the factors that affect the quality of the DBN models. The study involved training hundreds of DBNs with different configurations on two distinct handwritten character recognition databases (MNIST and HHreco). It was found that the number and diversity of training samples is highly correlated to the quality of the resulting models. Moreover, the results empirically support that these two factors have a significant impact on the maximum useful depth in the sense of improving the classification performance of a DBN. The results also suggest that the lower-level layer plays a fundamental role within the networks structure. By extracting a significant number of characteristics from the original data right away in this layer is crucial in order to obtain quality models. In fact, the results suggest that we can improve the overall models by adding additional units to the first layer although probably there should be an upper bound from which the gain is residual or even negative. Nevertheless, further experiments are required to confirm this trend. Additionally, we found that (unlike the pre-conceived idea that

all the layers within the DBN should be pre-trained) adding an additional hidden layer with randomly initialized weights to the top layer of a DBN can actually improve its classification performance by allowing the resulting network to further refine its discriminative capacity. Finally, the results show that we can improve the classification performance of DBNs by adding MBP layers (with selective actuation neurons) to their architecture.

Overall, in this book we have presented large-scale ML algorithms that successfully address the scalability issues inherent to "Big Data". Specifically, the GPU parallel implementations of ML algorithms included in GPUMLib extend the applicability of these methods to much larger datasets. To this end, a very important component is settled in the well-designed kernels, which allows to adaptively balance the workload across the many-cores devices, thereby maximizing the occupancy and throughput. Moreover, we have proposed novel GPU-based methods (the SSNMF and the CD–k adaptive step size technique) that further enhance the scalability of the existing algorithms. This addresses the unifying idea of adaptive many-core machines for machine learning making it even more attractive.

A final note on the extension of GPUMLib is given. It is important to continue the development of this framework – by filling in the dots in Figure 2.11 – in order to augment its attractiveness to researchers worldwide.

9.2 Novel Trends in Scaling Up Machine Learning

The work developed in this book is valuable for the scientific community presenting several relevant aspects in machine learning for adaptive many-core machines. As pointed out in the previous Chapters, and reiterated here, it provides many possible lines of work. Beyond these extensions, new trends for scaling up machine learning have risen due to the challenges imposed by the big data. The machine learning research community is facing many interesting problems lying in the intersection of several disciplines linked with data.

In Figure 9.1 Big Data Learning is framed in terms of large-scale learning (batch supervised and unsupervised, and incremental), scalable distributed learning platforms and many-core hardware, and novel problems to handle big data (e.g. big data visualization, mining concept drifts in large-scale social analysis, graph mining, etc.). These areas will provide promising avenues of research. In this section we uncover the diamond picture starting by briefly describing the core internal facets to outwards in an attempt to highlight encouraging directions that are likely to develop quickly in the future.

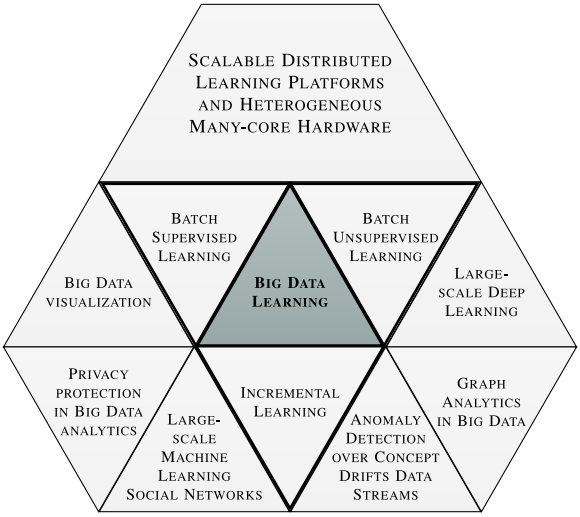


Fig. 9.1 Novel Trends in Scaling up Machine Learning

Large-Scale Batch Supervised and Unsupervised Learning

Neural network topics regained their attractiveness due to the vast success of deep learning models in many machine learning applications. Strikingly by stacking RBMs in large-scale deep architectures – as the DBNs – one can learn features from features in the hope to attain a high-level representation, closer to the latent variables. One possible way to explore this aspect is to further study the interplay between the networks architecture and its generalization performance. In particular, to use large-scale sample in order to match higher levels of representation and gain a deeper insight on how to attain a brain-like feature generation. In this line, by implementing the parallel tampering technique, which is credited for producing a quicker mix of the Markov Chain Monte Carlo (MCMC) a less biased gradient approximation could be explored. Ultimately parallel tampering may lead to better learning with a significantly higher likelihood values [63, 54].

Still in the neural networks field and by following the line developed in this book towards GPU parallel implementations of ML algorithms, it would also be interesting to investigate complex-valued neural networks [89, 161, 4]. The application field of these networks which have recently gained momentum due to their aptitude to deal with specific types of data (e.g. wave phenomena) is expected to be wider, since they can represent more information (e.g. phase and amplitude) [161].

Large-Scale Incremental Learning

Noteworthy, a new breed of data-mining, namely stream-mining where continuous data streams arrive into the system and get mined very quickly, stimulates the design of a new real-time (IHC) algorithm. The impact of different distance metrics [214, 21, 211, 12] is worthwhile to be investigated in particular for fine-grained capturing the concept drifts. Incremental Hypersphere Classifier scales linearly with an amount of pre-defined memory, independently of the volume of data to process, which makes it suitable for handling large-scale stream sources. Beyond this, more efficient ways to automatically fine-tune the gravity, g, parameter for each class, thereby assigning a distinct importance to each feature in IHC should be investigated.

Scalable Distributed Learning Platforms and Heterogeneous Many-Core Hardware

Today, the amount of data we are facing has crossed the limits of the zettabyte[1]. Such insurmountable data will drive the need not only to develop novel data analytics algorithms, new programming languages in the software point of view but also distributed learning platforms to significantly easing the handling of data.

The enormous computational power of GPUs will most likely continue to attract new researchers and programmers from a wide range of areas and in particular from the ML field, despite the difficulties inherent to the development of parallel programs for this platform. Within this framework, a promising area of research consists of taking advantage of multiple GPU systems, thereby increasing substantially the overall computational power, although this adds another layer of complexity. In this scope, the fundamental aspect lies in decomposing the domain problem to minimize communication and synchronization across GPUs. There are already successful approaches, which demonstrate the viability of using multi-GPU platforms, although this may not be viable for some problems. For example, in [152] a hybrid MPI/CUDA implementation of the NMF algorithm is presented.

Recently, the trend is to integrate CPUs and GPUs in the same die to further decrease the barriers for offloading work to the GPU [7]. Such systems, designated by Accelerated Processing Units (APUs) or Advanced Processing Units, are already part of the mainstream computers available worldwide. However, the resulting heterogeneous systems are far from perfect and new CPU designs are required. The rationale is that the CPU needs no longer to be fully general-purpose device, as it can be more effective if it specializes exclusively in the code that the GPUs can not (efficiently) parallelize [7]. Accordingly, it is likely that new research will focus on enhancing the combined (CPU-GPU) architecture and in the heterogeneous

[1] A zettabyte is a unit of information equal to one sextillion (10^{21}) or, strictly, 2^{70} bytes.

computing tools and techniques needed to fully exploit the resulting paradigm. As many other areas, ML will benefit from these innovations.

Concomitantly, there is a need to improve parallel programming, which today requires a very low level of abstraction [85]. As such, this is an area that is likely to evolve fast. For example, in the upcoming CUDA version 6.0, a unified memory system between CPUs and GPUs will be available, which will greatly simplify the tasks of memory management and data exchange, thus strikingly reducing the complexity and portability of CUDA programs.

At a different scale, distributed and cloud computing platforms represent another emerging paradigm with many advantages, such as allowing to simultaneously distribute data and computations and offering a scalable economy in terms of hosting the inherent services [85]. Subsequently, platforms such as Hadoop, Microsoft's Dryad, Amazon's EC2/S3 and other large-scale distributed systems, which are becoming increasingly accessible, provide a credible solution to many Big Data computational problems [198], although their role in scientific computing has yet to be clarified [85].

Large-Scale Machine Learning in Social Networks

Social networks have gained significant importance and have been widely studied in many fields in the last years, not only in computer science matters but also in social, political, business, and economical sciences [195, 176, 165, 56, 224, 227].

Modern challenges in large-scale social networks address new ways to look at the collective intelligence embedded in social media websites such as Facebook, Twitter, Google+, Foursquare, Instagram, LinkedIn, flicker, hi5, pinterest, goodreads, etc.. As the size of social networks largely grows, we need to study at which extent (and how) meaningful information can be extracted from the content and context by analyzing tweets, tags, blog posts, likes, posts and check-ins, in order to forge new knowledge and semantic constructs.

The importance of social media and Twitter in particular is evidenced by the topic trends raised by the occasion of recent events (e.g. the earthquake in Haiti, the uprising in Egypt, the mysterious case of the MH370 missing flight of Malaysian Airlines, etc.). Typically these platforms depend heavily on tagging. Hence, they present a flourishing ground to build cutting-edge ML applications and in the specific case above to identify the top trend a item belongs to. Having said that, large-scale social networks benefit from taking advantage of ML advances [26] by seamless integrating large data sets, infrastructures and algorithms. Accordingly, this may be an active area of research, worth of exploring [181]. In particular, new ML techniques should be researched, in order to create predictive models, about future events and trends, using the wellspring of data available in the social networks [117].

Anomaly Detection over Concept Drifts Data Streams

Big Data streams are nowadays everywhere, for instance, in computer network traffic, phone conversations, sensor networks, web searches, data centers log files and stock market transactions. In most situations, including those mentioned examples, a tremendous amount of data is continuously flowing thought diverse channels and it is not always possible to fully store the huge amounts of data circulating. As such, it raises certainly important issues such as extracting knowledge directly from the data streams and conceive new approaches to held information in Big Data repositories. Following these lines, information can be extracted or at least filtered and aggregated prior to be stored, in order to reduce significantly the Big Data burden. However, from the perspective of ML a rather not less important issue such as mining in data streams has to be taken into account. For instance, specifically monitoring across all input data streams in real-time to efficiently uncover the knowledge hidden within massive and big data can be of benefit for many applications. However, learning in non-stationary environments raises many challenges such as the concept drift as streaming data flows. The learning model must not only have the ability to continuously learn, but also the ability to adapt to concepts already acquired. Concept drift is a well-known problem in data analytics, in which the statistical properties of the attributes and their target classes shift over time, making the trained model less accurate. Many methods have been proposed for data mining in batch mode. Stream mining represents a new generation of data mining techniques, in which the model is updated in one pass whenever new data arrive. This one-pass mechanism is inherently adaptive and hence potentially more robust than its predecessors in handling concept drift in data streams [249, 218].

Anomaly detection over concept drifts data streams has been recently receiving great attention. Fraudulent and illegal activities, such as identity theft, have been increasing and are responsible for a negative and significance impact to the economy wordwide [220, 98]. Although, research efforts have been conducted on investigating and detecting external frauds (perpetrated by external entities) little or no effort has been put in detecting and preventing internal fraud (carried out by those within the organizations) [98]. Hence, new methods and tools are required to address both problems and minimize potential fraud activities [220]. In particular, the information obtained from social networks can be used to relate known fraudsters to other potential fraudulent individuals [220, 98]. Additionally, given the significant importance that social networks gained as information networks, there is an ever-growing interest in the extraction of complex information. Hence, given their dynamic nature, traditional learning models tend to be limited and a dynamic learning model strategy must be put forward. Consequently, both graph analysis and ML techniques will be pivotal for researching new solutions for this on-growing problem.

Big Graph Data Visualization

One important research topic is visualization of Big Data. A plethora of approaches each of which with specific properties and characteristics attempt to unveil the hidden knowledge embedded in data. Undoubtedly graphs have profound importance because of their versatility and expressibility. Graphs are used widely to represent physical networks, social connections, web search engines, genome sequencing or other abstract relationships. The field of graph pattern matching has been of significant importance and has wide-spread applications.

Above all, graph drawing algorithms aim at producing pleasant and readable visual representation of graphs. Traditional graph layout methods, such as force-directed or spectral layout algorithms, are all automatic graph layout techniques that take input data and generate one final layout. In [253] Yuan et al. use biochemical metabolic pathway data to demonstrate how graph layout methods can be used in a real-world scenario. The same authors have also used co-authorship network to extract style and related components while in movie graphs the goal is to define better recommendation systems. They claim that users of graph drawing systems understand their areas of application better, and often like to see certain parts of the graph drawn in a particular way to conform to the conventions of the field.

In a different perspective the use of graphs for better visualizing and understanding concepts drifts in data is presented in Pratt and Tschapek [177]. As mentioned above concept drift can affect the success of machine learning systems. Visual understanding of the interaction of features may help in the design of dynamic models that recognize and accommodate change. Visualization of concept drifts in power electric demand specifically in drift affecting peak power demands and minimal demands was successfully presented in Pratt and Tschapek [177]. The same authors use their approach for visualizing stock market returns. The goal was to visualize whether high returns associate with certain sectors and relative price ranges are stable throughout the year, and whether concept drift occurs.

Graph visualization and analysis are promising areas for spurring innovation to face the scale and complexity of Big Data. Future work and trends in this area can be found in Ma and Muelder [145].

Challenges of Privacy Protection in Big Data Analytics

In the Big Data paradigm, combining data from multiple sources to create richer datasets is a paramount task. Hence, it stands to reason that given enough data, eventually all sorts of information can be derived from the multiple pools of data, containing our digital footprints. Subsequently, Big Data embodies both opportunities and threats for individuals and organizations. The benefits, include among others, spurring innovation, improved decision making, gaining competitive advantages and even increasing the freedom of expression. On the other hand, the use of Big Data tools, including ML algorithms, to combine and analyze information

may lead to privacy and civil rights infringements in a large scale. Ultimately, Big Data systems must guarantee that an adequate set of safeguards are in place to protect the privacy of individuals [52].

9.3 Conclusion

In previous chapters of the book we dealt with machine learning in many-core adaptive machines and several algorithms and techniques based on development and deployment in GPU.

The book presents a series of new techniques to enhance and scale machine learning applications. The approaches combine high performance and robustness with an extensive assessment of the suggested techniques. They have proven to be essential to manage solutions and data in an intelligible way.

In this Chapter we gave an overview of the main techniques approached in the book where the focus was on both the data analytic algorithms and its implementation in GPU platforms. We present our view of the novel trends in scaling up machine learning algorithms from not only the data algorithmic perspective but also the distributed platforms which should be combined in a co-joint effort to handle the new coming Big Data Challenges. Moreover, the links and relations between new avenues of research were embedded in a diamond figure whose facets symbolize the intricate nature of information data, sources, platforms and new trends.

The potential of research to meet the challenges brought from big data is far from being fully exploited. And so it is our hope that we have provided a view of scaling up machine learning with many-core machines that is valuable and may be inspiring for different groups of readers: researchers and students, but also for practitioners looking for solutions for big data problems. Finally, we would remind readers that the scope of the book only provides an incomplete snapshot of scaling up big data machine learning.

Appendix A
Experimental Setup and Performance Evaluation

This Appendix describes the experimental setup configurations and the metrics used for performance evaluation, concerning the experiments carried out across this book framework. It is structured as follows. Appendix A.1 details the hardware and software configurations that were used to conduct the experiments. Appendix A.2 provides the metrics for evaluating the experiments' results. Appendix A.3 discusses the methodologies for validating a model. Appendices A.4 and A.5 receptively detail the benchmarks and the case studies that were used for conducting the experiments. Finally, Appendix A.6 describes the data preprocessing techniques that were applied to the datasets.

A.1 Hardware and Software Configurations

In order to conduct the experiments, three different computer systems were used. Table A.1 presents their main characteristics and Table A.2 the principal characteristics of the systems' GPU devices. Since the systems are heterogeneous, containing different CPUs and GPUs, in some cases we use the GPU name when referring to a specific system.

A.2 Evaluation Metrics

In this Section, we define metrics for evaluating several aspects related with the algorithms and the resulting models performance. In particular, we provide metrics for evaluating the: GPU parallel implementations, training progress, instance selection methods and the models classification performance.

Table A.1 Hardware and Software system main characteristics

	Main Characteristics
System 1 (8600 GT)	Intel Core 2 6600 (2.4GHz) NVIDIA GeForce 8600 GT Windows Vista (x64) 4GB memory
System 2 (GTX 280)	Intel Core 2 Quad Q 9300 (2.5GHz) NVIDIA GeForce GTX 280 Windows 7 (x64) 4GB memory
System 3 (GTX 460)	Intel Dual-Core i5-2410M (2.7GHz) NVIDIA GeForce GTX 460 Windows 7 (x64) 8GB memory
System 4 (GTX 570)	Intel Quad Core i5-750 (3.33GHz) NVIDIA GeForce GTX 570 Windows 7 (x64) 12GB memory

Table A.2 Main characteristics of the NVIDIA GeForce devices used in this work

	8600 GT	**GTX 280**	**GTX 460**	**GTX 570**
Compute capability	1.1	1.3	2.1	2.0
SMs	4	30	7	15
Number of cores	32	240	336	480
Peak performance (GFLOPS)	113.28	933.12	940.8	1405.4
Device memory	512MB	1GB	1GB	1.25GB
Shared Memory per block	16KB	16KB	48KB	48KB
Maximum threads per block	512	512	1024	1024
Memory bandwidth (GB/sec)	22.4	141.7	112.5	152.0
Shading clock speed (GHz)	1.2	1.3	1.4	1.56

GPU Parallel Performance

In order to compare the performance of the GPU parallel implementations with the corresponding CPU sequential ones, we use the speedup ($\times$), defined as (A.1):

$$speedup = \frac{time_S}{time_P} . \tag{A.1}$$

where $time_S$ is the time needed to execute the algorithm using a CPU sequential implementation and $time_P$ the corresponding time of the GPU parallel implementation. Accordingly, the speedup measures how many times faster the GPU parallel implementation is relative to the CPU baseline sequential implementation. Note that the time for accessing input and output devices is not included in the speedup computation. Moreover, the time for transferring information to the GPU is also excluded.

Training Progress

For measuring the NNs training progress, we use the RMSE, which is given by (A.2):

$$RMSE = \sqrt{\frac{1}{NC}\sum_{i=1}^{N}\sum_{j=1}^{C}(Y_{ij} - T_{ij})^2}\,. \tag{A.2}$$

Instance Selection Performance

When considering instance selection models, it is important to measure the storage reduction or space savings, which is given by (A.3):

$$storage = 1 - \frac{n}{N}\,. \tag{A.3}$$

where n is the number of samples stored.

Models Performance

For evaluating the performance of the classifier models and asserting its quality, we use the accuracy, precision, recall (sensitivity), specificity and F-measure (F_1 score) metrics, expressed as percentages. These metrics are based on a confusion matrix, containing the number of correctly and incorrectly classified examples for each class, in the form of true positives (tp), true negatives (tn), false positives (fp) and false negatives (fn). Table A.3 presents the confusion matrix for a two class (binary) problem. A perfect classifier should only present non-zero values in the confusion matrix main diagonal, as these correspond to correct classifications, while the remaining "cell" values represent mis-classified samples.

The accuracy is the most commonly used ML performance measure [213]. It represents the proportion of the predictions that are correct as given by (A.4):

$$accuracy = \frac{tp + tn}{tp + fp + fn + tn}\,. \tag{A.4}$$

Table A.3 Confusion matrix for a binary classification problem

Class predicted	Actual class	
	Positive	**Negative**
Positive	tp	fp
Negative	fn	tn

Although the accuracy gives an overall estimate of the performance of a classifier, it can be misleading, in particular for unbalanced datasets (with a big discrepancy in the number of samples belonging to each class) where a classifier may never predict correctly a class of interest and still obtain high-accuracy values. To solve this problem two other metrics, precision and recall (sensitivity), respectively given by (A.5) and (A.6) for a binary classification problem, are commonly used:

$$precision = \frac{tp}{tp + fp}\,, \tag{A.5}$$

$$recall = sensitivity = \frac{tp}{tp + fn}\,. \tag{A.6}$$

A classifier presenting a high precision rate is rarely wrong when it predicts that a sample belongs to the class of interest (positive). On the other hand, a classifier exhibiting a high recall rate rarely mis-classifies a sample that belongs to the class of interest. Usually there is a trade-off between the precision and the recall, and although there are cases where it is important to favor one in detriment of the other, generally it is important to balance and maximize both. This can be accomplished by using the F-measure, which is given by (A.7) for a binary classification problem:

$$Fmeasure = 2 \times \frac{precision \times recall}{precision + recall}\,. \tag{A.7}$$

Concerning binary classification problems, the precision, recall and F-measure metrics neglect the classification of negative examples [213]. Hence, depending on the problem, other measures such as the sensitivity (true positive rate), see (A.6), and specificity (true negative rate) may be more appropriate. The latter is given by (A.8):

$$specificity = \frac{tn}{fp + tn}\,. \tag{A.8}$$

When considering multi-class problems, the aforementioned metrics need to be replaced by others that take into account the performance of all the classes. For this purpose, we can use the macro-average precision and recall, which are given respectively by (A.9) and (A.10).

$$precision_M = \frac{\sum_{c=1}^{C} \frac{tp_c}{tp_c+fp_c}}{C}, \tag{A.9}$$

$$recall_M = \frac{\sum_{c=1}^{C} \frac{tp_c}{tp_c+fn_c}}{C}, \tag{A.10}$$

where tp_c, tn_c, fp_c and fn_c are respectively the number of true positives, true negatives, false positives and false negatives when considering the class c samples as positive examples and the remainder as negative examples. Note that the macro-average F-measure is computed as in (A.7), but using the macro-average precision and recall.

Throughout this Thesis, we rely mainly on the macro-average F-measure to evaluate the models classification performance, even when considering binary problems, so that all the classes have the same importance. However, due to the specific nature of some problems and for comparability purposes with previous work, other metrics are also used whenever appropriate.

A.3 Validation

A model can predict accurately the training data and still have a poor performance when classifying new (unseen) data. In particular, this is the case of overfitting models. Therefore, when building a model, we are predominantly interested in its ability for correctly classifying unseen data. The degree of success in doing so is called generalization [6, 18]. In this context, cross-validation techniques assume particular relevance for estimating the models generalization performance.

Cross-validation is a statistical method for evaluating the models performance that divides the available data into two types of partitions: (*i*) training data partitions that are used for creating the models and (*ii*) testing data partitions that are used for validating the resulting models [185]. In the remainder of this Section we summarize the main cross-validation strategies.

Hold-Out Validation

This is perhaps the simplest strategy, which consists of splitting the data into two disjoint datasets: a train dataset that is used for building a model and a test dataset that is used to estimate the models generalization performance (using the metrics specified in the previous Section). We use this method mainly when the objective consists of solving a particular problem (for which in many cases previous work exists and the data has already been divided into training and testing datasets).

k–Fold Cross-Validation

In this strategy data is partitioned into k (nearly) equally sized parts, called folds. Subsequently k different models are built using a distinct fold for validation and the remainder $k-1$ folds for building the models. The models performance is then averaged over all the experiments. Figure A.1 shows the experiments associated with a 4-fold cross-validation procedure.

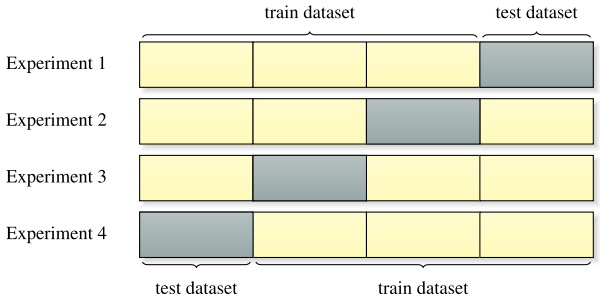


Fig. A.1 Experiments associated with a 4-fold cross-validation procedure

In order to compare the performance of different algorithms, we use 5–fold or 10–fold cross-validation, instead of the hold-out, to avoid undesirable bias that may arise from a particular training/test split. Moreover, the data is stratified so that each fold contains a representative subset of the original data, i.e. all the folds will contain (approximately) the same number of samples per class [185].

Repeated *k*-Fold Cross-Validation

This strategy is used to increase the number of estimates, by running the k-fold cross-validation multiple times. The rationale consists of obtaining a more trustworthy performance estimate. At each run, data is randomized and new k-folds, containing different samples, are obtained [185]. This strategy was used to validate the performance of the Incremental Hypersphere Classifier (IHC) in Chapter 6.

Leave-One-Out Cross-Validation

This strategy corresponds to the N–fold cross-validation, i.e. each fold contains a single sample. This method is appropriate when the available data is scarce, however due to its high-variance it can lead to unreliable estimates [185].

Leave-One-Out-Per-Class Cross-Validation

This strategy corresponds to the stratified k–fold cross-validation, for $k = \frac{N}{C}$, and it is adequate for comparing the performance of different algorithms in balanced datasets that contain only a small number of samples per class. Note that this strategy is different from the leave-one-out cross-validation, since each fold will contain C samples (one per class). We use this strategy to validate the performance of Non-Negative Matrix Factorization (NMF) based methods in Chapter 7.

Repeated Random Sub-sampling Validation

The rationale of this strategy is the same as the repeated k-fold cross-validation, i.e. increasing the number of estimates to obtain a more reliable performance estimate. As in the case of the repeated k-fold cross-validation, the process of splitting the data is repeated several times. However in this case it consists of randomly dividing data through the training and test datasets. The advantage of this method is that the proportion of training and testing subsets is independent of the number of experiments. However, some data samples may never be used for validation while others may be chosen more than once, thereby causing undesirable bias.

A.4 Benchmarks

Several benchmarks including face and character recognition databases are used to validate the algorithms and their respective models. Table A.4 presents the main characteristics of the benchmark datasets, after preprocessing (see Appendix A.6 for details about the preprocessing techniques used).

Since most of the benchmarks were obtained at the University of California, Irvine (UCI) Machine Learning Repository [11] and therefore are well-known and documented, we describe only the remaining datasets.

AT&T Face Database

The AT&T face database, formerly known as the ORL Database, includes the images of 40 different individuals and it is available at `http://www.cl.cam.ac.uk/research/dtg/attarchive/facedatabase.html`. For each individual there are 10 distinct images. Hence, the database is composed of 400 images with 112×92 pixels. The images were taken at different times, varying the lighting conditions, facial expressions and details (open / closed eyes, smiling / not smiling, glasses / no glasses). Figure A.2 presents randomly selected images from this database.

Table A.4 Main characteristics of the benchmark datasets

Dataset (Benchmark)	**Samples** (N)	**Features** (D)	**Classes** (C)
Adult	32,561	14	2
Annealing	898	47	5
AT&T (ORL)	400	10,304	40
Audiology	226	93	24
Breast cancer Wisconsin (Diagnostic)	569	30	2
Breast cancer Wisconsin (Original)	699	9	2
CBCL face database #1	2,429	361	–
Congressional	435	16	2
Ecoli	336	7	8
Forest cover type	11,340	54	7
Electricity demand (elec2)	45,312	4	2
German credit data	1,000	59	2
Glass identification	214	9	6
Haberman's survival	306	3	2
Heart - Statlog	270	20	2
Hepatitis	155	19	2
HHreco	7,791	784	13
Horse colic	368	92	2
Ionosphere	351	34	2
Iris	150	4	3
Japanese credit	690	42	2
KDD Cup 1999	4,898,431	40	5
Luxembourg Internet usage	1,901	31	2
Mammographic	961	5	2
MNIST	70,000	784	10
Mushroom	8,124	110	2
Pima Indian diabetes	768	8	2
Poker hand	25,010	85	10
Sinus cardinalis	101	1	–
Sonar	208	60	2
Soybean	683	77	19
Tic-Tac-Toe	958	9	2
Two-spirals	194	2	2
Vehicle	946	18	4
Wine	178	13	3
Yale face database	165	4,096	15
Yeast	1,484	8	10

Fig. A.2 Randomly selected examples from the AT&T (ORL) face images

CBCL face database #1

The CBCL face database #1 of the Massachusetts Institute of Technology (MIT) is available at `http://cbcl.mit.edu/cbcl/software-datasets/FaceData2.html`. This database contains both facial and non-facial gray-scale images of $19 \times 19 = 361$ pixels. The training dataset includes a total of $2,429$ facial and $4,548$ non-facial images, while the test dataset contains a total of 472 facial and $23,573$ non-facial images. However, we have only used the $2,429$ faces of the training dataset, from which randomly selected images are presented in Figure A.3.

Electricity Demand (Elec2)

The *Electricity Demand* is a real-world problem that was obtained at `http://www.liaad.up.pt/area/jgama/ales/ales_5.html`. It contains data from the Australian New South Wales (NSW) Electricity Market, where the prices depend both on the demand and supply. The goal is to predict if the price will drop or increase [65].

HHreco Multi-stroke Symbol Database

The HHreco multi-stroke symbol database, available at `http://embedded.eecs.berkeley.edu/research/hhreco/`, contains a total of $7,791$ samples generated by 19 different persons. Overall, the database contains a total of 13 different symbol classes: ellipse, heart, trapezoid, pentagon, arch, hexagon, square, triangle, cube, cylinder, parallelogram, moon and callout [92]. Each user created at least 30 multi-stroke images per class, which means that for each symbol there are at least $19 \times 30 = 570$ samples. We converted the original HHreco vector strokes into a $28 \times 28 = 784$ raster pixel image, maintaining the aspect ratio of the original shapes. Moreover, the resulting images were binarized. Note that both the number of strokes and the time span information were discarded, since this particular dataset is used to evaluate the capacity of Deep Belief Networks (DBNs) to extract information from the original (images) raw data (see 8.6). Figure A.4 presents examples of the HHreco images.

KDD Cup 1999 Database

The KDD cup 1999 database, available at `http://www.kdd.org/kddcup/index.php`, contains approximately 5 million samples. The objective consists of building a computer network intrusion detector capable of distinguishing between normal connections and four different

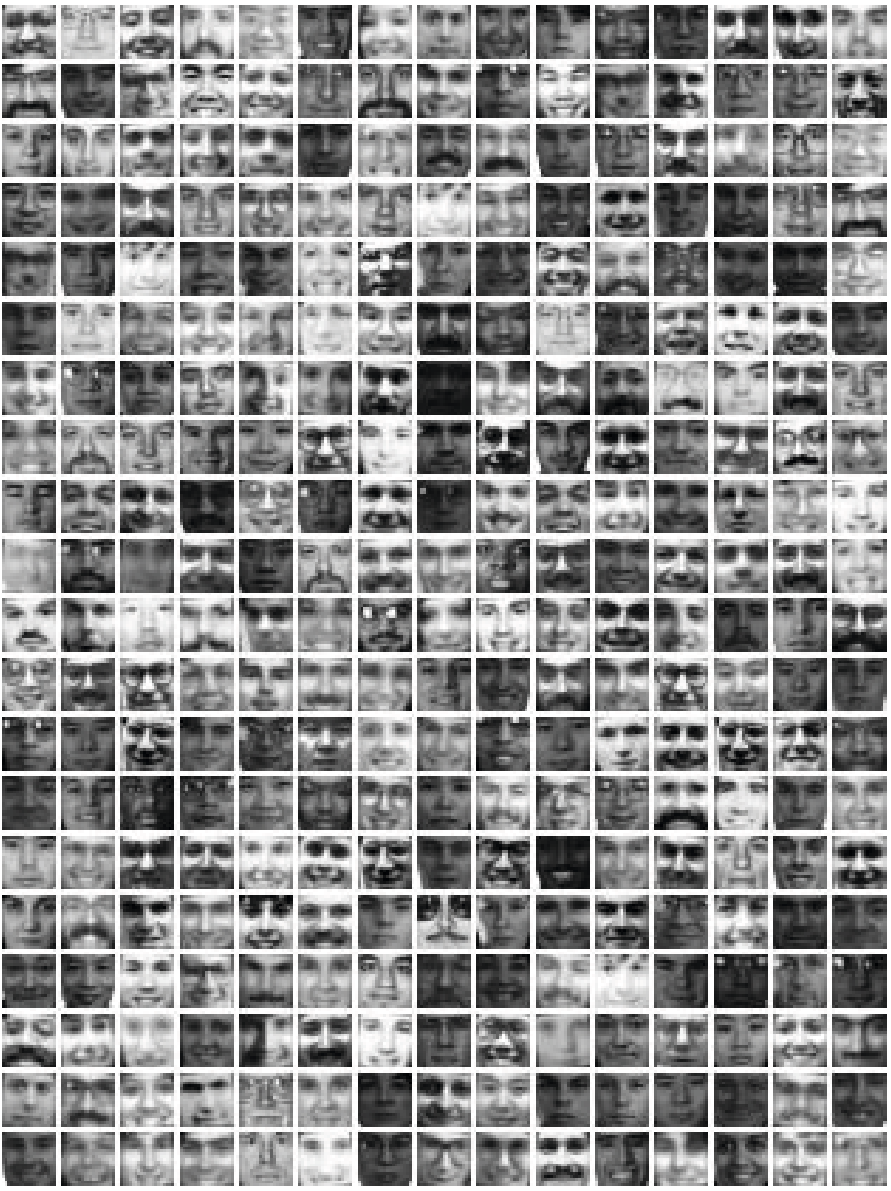


Fig. A.3 Randomly selected examples of the face images contained in the CBCL training dataset

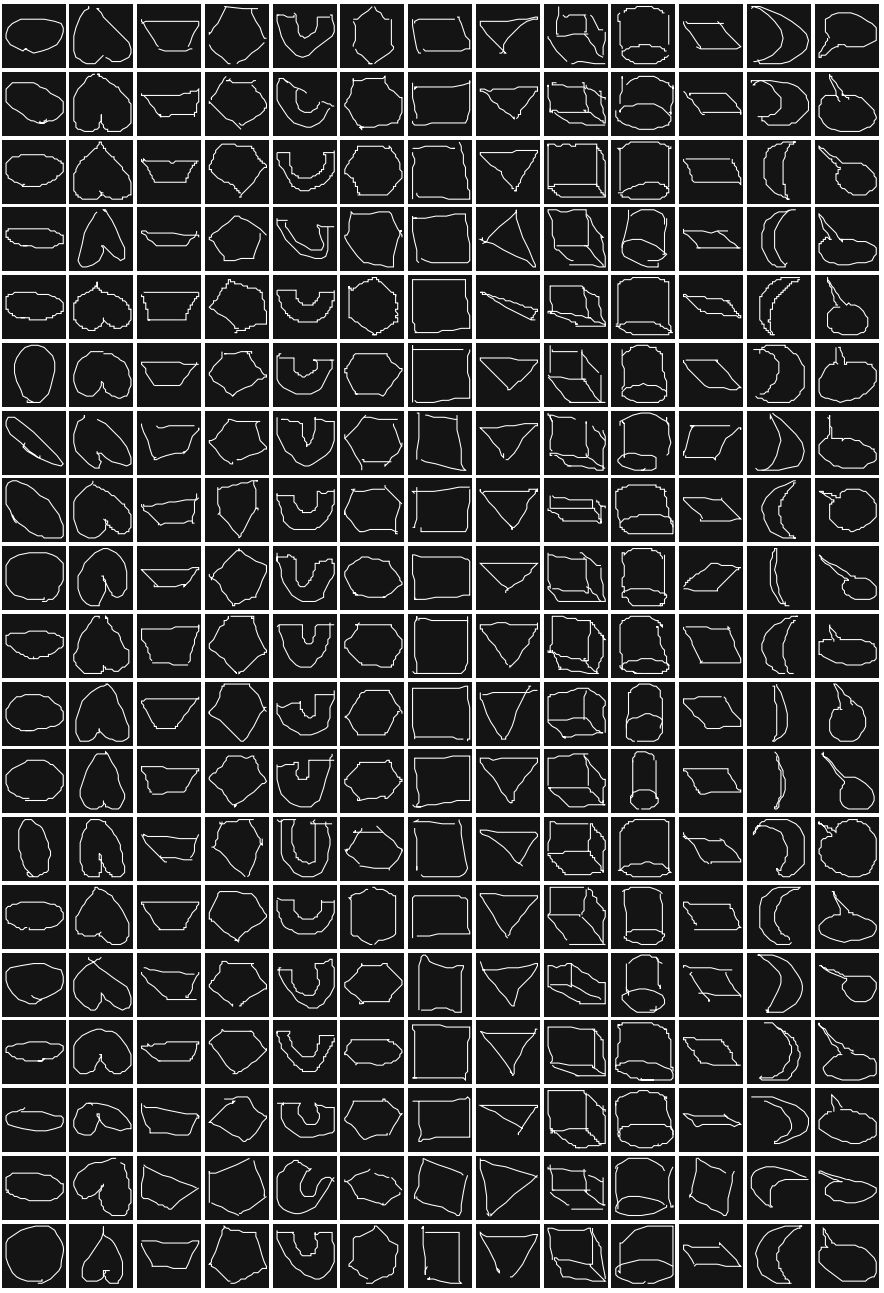


Fig. A.4 Examples of the HHreco multi-stroke images. Each column contains a symbol while each row contains the images drawn by one of the users.

types of attacks: Denial Of Service (DOS), unauthorized access from a remote machine (R2L), U2R and probing.

Luxembourg Internet Usage

The *Luxembourg Internet usage* is a real-world problem that was obtained at `https://sites.google.com/site/zliobaite/resources-1`. It contains the answers given by several individuals in a survey questionnaire. The objective consists of classifying each individual with respect to its Internet usage (high or low) [100, 237]. The questionnaires were collected over a period of 5 years, thus it is expected that the concept will change over time.

MNIST Hand-Written Digits Database

The MNIST database of hand-written digits is available at `http://yann.lecun.com/exdb/mnist/` and contains a total of 70,000 samples (60,000 train samples and 10,000 test samples). Each sample consists of a $28 \times 28 = 784$ pixels image of a hand-written digit. Figure A.5 presents examples of the MNIST images. Note that all the images were binarized.

Sinus Cardinalis

The *Sinus Cardinalis* benchmark is a regression problem, that consists of approximating the following function:

$$sinc(x) = \frac{\sin(x)}{x} . \tag{A.11}$$

In order to build the training dataset we collected a total of 101 samples from the referred function, uniformly distributed in the interval $[-10, 10]$. Note that for $x = 0$ $sinc(x)$ is considered to be 1. Figure A.6 presents a graphical plot of this function.

Two-Spirals

The *two-spirals* benchmark, obtained from the Carnegie Mellon University (CMU) learning benchmark archive is available at `http://www.cs.cmu.edu/afs/cs/project/ai-repository/ai/areas/neural/bench/cmu/` and it is considered to be an extremely hard problem to solve for algorithms of the BP family [60]. It consists of discriminating between the points of two distinct spirals which coil three times around one another and around the x–y plane origin as depicted in Figure A.7.

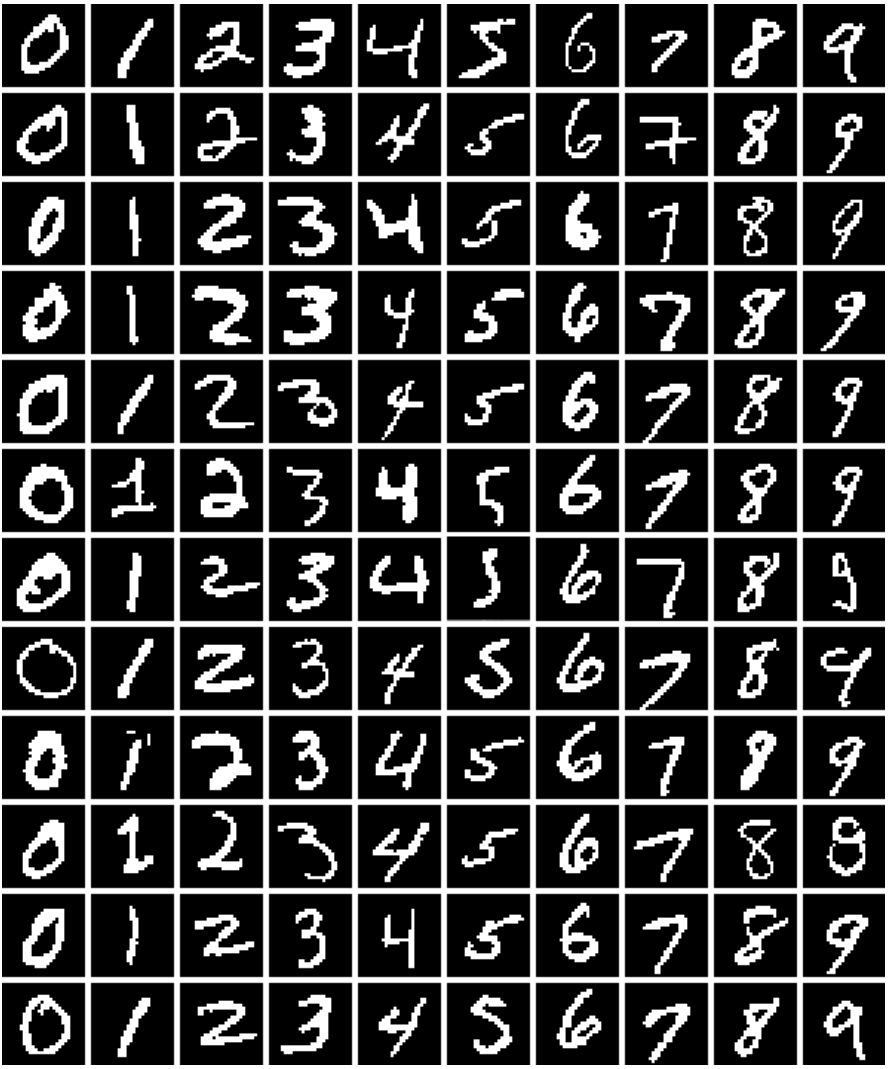


Fig. A.5 Examples of the MNIST hand-written digits. Each column contains a different digit, starting with 0 in the left-most column and ending with 9 in the right-most column.

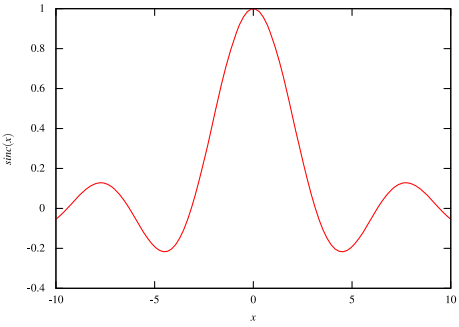


Fig. A.6 *Sinus Cardinalis* function

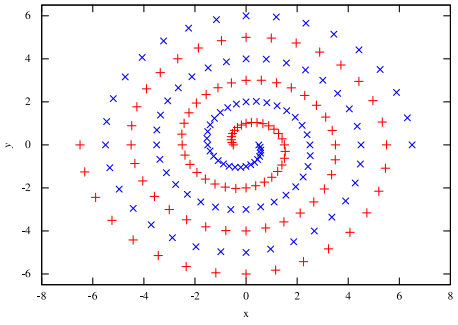


Fig. A.7 Two spirals dataset

Yale Face Database

The Yale face database, available at `http://cvc.yale.edu/projects/yalefaces/yalefaces.html`), is comprised of 165 gray-scale images encompassing 15 individuals. Each individual appears in 11 images, each representing a different facial expression (happy, normal, sad, sleepy, surprised, winking) or configuration (center-light, left-light, right-light, with glasses, without glasses). The images were cropped to the size of 64×64 pixels. Figure A.8 shows the Yale face database images.

A.5 Case Studies

To complement the benchmarks, we address real-world problems within the fields of biomedical, finance and business, and bio-informatics for practical validation of our computational experiments. Table A.5 presents their main characteristics.

Fig. A.8 Yale face images. Each row contains the images of a specific individual and each column a different expression/ configuration.

Table A.5 Main characteristics of the real-world case studies

Dataset (Benchmark)	Samples (N)	Features (D)	Classes (C)
Bankruptcy	3,048	89	2
Peptidases	20,778	3,875	2
MP3 Steganalysis	1,994	742	2
Ventricular arrhythmias	19,391	18	2

Financial Distress Prediction

In recent years, due to the global financial crisis (triggered by the sub-prime mortgage crisis), the rate of insolvency has been aggravated globally. As a result investors are now more careful about entrusting their money. Moreover, determining whether or not firms are healthy is of major importance, not only to investors and stakeholders but also to everyone else that has a relationship with the analyzed companies (e.g. suppliers, workers, banks, insurance firms). Although this is a widely studied topic, estimating the real healthy conditions of firms is becoming a much harder task, as companies become more complex and develop sophisticated schemes to conceal their real situation. In this context, automated ML systems that can accurately predict the risk of insolvency and warn, in advance, all those who may be affected by a bankruptcy process are of major importance [130].

The datasets of this problem were obtained from a large database of French companies, containing information of an ample set of financial ratios spawning over a period of several years. The referred database contains information about $107,932$ companies, out of which $1,653$ became insolvent in 2006. The objective consists of discriminating between healthy and distressed companies based on the record of the financial indicators from previous years. For this purpose, we considered 29 financial ratios over the immediate previous three years (see Table A.6) as well as two more features: the number of employees and the turnover. Thus, altogether a total of 89 features were considered [130]. Additional details on the construction of the dataset are given later in Section 4.5 (page 82).

Protein Membership Prediction

The study of proteins plays a prominent role in understanding many biological systems. In particular, the classification of protein sequences into functional and structural groups based on sequence similarity is a contemporary and relevant task, in the bio-informatics domain, for which huge amounts of data already exist. However, despite all the energy spent into deciphering the proteomes, the available knowledge is still limited [156]. Peptidases are a class of proteolytic enzymes that catalyze chemical reactions, allowing the decomposition of protein substances into smaller molecules. They are involved in several processes that are crucial for the correct functioning of organisms. Their importance is proved by the fact that

Table A.6 Financial ratios selected to create a bankruptcy model

Financial ratios	
Financial Debt / Capital Employed (%)	Working Capital / Turnover (days)
Capital Employed / Fixed Assets	Net Current Assets / Turnover (days)
Depreciation of Tangible Assets (%)	Working Capital Needs / Turnover (%)
Working Capital / Current Assets	Export (%)
Current Ratio	Value Added per Employee
Liquidity Ratio	Total Assets / Turnover
Stock Turnover days	Operating Profit Margin (%)
Collection Period	Net Profit Margin (%)
Credit Period	Added Value Margin (%)
Turnover per Employee	Part of Employees (%)
Interest / Turnover	Return on Capital Employed (%)
Debt Period (days)	Return on Total Assets (%)
Financial Debt / Equity (%)	EBIT Margin (%)
Financial Debt / Cashflow	EBITDA Margin (%)
Cashflow / Turnover (%)	

approximately 2% of the genes in all kinds of organisms encode peptidases and their homologues [184]. Hence, its detection is central to a better understand of their role in a biological system [140].

For the purpose of peptidase detection, a dataset constructed from the MEROPS [184] and the Structural Classification Of Proteins (SCOP) [158] databases was used. A total of 20,778 proteins sequences were randomly selected from both databases: 18,068 positive samples from MEROPS 9.4 and 2,710 sequences of non-peptidases from SCOP 1.75. The dataset was then divided into two groups, 17,164 sequences for training (15,358 positive and 1,806 negative examples) and 3,614 sequences for testing purposes (2,710 positive examples and 904 negative) [173].

The features of the protein primary structure were extracted using text mining techniques [46]. The idea consists of splitting the continuous flow of amino acids into substrings (n-grams) of length n. The n-grams are formed by n consecutive characters and each corresponds to a feature in a particular sequence. For example, considering the partial sequence 'PKIYGY', the trigrams are 'PKI', 'KIY', 'IYG' and 'YGY'. The Word Vector Tool (WVTool) library [245] was used in order to obtain the unigrams, bigrams, trigrams and the combinations of n-grams. The dataset was then built by taking into account the relevance of each feature (n-gram) [50].

MP3 Steganalysis

The MP3 Steganalysis real-world problem consists of discriminating between normal MP3 audio files (cover) and MP3 files with hidden information (stego). The dataset was created using the four methods described in Qiao et al. [178].

Ventricular Arrhythmias

In the Ventricular Arrhythmias (VAs) problem, the objective consists of detecting Premature Ventricular Contractions (PVCs), based on the time and frequency domain features that were extracted in Marques [149] directly from the bio-signal Electrocardiographs (ECGs) data, available at the MIT-BIH Arrhythmia Database (`http://www.physionet.org/physiobank/`).

This problem is particularly important, because nowadays most countries face high and increasing rates of cardiovascular diseases. In Portugal there is a 42% probability of dying of these diseases and worldwide they are accountable by 16.7 million deaths per year [243]. In this context, VAs assume a significant role, since their prevalence can lead to life threatening conditions, which may result in cardiac arrest and sudden death. VAs evolve from simple PVCs, which are usually benign, to ventricular tachycardia and finally to critical ventricular fibrillation episodes which are potentially fatal and the main cause of sudden cardiac death. Hence, the detection of PVCs from an ECG is of major importance, since they are associated with an increased risk of adverse cardiac events [187].

A typical ECG tracing of an ordinary heartbeat consists of a P wave, a QRS complex and a T wave (observe Figure A.9). PVCs result from an ectopic depolarization on the ventricles, which causes a wider and abnormally shaped QRS complex. Typically, these complexes are not preceded by a P wave, and the T wave is large and with an opposite direction to the major QRS deflection [187].

Table A.7 identifies the selected features from the ECG signal. For comparison purposes, we used the same training, test and validation datasets (each one with 19,391 samples) that were used in Marques [149] and in Ribeiro et al. [187].

A.6 Data Preprocessing

Datasets are often disturbed by problems of noise, bias and large variations in variables dynamic range [120]. Accordingly, the data preprocessing task assumes particular relevance for designing good generalization performance models [105]. Typically, in the preprocessing phase, the original input vectors are projected into a new space of variables where (hopefully) better solutions can be found. Note that the same preprocessing techniques must be applied for both training and test data [18].

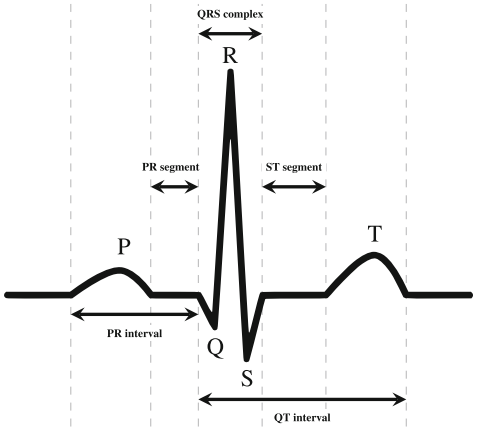


Fig. A.9 Typical ECG diagram of a normal sinus rhythm for a human heart

Table A.7 Selected features from the ECG signal

Feature	Description
RRav	RR mean interval
RR0	Last RR interval
SN	Signal/Noise estimation
Ql	Q-wave length
(Qcx, Qcy)	Q-wave mass center (x,y) coordinates
(Qpx, Qpy)	Q-wave peak (x,y) coordinates
Rl	R-wave length
(Rcx, Rcy)	R-wave mass center (x,y) coordinates
(Rpx, Rpy)	R-wave peak (x,y) coordinates
Sl	S-wave length
(Scx, Scy)	S-wave mass center (x,y) coordinates
(Spx, Spy)	S-wave peak (x,y) coordinates

In terms of preprocessing, we typically perform the following operations, in the specified order:

1. Remove outliers;
2. Replace qualitative variables by quantitative variables;
3. Rescale the variables.

However other operations may be required, depending on the problem.

Regarding the qualitative variables, those containing only two possible values (e.g. ‘yes’, ‘no’; ‘true’, ‘false’; ‘success’, ‘failure’) are replaced by a single binary variable. Otherwise, if there is an explicit order between values that makes sense, then the (qualitative) variable is replaced by a single quantitative variable using non-negative integers numbers and preserving the order of the original values. In case

none of the previous apply, the original variable is replaced by k different binary variables, such that each one of the k domain values has its own associated variable that is 1 when the original variable presents that specific value and 0 in the remaining cases [47].

Rescaling the variables is important to avoid different (magnitude) scales between the variables that may adversely impose a bias on the algorithms. For this purpose, we use the min-max rescaling, which is given by (A.12):

$$x = \frac{x' - min}{max - min}(nmax - nmin) + nmin\,, \tag{A.12}$$

where x' is the original feature value, x the new value, min and max respectively the old minimum and maximum values, and $nmin$ and $nmax$ respectively the new minimum and maximum values for the variable [105]. All input variables are rescaled between -1 and 1, except for the Restricted Boltzmann Machines (RBMs) and Deep Belief Networks (DBNs) which require binary inputs and for the Non-Negative Matrix Factorization (NMF) algorithms that require non-negative input data. Thus, in the latter, the variables are rescaled between 0 and 1.

Histogram Equalization

Concerning the face recognition datasets (AT&T, CBCL and Yale), a histogram equalization was applied to the face images, to reduce the influence of the surrounding illumination. This method improves the contrast of the images by changing its gray levels [264]. Figures A.10, A.11 and A.12 show the histogram equalization results respectively for the AT&T, CBCL and Yale face images that are depicted in Figures A.2, A.3 and A.8.

Fig. A.10 AT&T face images after applying a histogram equalization to the original images presented in Figure A.2

Fig. A.11 CBCL face images after applying a histogram equalization to the original images presented in Figure A.3

Fig. A.12 Yale face images after applying a histogram equalization to the original images presented in Figure A.8

References

[1] Abramov, A., Kulvicius, T., Wörgötter, F., Dellen, B.: Real-time image segmentation on a GPU. In: Keller, R., Kramer, D., Weiss, J.-P. (eds.) Facing the Multicore-Challenge. LNCS, vol. 6310, pp. 131–142. Springer, Heidelberg (2010)

[2] Aha, D.W., Kibler, D., Albert, M.K.: Instance-based learning algorithms. Machine Learning 6(1), 37–66 (1991)

[3] Alavala, C.R.: Fuzzy Logic and Neural Networks: Basic Concepts & Applications. New Age International Pushishers (2008)

[4] Alexandre, L.A.: Single layer complex valued neural network with entropic cost function. In: Honkela, T. (ed.) ICANN 2011, Part I. LNCS, vol. 6791, pp. 331–338. Springer, Heidelberg (2011)

[5] Almeida, L.B.: C1.2 Multilayer perceptrons. In: Handbook of Neural Computation, pp. C1.2:1–C1.2:30. IOP Publishing Ltd., Oxford University Press (1997)

[6] Alpaydin, E.: Introduction to Machine Learning (Adaptive Computation and Machine Learning), 2nd edn. MIT Press (2010)

[7] Arora, M., Nath, S., Mazumdar, S., Baden, S.B., Tullsen, D.M.: Redefining the role of the CPU in the era of CPU-GPU integration. IEEE Micro 32(6), 4–16 (2012)

[8] Ayat, N.E., Cheriet, M., Remaki, L., Suen, C.Y.: KMOD - a new support vector machine kernel with moderate decreasing for pattern recognition. application to digit image recognition. In: Proceedings of the 6th International Conference on Document Analysis and Recognition, pp. 1215–1219 (2001)

[9] Ayat, N.E., Cheriet, M., Suen, C.Y.: KMOD - a two-parameter SVM kernel for pattern recognition. In: Proceedings of the 16th International Conference on Pattern Recognition, vol. 3, pp. 331–334 (2002)

[10] Ayuyev, V.V., Jupin, J., Harris, P.W., Obradovic, Z.: Dynamic clustering-based estimation of missing values in mixed type data. In: Pedersen, T.B., Mohania, M.K., Tjoa, A.M. (eds.) DaWaK 2009. LNCS, vol. 5691, pp. 366–377. Springer, Heidelberg (2009)

[11] Bache, K., Lichman, M.: UCI machine learning repository (2013), `http://archive.ics.uci.edu/ml`

[12] Bao, Y., Ishii, N., Du, X.-Y.: Combining multiple k-nearest neighbor classifiers using different distance functions. In: Yang, Z.R., Yin, H., Everson, R.M. (eds.) IDEAL 2004. LNCS, vol. 3177, pp. 634–641. Springer, Heidelberg (2004)

[13] Bekkerman, R., Bilenko, M., Langford, J. (eds.): Scaling Up Machine Learning: Parallel and Distributed Approaches. Cambridge University Press (2012)

[14] Bengio, Y.: Learning deep architectures for AI. Foundations and Trends in Machine Learning 2(1), 1–127 (2009)

[15] Beringer, J., Hüllermeier, E.: Efficient instance-based learning on data streams. Intelligent Data Analysis 11(6), 627–650 (2007)
[16] Bernhard, F., Keriven, R.: Spiking neurons on GPUs. In: Alexandrov, V.N., van Albada, G.D., Sloot, P.M.A., Dongarra, J. (eds.) ICCS 2006, Part IV. LNCS, vol. 3994, pp. 236–243. Springer, Heidelberg (2006)
[17] Bibi, S., Stamelos, I.: Selecting the appropriate machine learning techniques for the prediction of software development costs. In: Maglogiannis, I., Karpouzis, K., Bramer, M. (eds.) Artificial Intelligence Applications and Innovations. IFIP AICT, vol. 204, pp. 533–540. Springer, Heidelberg (2006)
[18] Bishop, C.M.: Pattern Recognition and Machine Learning. Springer (2006)
[19] Blackard, J.A., Dean, D.J.: Comparative accuracies of artificial neural networks and discriminant analysis in predicting forest cover types from cartographic variables. Computers and Electronics in Agriculture 24, 131–151 (1999)
[20] Bohn, C.-A.: Kohonen feature mapping through graphics hardware. In: Proceedings of the 1998 International Conference on Computational Intelligence and Neurosciences (ICCIN 1998), pp. 64–67 (1998)
[21] Bonet, I., Rodríguez, A., Grau, R., García, M.M., Saez, Y., Nowé, A.: Comparing distance measures with visual methods. In: Gelbukh, A., Morales, E.F. (eds.) MICAI 2008. LNCS (LNAI), vol. 5317, pp. 90–99. Springer, Heidelberg (2008)
[22] Bottou, L., Bousquet, O.: Learning using large datasets. In: Mining Massive DataSets for Security. NATO ASI Workshop Series. IOS Press (2008)
[23] De Brabanter, K., De Brabanter, J., Suykens, J.A.K., De Moor, B.: Optimized fixed-size kernel models for large data sets. Computational Statistics and Data Analysis 54(6), 1484–1504 (2010)
[24] Bramer, M.A.: Principles of data mining. Springer (2007)
[25] Brandstetter, A., Artusi, A.: Radial basis function networks GPU-based implementation. IEEE Transactions on Neural Networks 19(12), 2150–2154 (2008)
[26] Brennan, M., Greenstadt, R.: Coalescing Twitter trends: The under-utilization of machine learning in social media. In: 2011 IEEE International Conference on Privacy, Security, Risk, and Trust, and IEEE International Conference on Social Computing, pp. 641–646 (2011)
[27] Brunton, A., Shu, C., Roth, G.: Belief propagation on the GPU for stereo vision. In: Proceedings of the 3rd Canadian Conference on Computer and Robot Vision (CRV 2006), pp. 76–81. IEEE Computer Society (2006)
[28] Bucur, L., Florea, A.: Techniques for prediction in chaos: A comparative study on financial data. U.P.B. Scientific Bulletin, Series C 73(3), 17–32 (2011)
[29] Burges, C.J.C.: A tutorial on support vector machines for pattern recognition. Data Mining and Knowledge Discovery 2(2), 121–167 (1998)
[30] Campbell, A., Berglund, E., Streit, A.: Graphics hardware implementation of the parameter-less self-organising map. In: Gallagher, M., Hogan, J.P., Maire, F. (eds.) IDEAL 2005. LNCS, vol. 3578, pp. 343–350. Springer, Heidelberg (2005)
[31] Cano, A., Zafra, A., Ventura, S.: Speeding up the evaluation phase of GP classification algorithms on GPUs. Soft Computing 16(2), 187–202 (2012)
[32] Cao, L.J., Keerthi, S.S., Ong, C.-J., Zhang, J.Q., Periyathamby, U., Fu, X.J., Lee, H.P.: Parallel sequential minimal optimization for the training of support vector machines. IEEE Transactions on Neural Networks 17(4), 1039–1049 (2006)
[33] Carpenter, A.: CUSVM: A CUDA implementation of support vector classification and regression (2009), http://patternsonascreen.net/cuSVMDesc.pdf
[34] Carreira-Perpiñán, M.Á., Hinton, G.E.: On contrastive divergence learning. In: Proceedings of the 10th International Workshop on Artificial Intelligence and Statistics (AISTATS 2005), pp. 33–40 (2005)
[35] Catanzaro, B., Sundaram, N., Keutzer, K.: Fast support vector machine training and classification on graphics processors. In: Proceedings of the 25th International Conference on Machine Learning (ICML 2008), vol. 307, pp. 104–111. ACM (2008)

[36] Cavuoti, S., Garofalo, M., Brescia, M., Paolillo, M., Pescape', A., Longo, G., Ventre, G.: Astrophysical data mining with GPU. a case study: Genetic classification of globular clusters. New Astronomy 26, 12–22 (2014)
[37] Cavuoti, S., Garofalo, M., Brescia, M., Pescape', A., Longo, G., Ventre, G.: Genetic algorithm modeling with GPU parallel computing technology. In: Apolloni, B., Bassis, S., Esposito, A., Morabito, F.C. (eds.) Neural Nets and Surroundings. SIST, vol. 19, pp. 29–39. Springer, Heidelberg (2013)
[38] Cecilia, J.M., Nisbet, A., Amos, M., García, J.M., Ujaldón, M.: Enhancing GPU parallelism in nature-inspired algorithms. The Journal of Supercomputing 63(3), 773–789 (2013)
[39] Chacko, B.P., Krishnan, V.R.V., Anto, P.B.: Character recognition using multiple back propagation algorithm. In: Proceedings of the National Conference on Image Processing (2010)
[40] Chang, C.-C., Lin, C.-J.: LIBSVM: A library for support vector machines. ACM Transactions on Intelligent Systems and Technology 2(3), 1–27 (2011)
[41] Chapelle, O., Schölkopf, B., Zien, A.: Introduction to semi-supervised learning. In: Semi-Supervised Learning, ch. 1, pp. 1–14. MIT Press (2006)
[42] Che, S., Boyer, M., Meng, J., Tarjan, D., Sheaffer, J.W., Skadron, K.: A performance study of general-purpose applications on graphics processors using CUDA. Journal of Parallel and Distributed Computing 68(10), 1370–1380 (2008)
[43] Che, S., Li, J., Sheaffer, J.W., Skadron, K., Lach, J.: Accelerating compute-intensive applications with GPUs and FPGAs. In: Symposium on Application Specific Processors (SASP 2008), pp. 101–107 (2008b)
[44] Chellapilla, K., Puri, S., Simard, P.: High performance convolutional neural networks for document processing. In: Proceedings of the 10 International Workshop on Frontiers in Handwriting Recognition (2006)
[45] Chen, B., Zhao, S., Zhu, P., Prncipe, J.C.: Quantized kernel least mean square algorithm. IEEE Transactions on Neural Networks and Learning Systems 23(1), 22–32 (2012)
[46] Cheng, B.Y.M., Carbonell, J.G., Klein-Seetharaman, J.: Protein classification based on text document classification techniques. Proteins: Structure, Function, and Bioinformatics 58(4), 955–970 (2005)
[47] Cherkassky, V., Mulier, F.: Learning From Data: Concepts, Theory, and Methods, 2nd edn. John Wiley & Sons (2007)
[48] Chitty, D.M.: Fast parallel genetic programming: multi-core CPU versus many-core GPU. Soft Computing 16(10), 1795–1814 (2012)
[49] Clarke, B., Fokoué, E., Zhang, H.H.: Principles and Theory for Data Mining and Machine Learning. Springer (2009)
[50] Correia, D., Pereira, C., Verissimo, P., Dourado, A.: A platform for peptidase detection based on text mining techniques. In: International Symposium on Computational Intelligence for Engineering Systems (2011)
[51] Cortes, C., Vapnik, V.: Support-vector networks. Machine Learning 20(3), 273–297 (1995)
[52] Cumbley, R., Church, P.: Is "big data" creepy? Computer Law & Security Review 29(5), 601–609 (2013)
[53] Cybenko, G.: Approximation by superpositions of a sigmoidal function. Mathematics of Control, Signals, and Systems (MCSS) 2(4), 303–314 (1989)
[54] Desjardins, G., Courville, A., Bengio, Y., Vincent, P., Delalleau, O.: Tempered Markov Chain Monte Carlo for training of restricted Boltzmann machines. In: Proceedings of the 13th International Conference on Artificial Intelligence and Statistics, pp. 145–152 (2010)
[55] Do, T.-N., Nguyen, V.-H., Poulet, F.: Speed up SVM algorithm for massive classification tasks. In: Tang, C., Ling, C.X., Zhou, X., Cercone, N.J., Li, X. (eds.) ADMA 2008. LNCS (LNAI), vol. 5139, pp. 147–157. Springer, Heidelberg (2008)

[56] Doerr, B., Fouz, M., Friedrich, T.: Why rumors spread so quickly in social networks. Communications of the ACM 55(6), 70–75 (2012)
[57] Duch, W., Jankowski, N.: Survey of neural transfer functions. Neural Computing Surveys 2, 163–213 (1999)
[58] Džeroski, S., Panov, P., Ženko, B.: Machine learning, ensemble methods in. In: Encyclopedia of Complexity and Systems Science, pp. 5317–5325. Springer (2009)
[59] Van Essen, B., Macaraeg, C., Gokhale, M., Prenger, R.: Accelerating a random forest classifier: Multi-core, GP-GPU, or FPGA? In: IEEE 20th Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM 2012), pp. 232–239 (2012)
[60] Fahlman, S.E., Lebiere, C.: The cascade-correlation learning architecture. In: Advances in Neural Information Processing Systems, vol. 2, pp. 524–532 (1990)
[61] Fan, R.-E., Chen, P.-H., Lin, C.-J.: Working set selection using second order information for training support vector machines. Journal of Machine Learning Research 6, 1889–1918 (2005)
[62] Fernando, R., Kilgard, M.J.: The Cg Tutorial: The Definitive Guide to Programmable Real-Time Graphics. Addison-Wesley Professional (2003)
[63] Fischer, A., Igel, C.: Training restricted Boltzmann machines: An introduction. Pattern Recognition (2013)
[64] Funahashi, K.: On the approximate realization of continuous mappings by neural networks. Neural Networks 2(3), 183–192 (1989)
[65] Gama, J., Medas, P., Rodrigues, P.: Concept drift in decision trees learning from data streams. In: European Symposium on Intelligent Technologies Hybrid Systems and their Implementation on Smart Adaptive Systems Eunite 2004, pp. 218–225 (2004)
[66] Gama, J., Sebastião, R., Rodrigues, P.: Issues in evaluation of stream learning algorithms. In: Proceedings of the 15th ACM SIGKDD International Conference on KnowledgeDiscovery and Data Mining (KDD 2009), pp. 329–338 (2009)
[67] Garcia, V., Debreuve, E., Barlaud, M.: Fast k nearest neighbor search using GPU. In: Proceedings of the 2008 IEEE Computer Society Conference on Computer Vision and Pattern Recognition Workshops (CVPRW 2008), pp. 1–6 (2008)
[68] García-Laencina,
P.J., Sancho-Gómez, J.-L., Figueiras-Vidal, A.R.: Pattern classification with missing data: a review. Neural Computing and Applications 19(2), 263–282 (2010)
[69] García-Pedrajas, N., Del Castillo, J.A.R., Ortiz-Boyer, D.: A cooperative coevolutionary algorithm for instance selection for instance-based learning. Machine Learning 78(3), 381–420 (2010)
[70] Garg, V.K., Murty, M.N.: Feature subspace SVMs (FS-SVMs) for high dimensional handwritten digit recognition. International Journal of Data Mining, Modelling and Management (IJDMMM) 1(4), 411–436 (2009)
[71] Garland, M., Kirk, D.B.: Understanding throughput-oriented architectures. Communications of the ACM 53(11), 58–66 (2010)
[72] Giannesini, F., Saux, L.B.: GPU-accelerated one-class SVM for exploration of remote sensing data. In: IEEE International Geoscience and Remote Sensing Symposium (IGARSS 2012), pp. 7349–7352 (2012)
[73] Gillis, N., Glineur, F.: Using under-approximations for sparse nonnegative matrix factorization. Pattern Recogntion 43(4), 1676–1687 (2010)
[74] Gonçalves, J., Lopes, N., Ribeiro, B.: Multi-threaded support vector machines for pattern recognition. In: Huang, T., Zeng, Z., Li, C., Leung, C.S. (eds.) ICONIP 2012, Part II. LNCS, vol. 7664, pp. 616–623. Springer, Heidelberg (2012)
[75] Gonçalves, J.: Development of support vector machines (SVMs) in graphics processing units for object recognition. Master's thesis, University of Coimbra (2012)
[76] Granmo, O.-C.: Short-term forecasting of electricity consumption using gaussian processes. Master's thesis, University of Agder (2012)

[77] Grauer-Gray, S., Kambhamettu, C., Palaniappan, K.: GPU implementation of belief propagation using CUDA for cloud tracking and reconstruction. In: Proceedings of the 5th IAPR Workshop on Pattern Recognition in Remote Sensing (PRRS 2008), pp. 1–4 (2008)
[78] Guzhva, A., Dolenko, S., Persiantsev, I.: Multifold acceleration of neural network computations using GPU. In: Alippi, C., Polycarpou, M., Panayiotou, C., Ellinas, G. (eds.) ICANN 2009, Part I. LNCS, vol. 5768, pp. 373–380. Springer, Heidelberg (2009)
[79] Halfhill, T.R.: Looking beyond graphics. Technical report, In-Stat (2009)
[80] Hall, M., Frank, E., Holmes, G., Pfahringer, B., Reutemann, P., Witten, I.H.: The WEKA data mining software: an update. SIGKDD Explorations Newsletter 11(1), 10–18 (2009)
[81] Harding, S., Banzhaf, W.: Fast genetic programming on GPUs. In: Ebner, M., O'Neill, M., Ekárt, A., Vanneschi, L., Esparcia-Alcázar, A.I. (eds.) EuroGP 2007. LNCS, vol. 4445, pp. 90–101. Springer, Heidelberg (2007)
[82] Haykin, S.: Neural Networks: A Comprehensive Foundation, 2nd edn. Prentice Hall (1998)
[83] Herrero-Lopez, S.: Accelerating SVMs by integrating GPUs into mapreduce clusters. In: IEEE International Conference on Systems, Man, and Cybernetics (SMC), pp. 1298–1305 (2011)
[84] Herrero-Lopez, S., Williams, J.R., Sanchez, A.: Parallel multiclass classification using SVMs on GPUs. In: Proceedings of the 3rd Workshop on General-Purpose Computation on Graphics Processing Units, pp. 2–11 (2010)
[85] Hey, T., Tansley, S., Tolle, K. (eds.): The Fourth Paradigm: Data-Intensive Scientific Discovery. Microsoft Research (2009)
[86] Hinton, G.E.: Training products of experts by minimizing contrastive divergence. Neural Computation 14(8), 1771–1800 (2002) ISSN 0899-7667
[87] Hinton, G.E.: A practical guide to training restricted Boltzmann machines. Technical report, Department of Computer Science, University of Toronto (2010)
[88] Hinton, G.E., Osindero, S., Teh, Y.-W.: A fast learning algorithm for deep belief nets. Neural Computation 18(7), 1527–1554 (2006) ISSN 0899-7667
[89] Hirose, A. (ed.): Complex-Valued Neural Networks: Advances and Applications. John Wiley & Sons (2013)
[90] Hoegaerts, L., Suykens, J.A.K., Vandewalle, J., De Moor, B.: Subset based least squares subspace regression in RKHS. Neurocomputing 63, 293–323 (2005)
[91] Hornik, K., Stinchcombe, M., White, H.: Multilayer feedforward networks are universal approximators. Neural Networks 2(5), 359–366 (1989)
[92] Hse, H., Newton, A.R.: Sketched symbol recognition using Zernike moments. In: Proceedings of the 17th International Conference on Pattern Recognition, vol. 1, pp. 367–370 (2004)
[93] Hua, S., Sun, Z.: Support vector machine approach for protein subcellular localization prediction. Bioinformatics 17(8), 721–728 (2001)
[94] Hui, C.-L. (ed.): Artificial Neural Networks - Application. InTech (2011)
[95] Hung, Y., Wang, W.: Accelerating parallel particle swarm optimization via GPU. Optimization Methods and Software 27(1), 33–51 (2012)
[96] Jain, S., Lange, S., Zilles, S.: Towards a better understanding of incremental learning. In: Balcázar, J.L., Long, P.M., Stephan, F. (eds.) ALT 2006. LNCS (LNAI), vol. 4264, pp. 169–183. Springer, Heidelberg (2006)
[97] Jang, H., Park, A., Jung, K.: Neural network implementation using CUDA and OpenMP. In: Proceedings of the 2008 Digital Image Computing: Techniques and Applications (DICTA 2008), pp. 155–161 (2008) ISBN 978-0-7695-3456-5
[98] Jans, M., Lybaert, N., Vanhoof, K.: A framework for internal fraud risk reduction at IT integrating business processes: The IFR^2 framework. The International Journal of Digital Accounting Research 9, 1–29 (2009)

[99] Jian, L., Wang, C., Liu, Y., Liang, S., Yi, W., Shi, Y.: Parallel data mining techniques on graphics processing unit with compute unified device architecture (CUDA). The Journal of Supercomputing 64(3), 942–967 (2013)
[100] Jowell, R., and the Central Coordinating Team. European social survey 2002/2003; 2004/2005; 2006/2007, 2003, 2005, 2007
[101] Kanwisher, N.: Functional specificity in the human brain: a window into the functional architecture of the mind. Proceedings of the National Academy of Sciences of the United States of America 107(25), 11163–11170 (2010), http://eutils.ncbi.nlm.nih.gov/entrez/eutils/elink.fcgi?dbfrom=pubmed&id=20484679&retmode=ref&cmd=prlinks
[102] Karhunen, J.: Robust PCA methods for complete and missing data. Neural Network World 21(5), 357–392 (2011)
[103] Keerthi, S.S., Shevade, S.K., Bhattacharyya, C., Murthy, K.R.K.: Improvements to Platt's SMO algorithm for SVM classifier design. Neural Computation 13(3), 637–649 (2001)
[104] King, D.E.: Dlib-ml: A machine learning toolkit. Journal of Machine Learning Research 10, 1755–1758 (2009)
[105] Kotsiantis, S.B., Zaharakis, I.D., Pintelas, P.E.: Machine learning: a review of classification and combining techniques. Artificial Intelligence Review 26(3), 159–190 (2006)
[106] Kumar, S., Kumawat, T., Marwal, N.K., Singh, B.K.: Artificial neural network and its applications. International Journal of Computer Science and Management Research 2(2), 1621–1626 (2013)
[107] Lahabar, S., Agrawal, P., Narayanan, P.J.: High performance pattern recognition on GPU. In: Proceedings of the 2008 National Conference on Computer Vision Pattern Recognition Image Processing and Graphics, pp. 154–159 (2008)
[108] Langdon, W.B.: Graphics processing units and genetic programming: an overview. Soft Computing 15(8), 1657–1669 (2011)
[109] Langdon, W.B., Banzhaf, W.: A SIMD interpreter for genetic programming on GPU graphics cards. In: O'Neill, M., Vanneschi, L., Gustafson, S., Esparcia Alcázar, A.I., De Falco, I., Della Cioppa, A., Tarantino, E. (eds.) EuroGP 2008. LNCS, vol. 4971, pp. 73–85. Springer, Heidelberg (2008)
[110] Larochelle, H., Erhan, D., Courville, A., Bergstra, J., Bengio, Y.: An empirical evaluation of deep architectures on problems with many factors of variation. In: Proceedings of the 24th International Conference on Machine Learning (ICML 2007), pp. 473–480 (2007) ISBN 978-1-59593-793-3
[111] Lee, D.D., Seung, H.S.: Learning the parts of objects by non-negative matrix factorization. Nature 401, 788–791 (1999)
[112] Lee, D.D., Seung, H.S.: Algorithms for non-negative matrix factorization. In: Advances in Neural Information Processing Systems (NIPS 2000), pp. 556–562. MIT Press (2000)
[113] Lee, H., Grosse, R., Ranganath, R., Ng, A.Y.: Convolutional deep belief networks for scalable unsupervised learning of hierarchical representations. In: Proceedings of the 26th Annual International Conference on Machine Learning (ICML 2009), pp. 609–616. ACM (2009) ISBN 978-1-60558-516-1
[114] Li, Q., Salman, R., Test, E., Strack, R., Kecman, V.: Parallel multitask cross validation for support vector machine using GPU. Journal of Parallel and Distributed Computing 73(3), 293–302 (2013)
[115] Li, Z., Wu, X., Peng, H.: Nonnegative matrix factorization on orthogonal subspace. Pattern Recogntion Letters 31(9), 905–911 (2010)
[116] Lim, E.A., Zainuddin, Z.: A comparative study of missing value estimation methods: Which method performs better? In: Proceedings of the International Conference on Electronic Design (ICED 2008), pp. 1–5 (2008)

[117] Lin, J., Kolcz, A.: Large-scale machine learning at Twitter. In: Proceedings of the 2012 ACM SIGMOD International Conference on Management of Data, pp. 793–804. ACM (2012)
[118] Lin, T.-K., Chien, S.-Y.: Support vector machines on GPU with sparse matrix format. In: Proceedings of the 9th International Conference on Machine Learning and Applications (ICMLA 2010), pp. 313–318. IEEE Computer Society (2010)
[119] Little, R.J.A., Rubin, D.B.: Statistical analysis with missing data, 2nd edn. Wiley (2002)
[120] Lopes, N., Ribeiro, B.: A data pre-processing tool for neural networks (DTPNN) use in a moulding injection machine. In: Second World Manufacturing Congress, WMC 1999 (1999)
[121] Lopes, N., Ribeiro, B.: Hybrid learning in a multi-neural network architecture. In: INNS-IEEE International Joint Conference on Neural Networks (IJCNN 2001), vol. 4, pp. 2788–2793 (2001)
[122] Lopes, N., Ribeiro, B.: An efficient gradient-based learning algorithm applied to neural networks with selective actuation neurons. Neural, Parallel and Scientific Computations 11, 253–272 (2003)
[123] Lopes, N., Ribeiro, B.: Fast pattern classification of ventricular arrhythmias using graphics processing units. In: Bayro-Corrochano, E., Eklundh, J.-O. (eds.) CIARP 2009. LNCS, vol. 5856, pp. 603–610. Springer, Heidelberg (2009)
[124] Lopes, N., Ribeiro, B.: GPU implementation of the multiple back-propagation algorithm. In: Corchado, E., Yin, H. (eds.) IDEAL 2009. LNCS, vol. 5788, pp. 449–456. Springer, Heidelberg (2009)
[125] Lopes, N., Ribeiro, B.: MBPGPU: A supervised pattern classifier for graphical processing units. In: Portuguese Conference on Pattern Recognition (RECPAD 2009), 15th edn. (2009c)
[126] Lopes, N., Ribeiro, B.: A hybrid face recognition approach using GPUMLib. In: Bloch, I., Cesar Jr., R.M. (eds.) CIARP 2010. LNCS, vol. 6419, pp. 96–103. Springer, Heidelberg (2010)
[127] Lopes, N., Ribeiro, B.: A strategy for dealing with missing values by using selective activation neurons in a multi-topology framework. In: IEEE International Joint Conference on Neural Networks, IJCNN 2010 (2010b)
[128] Lopes, N., Ribeiro, B.: Non-negative matrix factorization implementation using graphic processing units. In: Fyfe, C., Tino, P., Charles, D., Garcia-Osorio, C., Yin, H. (eds.) IDEAL 2010. LNCS, vol. 6283, pp. 275–283. Springer, Heidelberg (2010)
[129] Lopes, N., Ribeiro, B.: GPUMLib: An efficient open-source GPU machine learning library. International Journal of Computer Information Systems and Industrial Management Applications 3, 355–362 (2011a)
[130] Lopes, N., Ribeiro, B.: A robust learning model for dealing with missing values in many-core architectures. In: Dobnikar, A., Lotrič, U., Šter, B. (eds.) ICANNGA 2011, Part II. LNCS, vol. 6594, pp. 108–117. Springer, Heidelberg (2011b)
[131] Lopes, N., Ribeiro, B.: An incremental class boundary preserving hypersphere classifier. In: Lu, B.-L., Zhang, L., Kwok, J. (eds.) ICONIP 2011, Part II. LNCS, vol. 7063, pp. 690–699. Springer, Heidelberg (2011)
[132] Lopes, N., Ribeiro, B.: A fast optimized semi-supervised non-negative matrix factorization algorithm. In: IEEE International Joint Conference on Neural Networks (IJCNN 2011), pp. 2495–2500 (2011)
[133] Lopes, N., Ribeiro, B.: An evaluation of multiple feed-forward networks on GPUs. International Journal of Neural Systems (IJNS) 21(1), 31–47 (2011)
[134] Lopes, N., Ribeiro, B.: Incremental learning for non-stationary patterns. In: Portuguese Conference on Pattern Recognition (RECPAD 2011), 17th edn. (2011f)
[135] Lopes, N., Ribeiro, B.: Improving convergence of restricted Boltzmann machines via a learning adaptive step size. In: Alvarez, L., Mejail, M., Gomez, L., Jacobo, J. (eds.) CIARP 2012. LNCS, vol. 7441, pp. 511–518. Springer, Heidelberg (2012)

[136] Lopes, N., Ribeiro, B.: Towards a hybrid NMF-based neural approach for face recognition on GPUs. International Journal of Data Mining, Modelling and Management (IJDMMM) 4(2), 138–155 (2012)
[137] Lopes, N., Ribeiro, B.: Handling missing values via a neural selective input model. Neural Network World 22(4), 357–370 (2012)
[138] Lopes, N., Ribeiro, B.: Towards adaptive learning with improved convergence of deep belief networks on graphics processing units. Pattern Recognition (2013), http://dx.doi.org/10.1016/j.patcog.2013.06.029
[139] Lopes, N., Ribeiro, B., Quintas, R.: GPUMLib: A new library to combine machine learning algorithms with graphics processing units. In: 10th International Conference on Hybrid Intelligent Systems (HIS 2010), pp. 229–232 (2010)
[140] Lopes, N., Correia, D., Pereira, C., Ribeiro, B., Dourado, A.: An incremental hypersphere learning framework for protein membership prediction. In: Corchado, E., Snášel, V., Abraham, A., Woźniak, M., Graña, M., Cho, S.-B. (eds.) HAIS 2012, Part III. LNCS, vol. 7208, pp. 429–439. Springer, Heidelberg (2012a)
[141] Lopes, N., Ribeiro, B., Gonçalves, J.: Restricted Boltzmann machines and deep belief networks on multi-core processors. In: IEEE International Joint Conference on Neural Networks, IJCNN 2012 (2012b), doi:10.1109/IJCNN.2012.6252431
[142] López-Molina, T., Pérez-Méndez, A., Rivas-Echeverría, F.: Missing values imputation techniques for neural networks patterns. In: Proceedings of the 12th WSEAS International Conference on Systems (ICS 2008), pp. 290–295 (2008)
[143] Luo, Z., Liu, H., Wu, X.: Artificial neural network computation on graphic process unit. In: Proceedings of the 2005 IEEE International Joint Conference on Neural Networks (IJCNN 2005), vol. 1, pp. 622–626 (2005)
[144] Lyman, P., Varian, H.R., Swearingen, K., Charles, P., Good, N., Jordan, L.L., Pal, J.: How much information (2003), http://www.sims.berkeley.edu/how-much-info-2003
[145] Ma, K.-L., Muelder, C.W.: Large-scale graph visualization and analytics. Computer 46(7), 39–46 (2013)
[146] Manyika, J., Chui, M., Brown, B., Bughin, J., Dobbs, R., Roxburgh, C., Byers, A.H.: Big data: The next frontier for innovation, competition, and productivity. Technical report, McKinsey Global Institute (2011)
[147] Markey, M.K., Tourassi, G.D., Margolis, M., DeLong, D.M.: Impact of missing data in evaluating artificial neural networks trained on complete data. Computers in Biology and Medicine 36(5), 516–525 (2006)
[148] Markoff, J.: Giant steps in teaching computers to think like us: 'neural nets' mimic the ways human minds listen, see and execute. International Herald Tribune 24-25, 1–8 (2012)
[149] Marques, A.: Feature extraction and PVC detection using neural networks and support vector machines. Master's thesis, University of Coimbra (2007)
[150] Martínez-Zarzuela, M., Díaz Pernas, F.J., Díez Higuera, J.F., Rodríguez, M.A.: Fuzzy ART neural network parallel computing on the GPU. In: Sandoval, F., Prieto, A.G., Cabestany, J., Graña, M. (eds.) IWANN 2007. LNCS, vol. 4507, pp. 463–470. Springer, Heidelberg (2007)
[151] Masud, M.M., Chen, Q., Khan, L., Aggarwal, C.C., Gao, J., Han, J., Thuraisingham, B.M.: Addressing concept-evolution in concept-drifting data streams. In: Proceedings of the 10th IEEE International Conference on Data Mining (ICDM 2010), pp. 929–934 (2010)

[152] Mejía-Roa, E., García, C., Gómez, J.I., Prieto, M., Tirado, F., Nogales, R., Pascual-Montano, A.: Biclustering and classification analysis in gene expression using nonnegative matrix factorization on multi-GPU systems. In: 11th International Conference on Intelligent Systems Design and Applications, pp. 882–887 (2011)
[153] Mjolsness, E., DeCoste, D.: Machine learning for science: State of the art and future prospects. Science 293(5537), 2051–2055 (2001)
[154] Mockus, A.: Missing Data in Software Engineering. In: Guide to Advanced Empirical Software Engineering, ch. 7, pp. 185–200. Springer (2008)
[155] Moens, M.-F.: Information Extraction: Algorithms and Prospects in a Retrieval Context. The Information Retrieval Series. Springer (2006)
[156] Morgado, L., Pereira, C., Veríssimo, P., Dourado, A.: A support vector machine based framework for protein membership prediction. In: Computational Intelligence for Engineering Systems. Intelligent Systems, Control and Automation: Science and Engineering, vol. 46, pp. 90–103. Springer (2011)
[157] Munakata, T.: Fundamentals of the New Artificial Intelligence: Neural, Evolutionary, Fuzzy and More (Texts in Computer Science), 2nd edn. Springer (2008)
[158] Murzin, A.G., Brenner, S.E., Hubbard, T., Chothia, C.: SCOP: a structural classification of proteins database for the investigation of sequences and structures. Journal of Molecular Biology 247(4), 536–540 (1995)
[159] Nageswaran, J.M., Dutt, N., Krichmar, J.L., Nicolau, A., Veidenbaum, A.V.: A configurable simulation environment for the efficient simulation of large-scale spiking neural networks on graphics processors. Neural Networks 22(5-6), 791–800 (2009) ISSN 0893-6080
[160] Nelwamondo, F.V., Mohamed, S., Marwala, T.: Missing data: A comparison of neural network and expectation maximization techniques. Current Science 93(11), 1514–1521 (2007)
[161] Nitta, T.: Local minima in hierarchical structures of complex-valued neural networks. Neural Networks 43, 1–7 (2013)
[162] NVIDIA. NVIDIA's next generation CUDA compute architecture: Fermi (2009)
[163] NVIDIA. CUDA C best practices guide: Design guide (January 2012)
[164] NVIDIA. NVIDIA CUDA C programming guide: Version 4.2 (2012)
[165] O'Connor, B., Balasubramanyan, R., Routledge, B.R., Smith, N.A.: From tweets to polls: Linking text sentiment to public opinion time series. In: Proceedings of the International AAAI Conference on Weblogs and Social Media (2010)
[166] Oh, K.-S., Jung, K.: GPU implementation of neural networks. Pattern Recognition 37(6), 1311–1314 (2004) ISSN 0031-3203
[167] Olvera-López, J.A., Carrasco-Ochoa, J.A., Martínez-Trinidad, J.F., Kittler, J.: A review of instance selection methods. Artificial Intelligence Review 34(2), 133–143 (2010)
[168] Osuna, E., Freund, R., Girosi, F.: An improved training algorithm for support vector machines. In: Proceedings of the 1997 IEEE Neural Networks in Signal Processing, pp. 276–285. IEEE Computer Society (1997)
[169] Osuna, E.E., Freund, R., Girosi, F.: Support vector machines: Training and applications. Technical report, Massachusetts Institute of Technology (1997)
[170] Owens, J.D., Luebke, D., Govindaraju, N., Harris, M., Krüger, J., Lefohn, A., Purcell, T.J.: A survey of general-purpose computation on graphics hardware. Computer Graphics Forum 26(1), 80–113 (2007)
[171] Owens, J.D., Houston, M., Luebke, D., Green, S., Stone, J.E., Phillips, J.C.: GPU computing. Proceedings of the IEEE 96(5), 879–899 (2008)
[172] Pappa, G.L., Freitas, A.: Automating the Design of Data Mining Algorithms: An Evolutionary Computation Approach. Natural Computing Series. Springer (2010)
[173] Pereira, C., Morgado, L., Correia, D., Verissimo, P., Dourado, A.: Kernel machines for proteomics data analysis: Algorithms and tools. Presented at the European Network for Business and Industrial Statistics, Coimbra, Portugal (2011)

[174] Piękniewski, F., Rybicki, L.: Visual comparison of performance for different activation functions in MLP networks. In: IEEE International Joint Conference on Neural Networks (IJCNN 2004), vol. 4, pp. 2947–2952 (2004)
[175] Platt, J.C.: Sequential minimal optimization: A fast algorithm for training support vector machines. Technical report, Microsoft Research (1998)
[176] Popescu, A.-M., Pennacchiotti, M.: Detecting controversial events from Twitter. In: Proceedings of the 19th ACM International Conference on Information and Knowledge Management, pp. 1873–1876. ACM (2010)
[177] Pratt, K.B., Tschapek, G.: Visualizing concept drift. In: Proceedings of the 9th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining, pp. 735–740. ACM (2003)
[178] Qiao, M., Sung, A.H., Liu, Q.: Feature mining and intelligent computing for MP3 steganalysis. In: Proceedings of the International Joint Conference on Bioinformatics, Systems Biology and Intelligent Computing, pp. 627–630. IEEE Computer Society (2009)
[179] Quintas, R.: GPU implementation of RBF neural networks in audio steganalysis. Master's thesis, University of Coimbra (2010)
[180] Raina, R., Madhavan, A., Ng, A.Y.: Large-scale deep unsupervised learning using graphics processors. In: Proceedings of the 26th Annual International Conference on Machine Learning (ICML 2009), vol. 382, pp. 873–880. ACM (2009)
[181] Rajaraman, A., Leskovec, J., Ullman, J.D.: Mining of Massive Datasets. Cambridge University Press (2014)
[182] Ranzato, M., Boureau, Y.-L., LeCun, Y.: Sparse feature learning for deep belief networks. In: Advances in Neural Information Processing Systems (NIPS 2007), vol. 20, pp. 1185–1192 (2007)
[183] Rätsch, G., Sonnenburg, S., Schäfer, C.: Learning interpretable SVMs for biological sequence classification. BMC Bioinformatics 7(S-1) (2006)
[184] Rawlings, N.D., Barrett, A.J., Bateman, A.: MEROPS: the peptidase database. Nucleic Acids Research 38, 227–233 (2010)
[185] Refaeilzadeh, P., Tang, L., Liu, H.: Cross-validation. In: Encyclopedia of Database Systems, pp. 532–538. Springer (2009)
[186] Reinartz, T.: A unifying view on instance selection. Data Mining and Knowledge Discovery 6(2), 191–210 (2002)
[187] Ribeiro, B., Marques, A., Henriques, J., Antunes, M.: Choosing real-time predictors for ventricular arrhythmia detection. International Journal of Pattern Recognition and Artificial Intelligence (IJPRAI) 21(8), 1249–1263 (2007)
[188] Ribeiro, B., Silva, C., Vieira, A., Neves, J.: Extracting discriminative features using non-negative matrix factorization in financial distress data. In: Kolehmainen, M., Toivanen, P., Beliczynski, B. (eds.) ICANNGA 2009. LNCS, vol. 5495, pp. 537–547. Springer, Heidelberg (2009)
[189] Ribeiro, B., Lopes, N., Silva, C.: High-performance bankruptcy prediction model using graphics processing units. In: IEEE World Congress on Computational Intelligence, WCCI 2010 (2010)
[190] Richtárik, P., Takác, M., Ahipasaoglu, S.D.: Alternating maximization: Unifying framework for 8 sparse PCA formulations and efficient parallel codes. Cornell Universty Library (2012)
[191] Robilliard, D., Marion-Poty, V., Fonlupt, C.: Genetic programming on graphics processing units. Genetic Programming and Evolvable Machines 10(4), 447–471 (2009)
[192] Roux, N.L., Bengio, Y.: Representational power of restricted Boltzmann machines and deep belief networks. Neural Computation 20(6), 1631–1649 (2008)
[193] Roux, N.L., Bengio, Y.: Deep belief networks are compact universal approximators. Neural Computation 22(8), 2192–2207 (2010)

[194] Ryoo, S., Rodrigues, C.I., Baghsorkhi, S.S., Stone, S.S., Kirk, D.B., Hwu, W.W.: Optimization principles and application performance evaluation of a multithreaded GPU using CUDA. In: Proceedings of the 13th ACM Symposium on Principles and practice of parallel programming (PPoPP 2008), pp. 73–82 (2008)
[195] Sakaki, T., Okazaki, M., Matsuo, Y.: Earthquake shakes Twitter users: Real-time event detection by social sensors. In: Proceedings of the 19th International Conference on World Wide Web, pp. 851–860. ACM (2010)
[196] Samarasinghe, S.: Neural Networks for Applied Sciences and Engineering: From Fundamentals to Complex Pattern Recognition. Auerbach Publications (2007)
[197] Schaa, D., Kaeli, D.: Exploring the multiple-GPU design space. In: Proceedings of the 2009 IEEE International Symposium on Parallel & Distributed Processing (IPDPS 2009), pp. 1–12. IEEE Computer Society (2009)
[198] Schadt, E.E., Linderman, M.D., Sorenson, J., Lee, L., Nolan, G.P.: Cloud and heterogeneous computing solutions exist today for the emerging big data problems in biology. Nature Reviews Genetics 12(3) (2011)
[199] Schafer, J.L.: Norm: Multiple imputation of incomplete multivariate data under a normal model, version 2 (1999), `http://www.stat.psu.edu/~jls/misoftwa.html`
[200] Schafer, J.L., Graham, J.W.: Missing data: our view of the state of the art. Psychological Methods 7(2), 147–177 (2002)
[201] Schölkopf, B., Mika, S., Burges, C.J.C., Knirsch, P., Müller, K.-R., Rätsch, G., Smola, A.: Input space vs. feature space in kernel-based methods. IEEE Transactions on Neural Networks 10, 1000–1017 (1999)
[202] Schulz, H., Müller, A., Behnke, S.: Investigating convergence of restricted boltzmann machine learning. In: NIPS 2010 Workshop on Deep Learning and Unsupervised Feature Learning, Whistler, Canada (2010)
[203] Serpen, G.: A heuristic and its mathematical analogue within artificial neural network adaptation context. Neural Network World 15(2), 129–136 (2005)
[204] Shalom, S.A.A., Dash, M., Tue, M.: Efficient k-means clustering using accelerated graphics processors. In: Song, I.-Y., Eder, J., Nguyen, T.M. (eds.) DaWaK 2008. LNCS, vol. 5182, pp. 166–175. Springer, Heidelberg (2008)
[205] Sharp, T.: Implementing decision trees and forests on a GPU. In: Forsyth, D., Torr, P., Zisserman, A. (eds.) ECCV 2008, Part IV. LNCS, vol. 5305, pp. 595–608. Springer, Heidelberg (2008)
[206] Shawe-Taylor, J., Sun, S.: A review of optimization methodologies in support vector machines. Neurocomputing 74(17), 3609–3618 (2011)
[207] Shawe-Taylor, J., Bartlett, P.L., Williamson, R.C., Anthony, M.: Structural risk minimization over data-dependent hierarchies. IEEE Transactions on Information Theory 44(5), 1926–1940 (1998)
[208] She, R., Chen, F., Wang, K., Ester, M., Gardy, J.L., Brinkman, F.S.L.: Frequent-subsequence-based prediction of outer membrane proteins. In: Proceedings of the 9th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining (KDD 2003), pp. 436–445 (2003)
[209] Shenouda, E.A.M.A.: A quantitative comparison of different MLP activation functions in classification. In: Wang, J., Yi, Z., Zurada, J.M., Lu, B.-L., Yin, H. (eds.) ISNN 2006. LNCS, vol. 3971, pp. 849–857. Springer, Heidelberg (2006), `http://dx.doi.org/10.1007/11759966_125`
[210] Silva, M., Moutinho, L., Coelho, A., Marques, A.: Market orientation and performance: modelling a neural network. European Journal of Marketing 43(3/4), 421–437 (2009)
[211] Skala, M.A.: Aspects of metric spaces in computation. PhD thesis, University of Waterloo (2008)
[212] Smola, A.J., Bartlett, P., Schölkopf, B., Schuurmans, D. (eds.): Advances in Large Margin Classifiers. MIT Press (2000)

[213] Sokolova, M., Lapalme, G.: A systematic analysis of performance measures for classification tasks. Information Processing & Management 45(4), 427–437 (2009)
[214] Somorjai, R.L., Dolenko, B., Nikulin, A., Roberson, W., Thiessen, N.: Class proximity measures – dissimilarity-based classification and display of high-dimensional data. Journal of Biomedical Informatics 44(5), 775–788 (2011)
[215] Sonnenburg, S., Braun, M.L., Ong, C.S., Bengio, S., Bottou, L., Holmes, G., LeCun, Y., Müller, K.-R., Pereira, F., Rasmussen, C.E., Rätsch, G., Schölkopf, B., Smola, A., Vincent, P., Weston, J., Williamson, R.C.: The need for open source software in machine learning. Journal of Machine Learning Research 8, 2443–2466 (2007)
[216] Stamatopoulos, C., Chuang, T.Y., Fraser, C.S., Lu, Y.Y.: Fully automated image orientation in the absence of targets. In: International Archives of the Photogrammetry, Remote Sensing and Spatial Information Sciences (XXII ISPRS Congress), vol. XXXIX-B5, pp. 303–308 (2012)
[217] Steinkraus, D., Buck, I., Simard, P.Y.: Using GPUs for machine learning algorithms. In: Proceedings of the 2005 Eight International Conference on Document Analysis and Recognition (ICDAR 2005), vol. 2, pp. 1115–1120 (2005)
[218] Steinwart, I., Hush, D., Scovel, C.: A classification framework for anomaly detection. Journal of Machine Learning Research 6, 211–232 (2005)
[219] Swersky, K., Chen, B., Marlin, B., de Freitas, N.: A tutorial on stochastic approximation algorithms for training restricted Boltzmann machines and deep belief nets. In: Information Theory and Applications Workshop, pp. 1–10 (2010)
[220] Sylla, Y., Morizet-Mahoudeaux, P., Brobst, S.: Fraud detection on large scale social networks. In: 2013 IEEE International Congress on Big Data, pp. 413–414 (2013)
[221] Tahir, M.A., Smith, J.: Creating diverse nearest-neighbour ensembles using simultaneous metaheuristic feature selection. Pattern Recognition Letters 31(11), 1470–1480 (2010)
[222] Tang, H.-M., Lyu, M.R., King, I.: Face recognition committee machine. In: Proceedings of the 2003 IEEE International Conference on Acoustics, Speech, and Signal Processing (ICASSP 2003), vol. 2, pp. 837–840 (2003)
[223] Tang, H., Tan, K.C., Yi, Z.: Neural Networks: Computational Models and Applications. SCI, vol. 53. Springer, Heidelberg (2007)
[224] Tantipathananandh, C., Berger-Wolf, T.Y.: Finding communities in dynamic social networks. In: IEEE 11th International Conference on Data Mining (ICDM 2011), pp. 1236–1241 (2011)
[225] Tieleman, T.: Training restricted Boltzmann machines using approximations to the likelihood gradient. In: Proceedings of the 25th International Conference on Machine Learning (ICML 2008), pp. 1064–1071 (2008)
[226] Trebatický, P., Pospíchal, J.: Neural network training with extended kalman filter using graphics processing unit. In: Kůrková, V., Neruda, R., Koutník, J. (eds.) ICANN 2008, Part II. LNCS, vol. 5164, pp. 198–207. Springer, Heidelberg (2008)
[227] Treiber, M., Schall, D., Dustdar, S., Scherling, C.: Tweetflows: Flexible workflows with Twitter. In: Proceedings of the 3rd International Workshop on Principles of Engineering Service-Oriented Systems, pp. 1–7. ACM (2011)
[228] Tuikkala, J., Elo, L.L., Nevalainen, O.S., Aittokallio, T.: Missing value imputation improves clustering and interpretation of gene expression microarray data. BMC Bioinformatics 9(202), 1–14 (2008)
[229] Vapnik, V.: Estimation of Dependences Based on Empirical Data. Springer-Verlag New York, Inc. (1982)
[230] Vapnik, V.: Statistical Learning Theory. Wiley New York, Inc. (1998)
[231] Vapnik, V.N.: The nature of statistical learning theory. Springer (1995)
[232] Čerňanský, M.: Training recurrent neural network using multistream extended kalman filter on multicore processor and CUDA enabled graphic processor unit. In: Alippi, C., Polycarpou, M., Panayiotou, C., Ellinas, G. (eds.) ICANN 2009, Part I. LNCS, vol. 5768, pp. 381–390. Springer, Heidelberg (2009)

[233] Verleysen, M.: Learning high-dimensional data. In: Ablameyko, S., Gori, M., Goras, L., Piuri, V. (eds.) Limitations and Future Trends in Neural Computation. NATO Science Series: Computer and Systems Sciences, vol. 186, pp. 141–162. IOS Press (2003)
[234] Verleysen, M., Rossi, F., François, D.: Advances in feature selection with mutual information. In: Biehl, M., Hammer, B., Verleysen, M., Villmann, T. (eds.) Similarity-Based Clustering. LNCS (LNAI), vol. 5400, pp. 52–69. Springer, Heidelberg (2009)
[235] Vieira, A.S., Duarte, J., Ribeiro, B., Neves, J.C.: Accurate prediction of financial distress of companies with machine learning algorithms. In: Kolehmainen, M., Toivanen, P., Beliczynski, B. (eds.) ICANNGA 2009. LNCS, vol. 5495, pp. 569–576. Springer, Heidelberg (2009)
[236] Vonk, E., Jain, L.C., Veelenturf, L.P.J.: Neural network applications. In: Electronic Technology Directions, pp. 63–67 (1995)
[237] Žliobaitė, I.: Combining time and space similarity for small size learning under concept drift. In: Rauch, J., Raś, Z.W., Berka, P., Elomaa, T. (eds.) ISMIS 2009. LNCS (LNAI), vol. 5722, pp. 412–421. Springer, Heidelberg (2009)
[238] Wang, J., Zhang, B., Wang, S., Qi, M., Kong, J.: An adaptively weighted sub-pattern locality preserving projection for face recognition. Journal of Network and Computer Applications 33(3), 323–332 (2010)
[239] Wang, S.: Classification with incomplete survey data: a hopfield neural network approach. Computers & Operations Research 32(10), 2583–2594 (2005)
[240] Wang, W.: Some fundamental issues in ensemble methods. In: International Joint Conference on Neural Networks (IJCNN 2008), pp. 2243–2250 (2008)
[241] Widrow, B., Rumelhart, D.E., Lehr, M.A.: Neural networks: applications in industry, business and science. Communications of the ACM 37(3), 93–105 (1994)
[242] Wilson, D.R., Martinez, T.R.: Reduction techniques for instance-based learning algorithms. Machine Learning 38(3), 257–286 (2000)
[243] Alpha, W.: WolframAlpha – computational knowledge engine (2013), http://www.wolframalpha.com
[244] Wong, M.-L., Wong, T.-T., Fok, K.-L.: Parallel evolutionary algorithms on graphics processing unit. In: Proceedings of the 2005 IEEE Congress on Evolutionary Computation, vol. 3, pp. 2286–2293 (2005)
[245] Wurst, M.: The word vector tool user guide operator reference developer tutorial (2007)
[246] Xiang, X., Zhang, M., Li, G., He, Y., Pan, Z.: Real-time stereo matching based on fast belief propagation. Machine Vision and Applications 23(6), 1219–1227 (2012)
[247] Xu, B., Lu, J., Huang, G.: A constrained non-negative matrix factorization in information retrieval. In: IEEE International Conference on Information Reuse and Integration (IRI 2003), pp. 273–277 (2003)
[248] Xu, Y., Chen, H., Klette, R., Liu, J., Vaudrey, T.: Belief propagation implementation using CUDA on an NVIDIA GTX 280. In: Nicholson, A., Li, X. (eds.) AI 2009. LNCS (LNAI), vol. 5866, pp. 180–189. Springer, Heidelberg (2009)
[249] Yang, H., Fong, S.: Countering the concept-drift problem in big data using iOVFDT. In: 2013 IEEE International Congress on Big Data, pp. 126–132 (2013)
[250] Yang, Q., Wang, L., Yang, R., Wang, S., Liao, M., Nistér, D.: Real-time global stereo matching using hierarchical belief propagation. In: Proceedings of the 2006 British Machine Vision Conference (BMVC 2006), vol. 3, pp. 989–998 (2006), http://www.macs.hw.ac.uk/bmvc2006/proceedings.html
[251] Yu, D., Deng, L.: Deep learning and its applications to signal and information processing. IEEE Signal Processing Magazine 28(1), 145–154 (2011) ISSN 1053-5888
[252] Yu, Q., Chen, C., Pan, Z.: Parallel genetic algorithms on programmable graphics hardware. In: Wang, L., Chen, K., S. Ong, Y. (eds.) ICNC 2005. LNCS, vol. 3612, pp. 1051–1059. Springer, Heidelberg (2005)

[253] Yuan, X., Che, L., Hu, Y., Zhang, X.: Intelligent graph layout using many users' input. IEEE Transactions on Visualization and Computer Graphics 18(12), 2699–2708 (2012)
[254] Yuksel, S.E., Wilson, J.N., Gader, P.D.: Twenty years of mixture of experts. IEEE Transactions on Neural Networks and Learning Systems 23(8), 1177–1193 (2012)
[255] Yuming, M., Yuanyuan, Z.: Research on method of double-layers BP neural network in bicycle flow prediction. In: International Conference on Industrial Control and Electronics Engineering (ICICEE 2012), pp. 86–88 (2012)
[256] Zanni, L., Serafini, T., Zanghirati, G.: Parallel software for training large scale support vector machines on multiprocessor systems. Journal of Machine Learning Research 7, 1467–1492 (2006)
[257] Zhang, R., Wang, W.: Facilitating the applications of support vector machine by using a new kernel. Expert Systems with Applications 38, 14225–14230 (2011)
[258] Zhang, T.: Solving large scale linear prediction problems using stochastic gradient descent algorithms. In: Proceedings of the 21st International Conference on Machine Learning (ICML 2004), pp. 919–926 (2004)
[259] Zhang, Y., Shalabi, Y.H., Jain, R., Nagar, K.K., Bakos, J.D.: FPGA vs. GPU for sparse matrix vector multiply. In: International Conference on Field-Programmable Technology (FPT 2009), pp. 255–262 (2009)
[260] Zhao, W., Chellappa, R., Phillips, P.J., Rosenfeld, A.: Face recognition: A literature survey. ACM Computing Surveys (CSUR) 35(4), 399–458 (2003)
[261] Zhi, R., Flierl, M., Ruan, Q., Kleijn, W.B.: Graph-preserving sparse nonnegative matrix factorization with application to facial expression recognition. IEEE Transactions on Systems, Man, and Cybernetics – Part B: Cybernetics 41(1), 38–52 (2011)
[262] Zhongwen, L., Hongzhi, L., Zhengping, Y., Xincai, W.: Self-organizing maps computing on graphic process unit. In: Proceedings of the 13th European Symposium on Artificial Neural Networks, pp. 557–562 (2005)
[263] Zhou, Z.-H.: Three perspectives of data mining. Artificial Intelligence 143(1), 139–146 (2003)
[264] Zilu, Y., Guoyi, Z.: Facial expression recognition based on NMF and SVM. In: Proceedings of the 2009 International Forum on Information Technology and Applications (IFITA 2009), vol. 3, pp. 612–615. IEEE Computer Society (2009)

Index