

# Machine Learning for Adaptive Multi-Core Machines



NOEL DE JESUS MENDONÇA LOPES

*PhD thesis submitted to the* Department of Informatics Engineering  
of the Faculty of Sciences and Technology, University of Coimbra, Portugal

Coimbra, July 2013

Supervised by Prof. Dr. Bernardete Martins Ribeiro



Dedicated to my daughter and son

Sara and Pedro





---

## Abstract

---

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 industry. 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 Machine Learning (ML) algorithms have to solve. This growth is driving the need to extend the applicability of existing ML algorithms to larger datasets and to devise parallel algorithms that scale well with the volume of data or, in other words, can handle “Big Data”. In this Thesis, we partly contribute to solving this problem, by making use of two complementary components: a body of novel ML algorithms and a set of high-performance ML parallel implementations for adaptive multi-core machines.

In the first component, 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), is presented. Also, a novel Semi-Supervised Non-Negative Matrix Factorization (SSNMF) algorithm, aiming at extracting the most discriminating characteristics of each class, while reducing substantially the overall time required for generating the models, is proposed. In addition, a novel Incremental Hypersphere Classifier (IHC) with built-in multi-class support, which is able to accommodate memory and computational restrictions while providing good classification performance, is presented. This highly-scalable algorithm can update models and classify new data in real-time as well as handle concept drift scenarios. Moreover, since it keeps the samples that are near the decision frontier while removing noisy and less relevant ones, it can select a representative subset of the data for applying more sophisticated algorithms in a fraction of the time required for the complete dataset. A learning framework (IHC-SVM), encompassing the IHC and Support Vector Machine (SVM) algorithms is validated in a real-world case study of protein membership prediction. Overall the resulting system proved to be able to excel the baseline SVM (with an F-measure of 96.39%) using only a subset of the data (ca. 50%) and demonstrated its capacity to deal with the everyday dynamic changes of real-world biological databases. In another direction, and motivated by the need to deal with missing data often

occurring in large-scale data, a novel solution, designated by Neural Selective Input Model (NSIM), is proposed. The method empowers Neural Networks (NNs) with the ability to handle Missing Values (MVs) and excels single imputation techniques while offering better or similar classification performance than the state-of-the-art multiple imputation methods. With the new methodology we have successfully addressed a real-world case study of bankruptcy prediction in a large dataset of French companies, with results (F-measure of 95.70%) that are superior to previous approaches.

The backbone of the second component of this Thesis is a Graphics Processing Unit (GPU) computational framework, named GPU Machine Learning Library (GPUMLib), which aims at providing the building blocks for developing high-performance GPU parallel ML software, promote cooperation within the field and contribute to the development of innovative applications. The rationale consists of taking advantage of the GPU high-throughput parallel architecture to expand the scalability of supervised, semi-supervised and unsupervised ML algorithms. Since its release, GPUMLib, now with over 2,000 downloads, has benefited researchers worldwide.

New GPU parallel implementations of the Back-Propagation (BP) and Multiple Back-Propagation (MBP) supervised algorithms, integrating the NSIM, are presented, providing significant speedups (up to 180 $\times$ ). In particular, these implementations played an important role for the detection of Ventricular Arrhythmias (VAs) (with a sensitivity of 98.07%) that improved previous work, by reducing the computation time from weeks to hours. In this line, an Autonomous Training System (ATS) is designed to automatically find GPU high-quality solutions. In the unsupervised verge, a GPU parallel implementation of the CD- $k$  algorithm, which boosts considerably the RBMs and DBNs training speed, is presented, achieving speedups up to 46 $\times$ . Additionally, new GPU parallel implementations of the Non-Negative Matrix Factorization (NMF) algorithm are presented, yielding speedups up to 706 $\times$ . Both unsupervised implementations are tested in benchmarks and in real datasets.

Overall, this Thesis contributes with adaptive multi-core machines for exploring “Big Data”, which – as we hope – will have a positive impact in solving otherwise intractable ML problems.

---

## Resumo

---

Hoje-em-dia, o aumento da complexidade e dos requisitos de desempenho é transversal às mais variadas atividades humanas, desde a ciência à indústria. Vários fatores, tais como os avanços nas tecnologias de armazenamento e aquisição de dados ou o crescimento da Internet contribuíram para a existência de um volume de dados sem precedente, o que rapidamente aumentou a complexidade dos sistemas de Aprendizagem Computacional (AC). Tal leva a uma necessidade crescente da aplicabilidade dos algoritmos existentes a conjuntos de dados de dimensão muito elevada e ao desenvolvimento de novos algoritmos que tenham em conta a escalabilidade, isto é, que consigam lidar com *Big Data*. Esta tese contribui em parte para solucionar este problema, através de dois componentes complementares: um conjunto de algoritmos inovadores e uma série de implementações paralelas de algoritmos de AC para máquinas *multi-core* adaptativas.

Relativamente ao primeiro componente é apresentada uma nova técnica (*adaptive step size*) que melhora a convergência das *Restricted Boltzmann Machines* (RBMs), reduzindo de forma efetiva o tempo de treino das *Deep Belief Networks* (DBNs). Apresentamos ainda um novo algoritmo, *Semi-Supervised Non-Negative Matrix Factorization* (SSNMF), capaz de reduzir efetivamente o tempo necessário para gerar modelos, que visa extrair características únicas para cada classe. Para além disso, apresentamos um algoritmo inovador, *Incremental Hypersphere Classifier* (IHC), capaz de lidar com *concept drifts* e de acomodar restrições de memória e de processamento, ao mesmo tempo que mantém um bom desempenho em termos de classificação. Desta forma é possível atualizar modelos e classificar novos dados em tempo real. Por outro lado, uma vez que o IHC retém as instâncias perto da fronteira de decisão, removendo as menos relevantes, pode ser utilizado para escolher um conjunto de dados representativo do original, permitindo utilizar algoritmos mais sofisticados numa fração do tempo original. É também proposta uma *framework* (IHC-SVM), que integra o IHC e as máquinas de vetores de suporte (SVM), para aplicação num problema real de classificação de proteínas. Esta *framework* permitiu obter melhores resultados (*F-measure* de 96.39%) do que o valor referência das SVMs, utilizando apenas cerca de metade dos dados. Noutra direção, foi abordado o problema da existência de *Missing Values* (MVs), que é bastante frequente quando

lidamos com grandes volumes de dados e neste contexto requer novas abordagens. Assim, é proposto um algoritmo inovador, *Neural Selective Input Model* (NSIM), que permite às redes neuronais lidarem diretamente com MVs. O desempenho deste algoritmo é melhor do que o dos métodos de *single imputation*, apresentando níveis de desempenho superiores ou similares aos métodos de *multiple imputation*. Utilizando o NSIM foi possível obter resultados melhores (*F-measure* de 95.70%) do que os conseguidos anteriormente num problema real de falência de empresas, num universo de mais de 60000.

O segundo componente desta tese centra-se numa *framework* computacional de programação paralela para *Graphics Processing Unit* (GPU), designada por GPUMLib. A GPUMLib visa facilitar a construção de sistemas de AC com elevado desempenho, promover a cooperação na área e contribuir para o desenvolvimento de aplicações inovadoras. A ideia é tirar partido da arquitetura massivamente paralela do GPU para expandir a escalabilidade dos algoritmos (de AC) supervisionados, semi-supervisionados e não supervisionados. Desde o seu lançamento, a *framework* que conta já com mais de 2000 *downloads*, tem sido amplamente usada por investigadores em todo o mundo.

Foram criadas implementações paralelas dos algoritmos *Back-Propagation* (BP) e *Multiple Back-Propagation* (MBP) para GPU, que integram o NSIM. Estas apresentaram *speedups* consideráveis (até 180×), permitindo no caso concreto de um problema real de deteção de arritmias ventriculares transformar semanas de trabalho em horas, o que se traduziu em resultados melhores (*sensitivity* de 98.07%) que os previamente obtidos. Ainda nesta linha foi criado um sistema de treino autónomo (ATS), com base nas referidas implementações, que é capaz de encontrar soluções de elevada qualidade de forma autónoma.

No âmbito dos algoritmos não supervisionados foi criada uma implementação paralela para GPU do algoritmo *CD-k*, que reduz consideravelmente o tempo de treino das RBMs e DBNs, providenciando *speedups* até 46×. Foram ainda criadas implementações paralelas para GPU do algoritmo *Non-Negative Matrix Factorization* (NMF), com *speedups* até 706×. Estas abordagens foram testadas em *benchmarks* e problemas reais.

Em geral e tirando partido das máquinas *multi-core* adaptativas, esta tese contribui para explorar volumes de dados de grande dimensão, pelo que esperamos que a mesma venha a ter um impacto tangível na resolução de problemas de AC que de outra forma seriam intratáveis.

---

## Acknowledgments

---

First and foremost, I wish to express my deepest gratitude to my supervisor, Professor Bernardete Martins Ribeiro who taught me the beauty of Machine Learning and Pattern Recognition. This work would not have been possible without her amazing efforts, availability, guidance, support and spirit of motivation.

I would like also to thank Professor Manuela Simões, Filipe Duarte and Paulo Vieira and to all the other friends and colleagues for their comments that overall helped to improve this Thesis manuscript.

I am also very grateful to Professor Samuel Walter Best for his friendship and for proof correcting this Thesis.

I also acknowledge Professor John Owens, from the University of California, Davis, USA, for the courtesy of Figure 2.11.

Finally, I am deeply indebted to my wife Emily, and family for their love, support and inspiration.

I wish to gratefully acknowledge the Foundation for Science and Technology (FCT) for funding the first two years of this work, under the scholarship SFRH/BD/62479/2009. Additionally I would like to thank the Centre for Informatics and Systems (CISUC) of the University of Coimbra and the Research Unit for Inland Development (UDI) of the Polytechnic of Guarda (IPG) for funding.

“O caminho faz-se caminhando.”

*António Machado*

Coimbra, 2013



---

# Table of Contents

---

|   |              |
|---|--------------|
| <b>Abstract</b>   | <b>i</b>     |
| <b>Resumo</b>   | <b>iii</b>   |
| <b>Acknowledgments</b>  | <b>vi</b>    |
| <b>Table of Contents</b>  | <b>ix</b>    |
| <b>List of Figures</b>  | <b>xv</b>    |
| <b>List of Tables</b>   | <b>xviii</b> |
| <b>List of Algorithms</b>   | <b>xix</b>   |
| <b>Code Listings</b>  | <b>xxi</b>   |
| <b>List of Acronyms</b>   | <b>xxiii</b> |
| <b>List of Symbols</b>  | <b>xxvii</b> |
| <b>1 Introduction</b>   | <b>1</b>     |
| 1.1 Motivation . . . . .  | 2            |
| 1.2 Challenges and Research Questions . . . . .                     | 2            |
| 1.3 Problem Statement . . . . .                                     | 5            |
| 1.4 Thesis Contributions . . . . .                                  | 7            |
| 1.5 Outline of the Thesis . . . . .                                 | 11           |
| <b>2 GPU Machine Learning Library (GPUMLib)</b>                     | <b>13</b>    |
| 2.1 Introduction . . . . .  | 14           |
| 2.2 A Review of GPU Parallel Implementations of ML Algorithms . . . | 17           |

|          |   |            |
|----------|---|------------|
| 2.3      | GPU Computing . . . . .   | 19         |
| 2.4      | Compute Unified Device Architecture (CUDA) . . . . .                    | 20         |
| 2.4.1    | CUDA Programming Model . . . . .  | 20         |
| 2.4.2    | CUDA architecture . . . . .   | 24         |
| 2.5      | GPUMLib architecture . . . . .  | 28         |
| 2.6      | Summary . . . . .   | 35         |
| <b>3</b> | <b>Experimental Setup and Performance Evaluation</b>                    | <b>37</b>  |
| 3.1      | Hardware and Software Configurations . . . . .                          | 37         |
| 3.2      | Evaluation Metrics . . . . .  | 38         |
| 3.3      | Validation . . . . .  | 41         |
| 3.4      | Benchmarks . . . . .  | 43         |
| 3.5      | Case Studies . . . . .  | 51         |
| 3.6      | Data Preprocessing . . . . .  | 55         |
| 3.7      | Summary . . . . .   | 57         |
| <b>4</b> | <b>Supervised algorithms</b>  | <b>61</b>  |
| 4.1      | Multiple Back-Propagation (MBP) . . . . .                               | 62         |
| 4.1.1    | Back-Propagation (BP) Algorithm . . . . .                               | 63         |
| 4.1.2    | Multiple Back-Propagation (MBP) Algorithm . . . . .                     | 69         |
| 4.1.3    | GPU Parallel Implementation . . . . .                                   | 75         |
| 4.1.4    | Autonomous Training System (ATS) . . . . .                              | 79         |
| 4.1.5    | Results and Discussion . . . . .  | 79         |
| 4.2      | Neural Selective Input Model (NSIM) . . . . .                           | 91         |
| 4.2.1    | Missing Data Mechanisms . . . . .                                       | 93         |
| 4.2.2    | Methods for Handling Missing Values (MVs) in Machine Learning . . . . . | 94         |
| 4.2.3    | Neural Selective Input Model (NSIM) Proposed Approach . . . . .         | 97         |
| 4.2.4    | GPU Parallel Implementation . . . . .                                   | 99         |
| 4.2.5    | Results and Discussion . . . . .  | 99         |
| 4.3      | Incremental Hypersphere Classifier (IHC) . . . . .                      | 104        |
| 4.3.1    | Proposed Incremental Hypersphere Classifier Algorithm . . . . .         | 106        |
| 4.3.2    | Results and Discussion . . . . .  | 110        |
| 4.4      | Summary . . . . .   | 121        |
| <b>5</b> | <b>Unsupervised and Semi-supervised algorithms</b>                      | <b>125</b> |
| 5.1      | Non-Negative Matrix Factorization (NMF) . . . . .                       | 127        |
| 5.1.1    | NMF Algorithm . . . . .   | 128        |
| 5.1.2    | Combining NMF with other ML Algorithms . . . . .                        | 131        |
| 5.1.3    | Semi-Supervised NMF (SSNMF) . . . . .                                   | 131        |
| 5.1.4    | GPU Parallel Implementation . . . . .                                   | 134        |
| 5.1.5    | Results and Discussion . . . . .  | 139        |
| 5.2      | Deep Belief Networks (DBNs) . . . . .                                   | 153        |
| 5.2.1    | Restricted Boltzmann Machines (RBMs) . . . . .                          | 156        |



|          |   |              |
|----------|---|--------------|
| 5.2.2    | Deep Belief Networks Architecture . . . . .             | 162          |
| 5.2.3    | Adaptive Step Size Technique . . . . .                  | 163          |
| 5.2.4    | GPU parallel implementation . . . . .                   | 163          |
| 5.2.5    | Results and Discussion . . . . .                        | 170          |
| 5.3      | Summary . . . . .                                       | 182          |
| <b>6</b> | <b>Conclusions and Perspectives</b>                     | <b>189</b>   |
| 6.1      | Main Research Accomplishments and Conclusions . . . . . | 189          |
| 6.2      | Future Work . . . . .                                   | 194          |
|          | <b>Bibliography</b>                                     | <b>XXII</b>  |
|          | <b>Index</b>  | <b>XXVII</b> |



---

## List of Figures

---

|      |  |    |
|------|--|----|
| 1.1  | Using Machine Learning (ML) algorithms to extract information from data. . . . .   | 3  |
| 1.2  | Machine Learning paradigms. . . . .  | 6  |
| 1.3  | Combining supervised and unsupervised models. . . . .  | 7  |
| 2.1  | Disparity between the Central Processing Unit (CPU) and the Graphics Processing Unit (GPU) peak floating point performance, over the years, in billions ( $10^9$ ) of floating-point operations per second (GFLOPS). . . . . | 15 |
| 2.2  | Chronology of Machine Learning (ML) software Graphics Processing Unit (GPU) implementations. . . . .   | 18 |
| 2.3  | Graphics hardware pipeline. . . . .  | 19 |
| 2.4  | Example of a kernel grid. . . . .  | 22 |
| 2.5  | Execution of the square kernel grid blocks (see Listings 2.1 and 2.2). . . . .   | 23 |
| 2.6  | NVIDIA (GPU) device architecture. . . . .  | 25 |
| 2.7  | Diagram of a Fermi Streaming Multiprocessor (SM). . . . .  | 26 |
| 2.8  | Execution of a kernel grid on different devices (Graphics Processing Units (GPUs)). . . . .  | 27 |
| 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. . . . .  | 28 |
| 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. . . . .  | 29 |
| 2.11 | Main components of the GPULib. . . . .   | 30 |
| 2.12 | Row-major versus column-major orders. . . . .  | 32 |
| 2.13 | Example of a sum reduction. . . . .  | 32 |
| 2.14 | Evolution of the number of downloads of GPULib. . . . .  | 34 |

|      |  |    |
|------|--|----|
| 2.15 | Number of GPUMLib downloads according to the operating system.   | 34 |
| 2.16 | Number of GPUMLib downloads per country. Regions with a higher number of downloads are represented with darker colors. . . . .   | 35 |
| 3.1  | Experiments associated with a 4-fold cross-validation procedure. . .   | 42 |
| 3.2  | Randomly selected examples from the AT&T (ORL) face images. .  | 45 |
| 3.3  | Randomly selected examples of the face images contained in the CBCL training dataset. . . . .  | 47 |
| 3.4  | Examples of the HHreco multi-stroke images. Each column contains a symbol while each row contains the images drawn by one of the users. . . . .                                | 48 |
| 3.5  | 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. . . . . | 50 |
| 3.6  | <i>Sinus Cardinalis</i> function. . . . .  | 51 |
| 3.7  | Two spirals dataset. . . . .   | 51 |
| 3.8  | Yale face images. Each row contains the images of a specific individual and each column a different expression/ configuration. . .   | 52 |
| 3.9  | Typical Electrocardiograph (ECG) diagram of a normal sinus rhythm for a human heart. . . . .   | 55 |
| 3.10 | AT&T face images after applying a histogram equalization to the original images presented in Figure 3.2. . . . .   | 58 |
| 3.11 | CBCL face images after applying a histogram equalization to the original images presented in Figure 3.3. . . . .   | 59 |
| 3.12 | Yale face images after applying a histogram equalization to the original images presented in Figure 3.8. . . . .   | 60 |
| 4.1  | Three-layer feed-forward network. . . . .  | 64 |
| 4.2  | Connection between two neurons. . . . .  | 64 |
| 4.3  | Neuron architecture. . . . .   | 65 |
| 4.4  | Sigmoid function. . . . .  | 66 |
| 4.5  | A neural network viewed as a black box system that maps $D$ inputs into $C$ outputs. . . . .   | 69 |
| 4.6  | Selective actuation neuron architecture. . . . .   | 71 |
| 4.7  | Architecture of a selective actuation neuron, with linear activation functions, which solves the XOR problem. . . . .  | 72 |
| 4.8  | Example of a multiple feed-forward network. . . . .  | 73 |
| 4.9  | Model of the kernels executed (in each epoch) to complete the forward phase of an MBP network. . . . .   | 75 |
| 4.10 | Model of the kernels executed (in each epoch) in the back-propagation phase of an MBP network. . . . .   | 78 |
| 4.11 | <i>Two-spirals</i> training time (MBP algorithm). . . . .  | 83 |

---

|      |   |     |
|------|---|-----|
| 4.12 | Number of epochs per minute using the BP algorithm for the <i>forest cover</i> problem. The GPU speedups are shown near the corresponding lines. . . . .          | 84  |
| 4.13 | Number of epochs per minute using the MBP algorithm for the <i>forest cover</i> problem. The GPU speedups are shown near the corresponding lines. . . . .         | 85  |
| 4.14 | Number of epochs per minute using the BP algorithm for the <i>poker</i> problem. The GPU speedups are shown near the corresponding lines.                         | 86  |
| 4.15 | Number of epochs per minute using the MBP algorithm for the <i>poker</i> problem. The GPU speedups are shown near the corresponding lines.                        | 86  |
| 4.16 | Number of epochs per minute for the Ventricular Arrhythmias case study, using the BP algorithm. . . . .   | 87  |
| 4.17 | Number of epochs per minute for the Ventricular Arrhythmias case study, using the MBP algorithm. . . . .  | 87  |
| 4.18 | Speedup ( $\times$ ) obtained for the Ventricular Arrhythmias case study, using an 8600 GT. . . . .   | 88  |
| 4.19 | Speedup ( $\times$ ) obtained for the Ventricular Arrhythmias case study, using a GTX 280. . . . .  | 88  |
| 4.20 | Networks trained, according to the number of hidden neurons. . . . .  | 90  |
| 4.21 | Speedups ( $\times$ ) versus processing threads. . . . .  | 92  |
| 4.22 | Overview of the types of techniques for handling Missing Values (MVs) in Machine Learning (ML). . . . .   | 95  |
| 4.23 | Physical and conceptual models of a network with a selective input ( $j = 3$ ). . . . .   | 98  |
| 4.24 | GPU speedups obtained for the bankruptcy problem. . . . .   | 104 |
| 4.25 | Application of the IHC algorithm to a toy problem. . . . .  | 107 |
| 4.26 | Regions of influence and decision surfaces generated by IHC for a toy problem ( $g = 1$ ). . . . .  | 108 |
| 4.27 | Average time required to update the IHC model (after presenting a new sample) for the <i>KDD Cup 1999</i> dataset. . . . .  | 114 |
| 4.28 | Accuracy of the IHC model for the <i>KDD Cup 1999</i> dataset. . . . .  | 114 |
| 4.29 | Evolution of the F-Measure for the Internet usage dataset. . . . .  | 116 |
| 4.30 | Evolution of the F-Measure for the electricity dataset. . . . .   | 116 |
| 4.31 | IHC-SVM learning framework. . . . .   | 118 |
| 4.32 | Average time required to update the IHC model (with a new sample) for the protein membership prediction case study. . . . .                                       | 119 |
| 4.33 | IHC and IHC-SVM macro-average F-measure performance for the protein membership prediction case study. . . . .   | 119 |
| 5.1  | NMF factorization. . . . .  | 129 |
| 5.2  | Combining NMF with other learning algorithms. . . . .   | 132 |
| 5.3  | Generation and combination of the individual class matrices. The white areas of the $\mathbf{H}_{\text{train}}$ matrix correspond to zero value elements. . . . . | 133 |

|      |   |     |
|------|---|-----|
| 5.4  | Typical basis vectors ( $\mathbf{W}$ columns) generated by NMF and SSNMF for the Yale face database (using $r_i = 3$ and $r = 45$ ( $3 \times 15$ )). . . . .   | 133 |
| 5.5  | Interpretation of the same data, using either row-major or column-major orders. . . . .   | 134 |
| 5.6  | Processing carried out, for each element $\mathbf{H}_{a\mu}$ , by the <code>UpdateH_MD</code> kernel. . . . .   | 139 |
| 5.7  | Time required to run the NMF algorithms on the CBCL face database, during 1,000 iterations, using the multiplicative update rules. The speedups ( $\times$ ) provided by the GPU are shown in the respective lines. . . . .   | 141 |
| 5.8  | Time required to run the NMF algorithms on the CBCL face database, during 1,000 iterations, using the additive update rules. The speedups ( $\times$ ) provided by the GPU are shown in the respective lines. . . . .   | 142 |
| 5.9  | NMF GPU Speedups for the Center for Biological and Computational Learning (CBCL) face database. . . . .   | 143 |
| 5.10 | Approximations and parts representation generated by the NMF algorithms. . . . .  | 143 |
| 5.11 | Parts-based faces representations, $\mathbf{W}$ , generated by NMF for the Yale dataset. . . . .  | 144 |
| 5.12 | Number of networks trained by the ATS, according to the number of neurons. . . . .  | 145 |
| 5.13 | Time to perform 10,000 NMF iterations on the Yale database. The speedups ( $\times$ ) provided by the GPU are shown in the respective lines. . . . .  | 146 |
| 5.14 | Time to perform 10,000 NMF iterations on the AT&T (ORL) database. The speedups ( $\times$ ) provided by the GPU are shown in the respective lines. . . . .  | 147 |
| 5.15 | Time required to compute the $\mathbf{W}$ and $\mathbf{H}_{\text{train}}$ matrices. . . . .   | 149 |
| 5.16 | Average accuracy yielded by the SVM algorithm for the Yale dataset. . . . .   | 151 |
| 5.17 | Average accuracy yielded by the SVM algorithm for the AT&T dataset. . . . .   | 152 |
| 5.18 | Deep architectures versus shallow ones. . . . .   | 154 |
| 5.19 | Schematic representation of a Restricted Boltzmann Machine (RBM). . . . .   | 156 |
| 5.20 | 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). . . . .  | 157 |
| 5.21 | 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. . . . .   | 160 |
| 5.22 | 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, purple color represents layers already trained, while cyan represents the Restricted Boltzmann Machine (RBM) being trained. . . . . | 162 |

---

|      |  |     |
|------|--|-----|
| 5.23 | Sequence of GPU kernel calls, per epoch, that implement the CD- $k$ algorithm. . . . .   | 164 |
| 5.24 | <code>ComputeStatusHiddenUnits</code> kernel grid and block structure. . . . .   | 165 |
| 5.25 | Implications of storing the connection weights using row-major order.  | 166 |
| 5.26 | Grid and block structure used by the first approach of the kernel <code>CorrectWeights</code> . . . . .  | 167 |
| 5.27 | Proportion of time spent, per epoch, in each kernel, as measured in the computer System 2 (with a GTX 280). . . . .  | 168 |
| 5.28 | Connections to the hidden unit $j$ . . . . .   | 168 |
| 5.29 | Connections to the visible unit $i$ . . . . .  | 169 |
| 5.30 | Block structure of the improved approach of the <code>CorrectWeights</code> kernel. . . . .  | 169 |
| 5.31 | MNIST average training time per epoch (GPU speedups are indicated).  | 172 |
| 5.32 | Average reconstruction error (RMSE) according to the learning parameters, $\eta$ and $\alpha$ . . . . .  | 173 |
| 5.33 | Impact of the step size technique on the convergence of a RBM ( $\alpha = 0.1$ ). . . . .  | 174 |
| 5.34 | Receptive fields of the best networks trained either with the adaptive step size or with a fixed learning rate. . . . .  | 175 |
| 5.35 | 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. . . . . | 176 |
| 5.36 | DBNs classification performance, according to the number of pre-training layers. . . . .   | 178 |
| 5.37 | DBNs classification performance, according to the number of neurons in the first hidden layer. . . . .   | 181 |
| 5.38 | DBNs classification performance, according to the topology of the additional layers (added to the pre-trained networks). . . . .   | 183 |
| 5.39 | DBNs classification performance, depending on whether or not an additional hidden layer was added to the pre-trained networks. . . . .   | 184 |





---

## List of Tables

---

|     |   |     |
|-----|---|-----|
| 2.1 | Principal technical specifications according to the CUDA device compute capability. . . . .   | 21  |
| 2.2 | Built-in CUDA kernel variables. . . . .   | 22  |
| 2.3 | Number of Scalar Processor (SP) cores per Streaming Multiprocessor (SM), according to the compute capability of the device (GPU). . .   | 25  |
| 2.4 | GPUMLib memory access framework classes. . . . .  | 30  |
| 2.5 | GPU parallel algorithms implemented in version 0.2.0 of GPUMLib. . . . .  | 33  |
| 3.1 | Hardware and Software system main characteristics. . . . .  | 38  |
| 3.2 | Main characteristics of the NVIDIA GeForce devices used in this work. . . . .   | 38  |
| 3.3 | Confusion matrix for a binary classification problem. . . . .   | 40  |
| 3.4 | Main characteristics of the benchmark datasets. . . . .   | 44  |
| 3.5 | Main characteristics of the real-world case studies. . . . .  | 51  |
| 3.6 | Financial ratios selected to create a bankruptcy model. . . . .   | 54  |
| 3.7 | Selected features from the ECG signal. . . . .  | 56  |
| 4.1 | Main characteristics of the training datasets used in the MBP experimental setup. . . . .   | 81  |
| 4.2 | Speedups ( $\times$ ) for the <i>sinus cardinalis</i> problem. . . . .  | 82  |
| 4.3 | Speedups ( $\times$ ) for the <i>two-spirals</i> problem. . . . .   | 82  |
| 4.4 | Performance results (%) for the ventricular arrhythmias problem. . . . .  | 89  |
| 4.5 | Main characteristics, proportion and distribution of the Missing Values (MVs) for the UCI database benchmark experiments (after data preprocessing). Note that the average (avg.) and the standard deviation (stdev.) of Missing Values (MVs) per feature does not include features without Missing Values (MVs). . . . . | 100 |

|      |  |     |
|------|--|-----|
| 4.6  | Macro-average F-Measure performance (%) according to the methods used to handle the Missing Values (MVs) and the algorithms used to train the Neural Networks (NNs). . . . . | 101 |
| 4.7  | Results of the NSIM for the bankruptcy problem. . . . .  | 103 |
| 4.8  | IHC and 1-nn classification performance (macro-average F-measure (%)) for the test datasets of the UCI benchmark experiments. . . . .  | 112 |
| 4.9  | Classification performance, macro-average F-measure (%), and storage reduction (%) of the IHC and IB3 algorithms for the UCI benchmark experiments. . . . .                  | 113 |
| 4.10 | IHC-SVM storage reduction and classification improvement over the baseline (SVM). . . . .  | 120 |
| 5.1  | Accuracy (%) results for the Yale dataset. . . . .   | 148 |
| 5.2  | Accuracy (%) results for the AT&T (ORL) dataset. . . . .   | 148 |
| 5.3  | Percentage of zero values present in the $\mathbf{H}_{\text{test}}$ matrix. . . . .  | 150 |
| 5.4  | Grid search average accuracy on the test folds. . . . .  | 150 |
| 5.5  | Accuracy (%) results for the Yale dataset. . . . .   | 150 |
| 5.6  | Accuracy results for the AT&T (ORL) dataset. . . . .   | 153 |
| 5.7  | Top 10 DBNs with the best classification performance for the MNIST dataset. The topology column refers to the topology of the added classification layers. . . . .           | 177 |
| 5.8  | Top 10 DBNs with the best classification performance for the HHreco dataset. The topology column refers to the topology of the added classification layers. . . . .          | 179 |
| 5.9  | Confusion matrix of the best MNIST DBN (trained with 60,000 samples). . . . .  | 180 |

---

## List of Algorithms

---

|   |   |     |
|---|---|-----|
| 1 | Autonomous Training System. . . . .                         | 80  |
| 2 | Incremental Hypersphere Classifier (IHC) algorithm. . . . . | 109 |
| 3 | CD- $k$ algorithm. . . . .                                  | 161 |



---

## Code Listings

---

|     |  |     |
|-----|--|-----|
| 2.1 | Example of a CUDA kernel function. Compute Unified Device Architecture (CUDA) specific keywords appear in blue. . . . .                                      | 23  |
| 2.2 | Example for calling a CUDA kernel function. . . . .  | 24  |
| 2.3 | Example for calling a CUDA kernel function using the GPULib memory access framework classes. . . . .   | 31  |
| 4.1 | FireLayer kernel. . . . .  | 77  |
| 4.2 | FireSelectiveInputs kernel. . . . .  | 100 |
| 5.1 | CUDA kernel used to implement the NMF algorithm for the multiplicative update rules, considering the Euclidean distance. . .                                 | 135 |
| 5.2 | NMF iteration code for the multiplicative update rules, considering the Euclidean distance. . . . .  | 136 |
| 5.3 | CUDA kernel used to implement the NMF algorithm for the additive update rules, considering the Euclidean distance. . . . .                                   | 137 |
| 5.4 | One of the CUDA kernels (SumW) used to implement the NMF algorithm for the multiplicative update rules, considering the Kullback-Leibler divergence. . . . . | 138 |



---

## List of Acronyms

---

|                |   |     |
|----------------|---|-----|
| <b>API</b>     | Application Programming Interface .....                     | 20  |
| <b>ATS</b>     | Autonomous Training System .....                            | 9   |
| <b>BP</b>      | Back-Propagation .....                                      | 5   |
| <b>CBCL</b>    | Center for Biological and Computational Learning.....       | xiv |
| <b>CD</b>      | Contrastive Divergence .....                                | 160 |
| <b>CMU</b>     | Carnegie Mellon University .....                            | 49  |
| <b>CPU</b>     | Central Processing Unit.....                                | 10  |
| <b>CUDA</b>    | Compute Unified Device Architecture .....                   | 11  |
| <b>DBN</b>     | Deep Belief Network .....                                   | 8   |
| <b>DOS</b>     | Denial Of Service .....                                     | 46  |
| <b>ECG</b>     | Electrocardiograph.....                                     | 54  |
| <b>EM</b>      | Expectation-Maximization .....                              | 70  |
| <b>ERM</b>     | Empirical Risk Minimization .....                           | 5   |
| <b>FF</b>      | Feed-Forward .....  | 63  |
| <b>FPGA</b>    | Field-Programmable Gate Array .....                         | 15  |
| <b>FPU</b>     | Floating-Point Unit .....                                   | 24  |
| <b>FRCM</b>    | Face Recognition Committee Machine.....                     | 147 |
| <b>GPGPU</b>   | General-Purpose computing on Graphics Processing Units..... | 19  |
| <b>GPU</b>     | Graphics Processing Unit.....                               | 1   |
| <b>GPUMLib</b> | GPU Machine Learning Library .....                          | 9   |
| <b>HPC</b>     | High-Performance Computing .....                            | 25  |
| <b>IB3</b>     | Instance Based learning.....                                | 108 |
| <b>ICA</b>     | Independent Component Analysis.....                         | 129 |
| <b>IHC</b>     | Incremental Hypersphere Classifier .....                    | 8   |

|                    |   |     |
|--------------------|---|-----|
| <b>KDD</b>         | Knowledge Discovery and Data mining.....                                      | 46  |
| <b>LIBSVM</b>      | Library for Support Vector Machines.....                                      | 111 |
| <b>MAR</b>         | Missing At Random.....  | 93  |
| <b>MBP</b>         | Multiple Back-Propagation.....  | 9   |
| <b>MCAR</b>        | Missing Completely At Random.....   | 93  |
| <b>MCMC</b>        | Markov Chain Monte Carlo.....   | 159 |
| <b>ME</b>          | Mixture of Experts.....   | 70  |
| <b>MFF</b>         | Multiple Feed-Forward.....  | 73  |
| <b>MIT</b>         | Massachusetts Institute of Technology.....                                    | 46  |
| <b>ML</b>          | Machine Learning.....   | 1   |
| <b>MLP</b>         | Multi-Layer Perceptron.....   | 63  |
| <b>MV</b>          | Missing Value.....  | 8   |
| <b>MVP</b>         | Missing Values Problem.....   | 9   |
| <b>NMAR</b>        | Not Missing At Random.....  | 93  |
| <b>NMF</b>         | Non-Negative Matrix Factorization.....  | 8   |
| <b><i>k</i>-nn</b> | <i>k</i> -nearest neighbor.....   | 19  |
| <b>NN</b>          | Neural Network.....   | 8   |
| <b>NORM</b>        | Multiple imputation of incomplete multivariate data under a normal model..... | 101 |
| <b>NSIM</b>        | Neural Selective Input Model.....   | 8   |
| <b>NSW</b>         | New South Wales.....  | 46  |
| <b>OpenCL</b>      | Open Computing Language.....  | 16  |
| <b>PCA</b>         | Principal Component Analysis.....   | 19  |
| <b>PVC</b>         | Premature Ventricular Contraction.....  | 54  |
| <b>R2L</b>         | unauthorized access from a remote machine.....                                | 46  |
| <b>RBF</b>         | Radial Basis Function.....  | 33  |
| <b>RBM</b>         | Restricted Boltzmann Machine.....   | 8   |
| <b>RMSE</b>        | Root Mean Square Error.....   | 39  |
| <b>SCOP</b>        | Structural Classification Of Proteins.....                                    | 53  |
| <b>SFU</b>         | Special Function Unit.....  | 25  |
| <b>SIMT</b>        | Single-Instruction Multiple-Thread.....                                       | 27  |
| <b>SM</b>          | Streaming Multiprocessor.....   | 24  |
| <b>SP</b>          | Scalar Processor.....   | 24  |



---

|               |   |    |
|---------------|---|----|
| <b>SRM</b>    | Structural Risk Minimization .....                      | 5  |
| <b>SSNMF</b>  | Semi-Supervised NMF .....                               | 8  |
| <b>SVM</b>    | Support Vector Machine .....                            | 5  |
| <b>U2R</b>    | unauthorized access to local superuser privileges ..... | 46 |
| <b>UCI</b>    | University of California, Irvine .....                  | 43 |
| <b>VA</b>     | Ventricular Arrhythmia .....                            | 10 |
| <b>VC</b>     | Vapnik-Chervonenkis .....                               | 5  |
| <b>WVTool</b> | Word Vector Tool .....                                  | 54 |



---

## List of Symbols

---

|                              |   |
|------------------------------|---|
| $a_j$                        | Activation of the neuron $j$ .                            |
| $a_i$                        | Accuracy of sample $i$ .                                  |
| $\mathbf{b}$                 | Bias of the hidden units.                                 |
| <i>accuracy</i>              | Accuracy.   |
| $Be$                         | 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.   |
| $F_{measure}$                | F-measure ( $F_1$ score).                                 |
| $fn$                         | 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.   |
| $m$                          | 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.                               |
| <i>precision</i>             | Precision.  |
| <i>precision<sub>M</sub></i> | Macro-average precision.                                  |
| $r$                          | Number of reduced features (rank).                        |

|                       |   |
|-----------------------|---|
| $r$                   | Robustness (reducing) factor.   |
| $s$                   | Number of shared parameters (between models).                               |
| $recall$              | Recall.   |
| $recall_M$            | Macro-average Recall.   |
| $RMSE$                | Root mean square error.   |
| $sensitivity$         | Sensitivity.  |
| $specificity$         | Specificity.  |
| $speedup$             | Speedup.  |
| $storage$             | Storage reduction (space savings).  |
| $\mathbf{t}$          | Targets (desired values).   |
| $\top$                | Transpose.  |
| $time_P$              | (Algorithms) Parallel time.   |
| $tn$                  | True negatives.   |
| $tp$                  | True positives.   |
| $time_S$              | (Algorithms) Sequential time.   |
| $u$                   | 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).              |
| $\alpha$              | Momentum term.  |
| $\gamma$              | Width of the Gaussian RBF kernel.   |
| $\delta$              | Local gradient.   |
| $\Delta$              | Change of a model parameter (e.g. $\Delta W_{ij}$ is the weight change).    |
| $\eta$                | Learning rate.  |
| $\theta$              | Model parameter.  |
| $\boldsymbol{\kappa}$ | Response indicator vector.  |
| $\xi$                 | Missing data mechanism parameter.   |
| $\rho_i$              | Radius of sample $i$ .  |
| $\sigma$              | Sigmoid function.   |
| $\phi$                | Neuron activation function.   |
| $\mathbb{R}$          | Set of real numbers.  |

# CHAPTER 1

---

## Introduction

---

---

|            |  |           |
|------------|--|-----------|
| <b>1.1</b> | <b>Motivation</b>                        | <b>2</b>  |
| <b>1.2</b> | <b>Challenges and Research Questions</b> | <b>2</b>  |
| <b>1.3</b> | <b>Problem Statement</b>                 | <b>5</b>  |
| <b>1.4</b> | <b>Thesis Contributions</b>              | <b>7</b>  |
| <b>1.5</b> | <b>Outline of the Thesis</b>             | <b>11</b> |

---

Today, Machine Learning (ML) algorithms play a central role in science and industry. The scale of the data from the Web growth and advances in sensor data collection technology have been rapidly increasing the magnitude and complexity of the tasks that ML algorithms have to solve. This growth is driving the need to extend the applicability of existing ML algorithms to larger datasets and to devise parallel algorithms that scale well with the volume of data, or in other words, can handle “Big Data”. In this Thesis, we present several contributions to tackle this problem. Specifically, we qualify these challenges and extend the applicability of well-known ML methods by developing Graphics Processing Unit (GPU) scalable parallel implementations of these algorithms. Then, we address new ML algorithms that scale well in the presence of large amounts of data. Additionally, we tackle the missing data problem, which often occurs in large databases. Finally, we present a computational framework GPU Machine Learning Library (GPUMLib) (Graphics Processor Units Machine Learning Library) for implementing these algorithms.

This Chapter is structured as follows. Section 1.1 presents the motivation of this Thesis. Section 1.2 addresses the challenges and research questions. Section 1.3 formalizes the supervised and unsupervised ML problems. Section 1.4 presents the

main contributions with corresponding relevant works. Finally, Section 1.5 outlines the organization of this Thesis.

### 1.1 Motivation

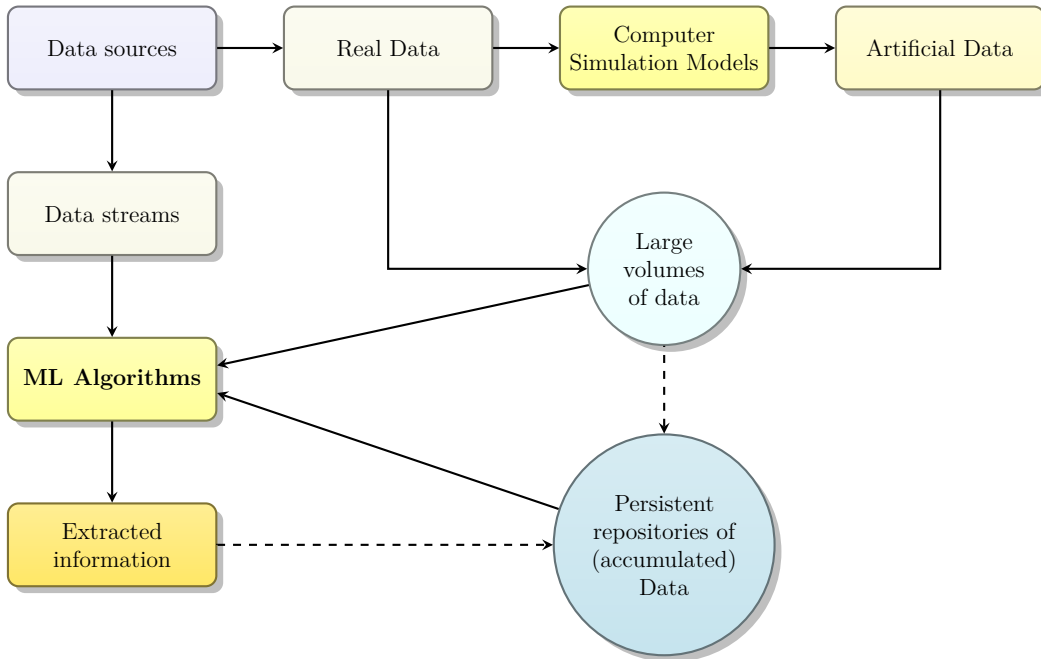
The volume of data being produced is increasing at an exponential rate, vastly exceeding our capacity to analyze it, and this trend is likely to endure. The growth of the Web and the emergence and confluence of new technologies, such as low-cost sensors, high-capacity storage devices, remote sensing, robotic systems, high-bandwidth networks and commodity computing has augmented the diversity of data sources, which resulted in an unprecedented capacity to generate, capture and share vast amounts of high-dimensional data.

According to Lyman et al., between 1999 and 2002, the amount of new information stored grew approximately 30% per year. In 2002 around 5 exabytes ( $10^{18}$  bytes) of new information were produced in the form of optical storage media, magnetic, print and film, whilst in 1999 only 2 exabytes had been produced [Lyman et al., 2003]. On average, (in 2002) each person produced 800 MB of recorded information. Moreover, 92% percent of this new information was stored in magnetic media (mainly on hard disks) [Lyman et al., 2003]. Furthermore, Lyman et al. estimate that, in 2002, nearly 18 exabytes of new information was flowing through several electronic channels (e.g. Internet, phone, television and radio), with the (worldwide) telephone calls representing 98% of these data streams [Lyman et al., 2003]. 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 [Hey et al., 2009].

Much of the accumulated data that we are generating and capturing will be made permanently available for the purposes of continued analysis [Hey et al., 2009]. 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 [Moens, 2006], which is illustrated in Figure 1.1.

### 1.2 Challenges and Research Questions

The need for gaining understanding of the information contained in large and complex datasets is common to virtually all fields of business, 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



**Figure 1.1:** Using ML algorithms to extract information from data.

knowledge can be obtained [Cherkassky and Mulier, 2007]. Moreover, science is gradually moving towards being computational and data centric [Hey et al., 2009].

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 [Hey et al., 2009]. Although the empirical, analytical and simulation paradigms still play their role in science, if we are to achieve major breakthroughs, we need to embrace this 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 [Hey et al., 2009].

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) [Alpaydin, 2010, Mjolsness and DeCoste, 2001]. According to these authors, ML encompasses the potential to amplify every aspect of the scientific work, by providing the basis for the semi-automation of scientific methods: from hypothesis generation to model construction and experimentation. 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 scientific observations [Mjolsness and DeCoste, 2001].

In this context, we expect ML algorithms to continue playing 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 the training data, is a limiting factor that can render the application of many algorithms to real-world problems, involving large datasets, impractical [Bottou and Bousquet, 2008, García-Pedrajas et al., 2010]. Thus, the challenge consists of processing large quantities of data in a realistic time scale [Hey et al., 2009]. Therefore, new scalable and high-performance implementations of ML methods are needed in order to handle efficiently the large amounts of data encompassing complex and hard to discover relationships [Hey et al., 2009]. 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 [Hey et al., 2009].

Modern GPUs are highly parallel devices that can perform general-purpose computations, providing significant speedups for many problems in a wide range of areas [Owens et al., 2008, Schaa and Kaeli, 2009]. 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 [Schaa and Kaeli, 2009]. Notwithstanding, parallel computer programs are by far more difficult to design, write, debug and fine-tune than their sequential counterparts [Hey et al., 2009]. Moreover, the GPU programming model is significantly different from the traditional models [Garland and Kirk, 2010, Owens et al., 2008]. 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 [Bottou and Bousquet, 2008]. Hence, traditional batch algorithms may not be up to the challenge and instead we must rely on incremental learning algorithms [Jain et al., 2006] 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.



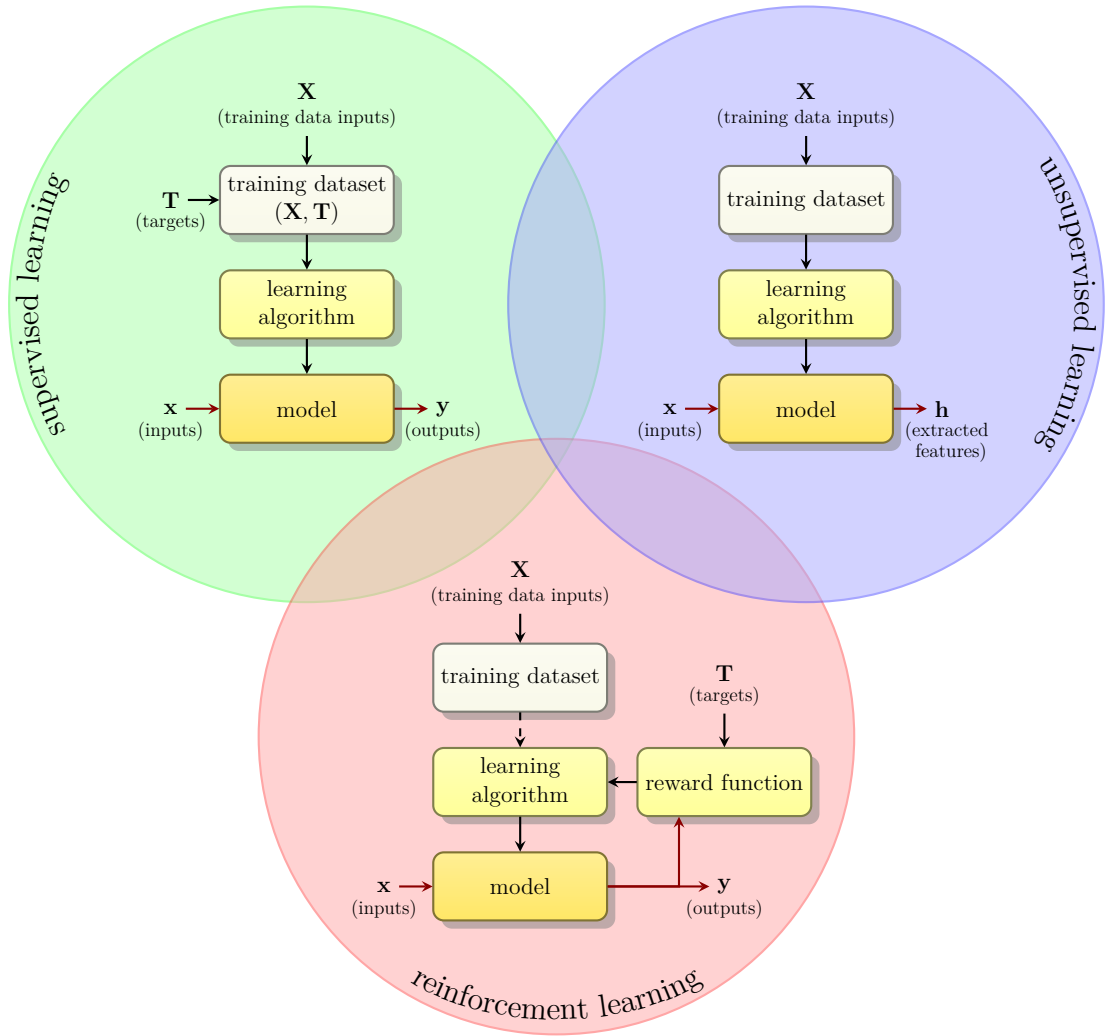
Finally, in practical scenarios, the problematic of handling large quantities of data is often exacerbated by the presence of incomplete data, which is an unavoidable problem for most real-world databases [Kotsiantis et al., 2006b, Karhunen, 2011]. Therefore, it is important to devise strategies to deal with this ubiquitous problem that does not affect significantly either the algorithms performance or the preprocessing burden.

## 1.3 Problem Statement

Learning in the context of ML corresponds to the task of adjusting the parameters,  $\theta$ , of an adaptive model, using the information contained in a so-called training dataset [Bishop, 2006]. 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) [Bishop, 2006], as depicted in Figure 1.2. 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 [Chapelle et al., 2006]. 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 include a subset of the targets or some constraint mechanism that guides the learning process [Chapelle et al., 2006].

In this Thesis 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,  $\mathbf{x} = [x_1, x_2, \dots, x_D]$ , 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  $\mathbf{t} = [t_1, t_2, \dots, t_C]$  where  $t_i \in \{0, 1\}$ .

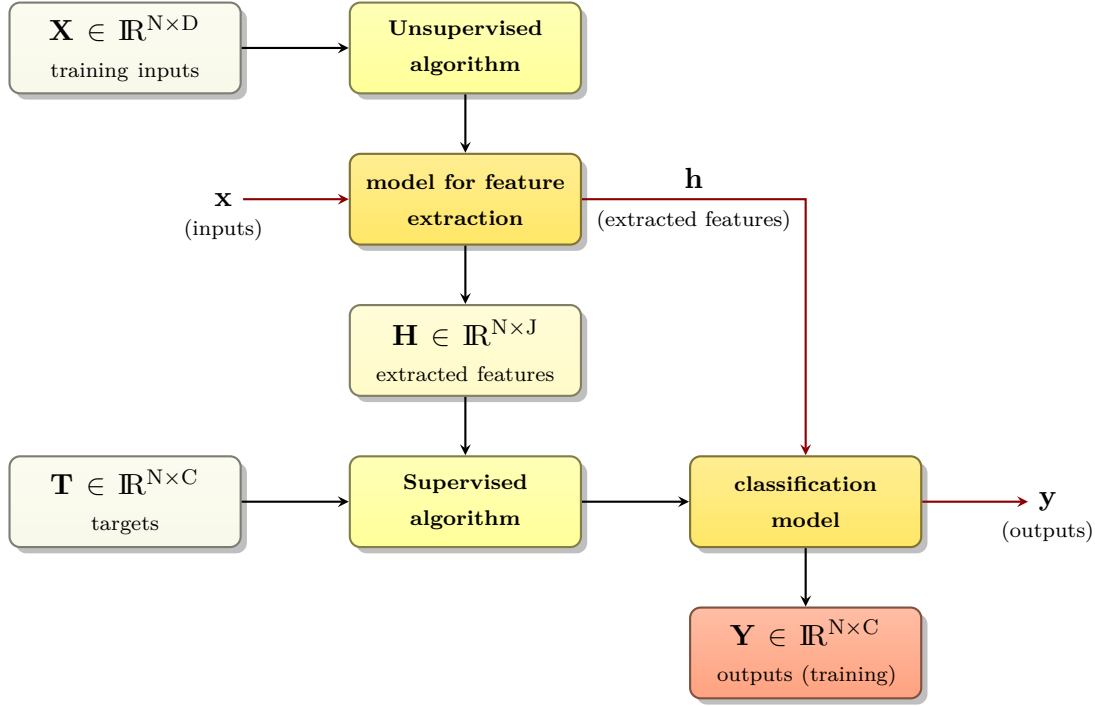
Accordingly, the goal of supervised learning algorithms consists of creating a dependency model that associates a specific output vector,  $\mathbf{y} \in \mathbb{R}^C$ , to each input vector,  $\mathbf{x} \in \mathbb{R}^D$ . Typically, algorithms relying on the Empirical Risk Minimization (ERM) principle, e.g. Back-Propagation (BP), adjust the model parameters, such that the resulting mapping function,  $f : \mathbb{R}^D \rightarrow \mathbb{R}^C$ , fits the training data. On the other hand, the Structural Risk Minimization (SRM), e.g. Support Vector Machines (SVMs), attempts to find the models with low Vapnik-Chervonenkis (VC) dimension [Osuna et al., 1997]. This is a core concept, which



**Figure 1.2:** Machine Learning paradigms.

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 [Bishop, 2006].

Let us assume that the training dataset input vectors,  $\{\mathbf{x}_1, \mathbf{x}_2, \dots, \mathbf{x}_N\}$ , 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,  $\{\mathbf{t}_1, \mathbf{t}_2, \dots, \mathbf{t}_N\}$ , 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



**Figure 1.3:** Combining supervised and unsupervised models.

vectors,  $\{\mathbf{y}_1, \mathbf{y}_2, \dots, \mathbf{y}_N\}$  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  $\mathbf{y} = [y_1, y_2, \dots, y_C] \in \mathbb{R}^C$ . 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 (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,  $\mathbf{h} = [h_1, h_2, \dots, h_J] \in \mathbb{R}^J$ , for each input vector,  $\mathbf{x} \in \mathbb{R}^D$ . By analogy, the extracted features' vectors,  $\{\mathbf{h}_1, \mathbf{h}_2, \dots, \mathbf{h}_N\}$ , 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 1.3.

## 1.4 Thesis Contributions

This Thesis contributes by partly answering the challenges addressed in the previous Sections. Specifically, it presents novel ways to extract relevant information from large dynamic datasets using scalable supervised, unsupervised and semi-supervised ML algorithms. Its primary objective consists of providing methods and instruments for significantly reducing the amount of time necessary to build accurate ML models

from sizable volumes of data, providing the means for handling “Big Data”. The following summarizes the list of contributions:

- **Main Scientific contributions**

- A new adaptive step size technique that improves considerably the training convergence of Restricted Boltzmann Machines (RBMs), 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 [Lopes and Ribeiro, 2013, Lopes and Ribeiro, 2012b, Lopes et al., 2012b] (see Section 5.2.3).
- 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 [Lopes and Ribeiro, 2011b] (see Section 5.1.3).
- A new incremental instance-based learning algorithm, Incremental Hypersphere Classifier (IHC), 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 [Lopes and Ribeiro, 2011d, Lopes and Ribeiro, 2011e, Lopes et al., 2012a] (see Section 4.3).
- A 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 [Lopes and Ribeiro, 2012a, Lopes and Ribeiro, 2011f, Lopes and Ribeiro, 2010d] (see Section 4.2).

- **GPU Computational Framework**

- An open-source GPU ML software library (GPUMLib – GPU Machine Learning Library) 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 [Lopes and Ribeiro, 2011c, Lopes et al., 2010] (see Section 2).
- A GPU parallel implementation of the Back-Propagation (BP) and Multiple Back-Propagation (MBP) algorithms, which reduces considerably the long training times of these types of NNs [Lopes and Ribeiro, 2011a, Lopes and Ribeiro, 2009b, Lopes and Ribeiro, 2010c, Lopes and Ribeiro, 2009a, Lopes and Ribeiro, 2009c] (see Section 4.1.3).
- 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) [Lopes and Ribeiro, 2011f] (see Section 4.2.4).
- An Autonomous Training System (ATS) that tries to mimic our heuristics for model selection (see Section 4.1.4). 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 [Lopes and Ribeiro, 2011a] (see Sections 4.1.5, 4.2.5 and 5.1.5).

- 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 [Lopes and Ribeiro, 2012c, Lopes and Ribeiro, 2010b] (see Section 5.1.4). The performance results of the GPU implementations excel by far those of the Central Processing Unit (CPU), yielding extremely high speedups [Lopes and Ribeiro, 2012c, Lopes and Ribeiro, 2010b, Lopes and Ribeiro, 2010a] (see Section 5.1.5).
- A GPU parallel implementation of the RBMs and DBNs, which accelerates significantly the (time consuming and computationally expensive) training process of these network architectures [Lopes and Ribeiro, 2013, Lopes et al., 2012b] (see Section 5.2.4). The RBM implementation incorporates a proposed adaptive step size procedure for tuning the learning parameters [Lopes and Ribeiro, 2013, Lopes and Ribeiro, 2012b, Lopes et al., 2012b] (see Section 5.2.3).
- **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 [Lopes et al., 2012a] (see Section 4.3.2).
  - 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 [Lopes and Ribeiro, 2011f] (see Section 4.2.5).
  - A new model for the detection of Ventricular Arrhythmias (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 [Lopes and Ribeiro, 2009a] (see Section 4.1.5).
  - A hybrid face recognition approach that combines the NMF-based methods with supervised learning algorithms [Lopes and Ribeiro, 2012c, Lopes and Ribeiro, 2010a] (see Sections 5.1.2 and 5.1.5). 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 (see Section 3.4), demonstrating its potential and usefulness, as well as evidencing robustness to different lighting conditions [Lopes and Ribeiro, 2012c, Lopes and Ribeiro, 2010a].

- 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 [Lopes and Ribeiro, 2013] (see Section 5.2.5).

## 1.5 Outline of the Thesis

The present Thesis is organized into six Chapters.

Following the general introduction in this Chapter, **Chapter 2** presents a new open-source GPU ML 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 GPUMLib and we detail its architecture.

**Chapter 3** describes the experimental setup configurations and the metrics used for performance evaluation, concerning the experiments carried out within this Thesis framework. Moreover, this Chapter also covers the benchmarks and case studies (used throughout the remainder of the Thesis) as well as the data preprocessing techniques applied to them.

**Chapter 4** presents techniques and algorithms for reducing the amount of time necessary to build supervised learning models. Accordingly, this Chapter covers the following algorithms and tools: Back-Propagation (BP) and Multiple Back-Propagation (MBP) GPU implementations; Autonomous Training System (ATS); Neural Selective Input Model (NSIM) (including its GPU implementation) and Incremental Hypersphere Classifier (IHC). Moreover, this Chapter also addresses three different real-world case studies involving the detection of Ventricular Arrhythmias (VAs), bankruptcy prediction and protein membership prediction.

**Chapter 5** deals with unsupervised and semi-supervised learning algorithms. Basically, two different unsupervised learning approaches, Non-Negative Matrix Factorization (NMF) and Deep Belief Networks (DBNs), and their corresponding GPU parallel implementations are addressed. Moreover, a new semi-supervised method, designated by Semi-Supervised NMF (SSNMF) is also presented. In addition, this Chapter also covers a hybrid NMF-based face recognition approach. Finally, an extensive experiment, involving the MNIST database of hand-written digits and the HHreco multi-stroke symbol database is presented (in order to gain a better understanding of the DBNs).

In **Chapter 6**, a general assessment of the research outcomes in face of the challenges and research questions established in the first Chapter is performed, followed by conclusions and possible directions of future work.



## CHAPTER 2

---

### GPU Machine Learning Library (GPUMLib)

---

---

|            |  |           |
|------------|--|-----------|
| <b>2.1</b> | <b>Introduction</b>  | <b>14</b> |
| <b>2.2</b> | <b>A Review of GPU Parallel Implementations of ML Algorithms</b> | <b>17</b> |
| <b>2.3</b> | <b>GPU Computing</b>   | <b>19</b> |
| <b>2.4</b> | <b>Compute Unified Device Architecture (CUDA)</b>                | <b>20</b> |
| 2.4.1      | CUDA Programming Model   | 20        |
| 2.4.2      | CUDA architecture  | 24        |
| <b>2.5</b> | <b>GPUMLib architecture</b>                                      | <b>28</b> |
| <b>2.6</b> | <b>Summary</b>   | <b>35</b> |

---

To cope with the increasingly large repositories of data, ML algorithms often demand prohibitive computational resources. In this context, the GPU represents a novel and compelling solution for this problem, due to its inherent high-parallelism. Unfortunately, few ML algorithms have been implemented on the GPU and most are not openly shared, posing difficulties for those aiming to take advantage of this architecture. To mitigate this problem, this Chapter describes a new open-source library (GPUMLib), developed as part of this Thesis, that aims to provide the building blocks for the development of efficient GPU ML software.

This Chapter is structured as follows. Section 2.1 argues for the need of an open-source GPU ML library. Section 2.2 presents an overview of the open-source and proprietary ML algorithms implemented on the GPU, prior to the development of GPUMLib. Section 2.3 focuses on the evolution of the GPU from a fixed-function device, designed to accelerate specific tasks, into a general-purpose computing

device. Section 2.4 details the CUDA programming model and architecture, which was used to develop GPUMLib. Finally, Section 2.5 describes the GPUMLib architecture and Section 2.6 summarizes this Chapter.

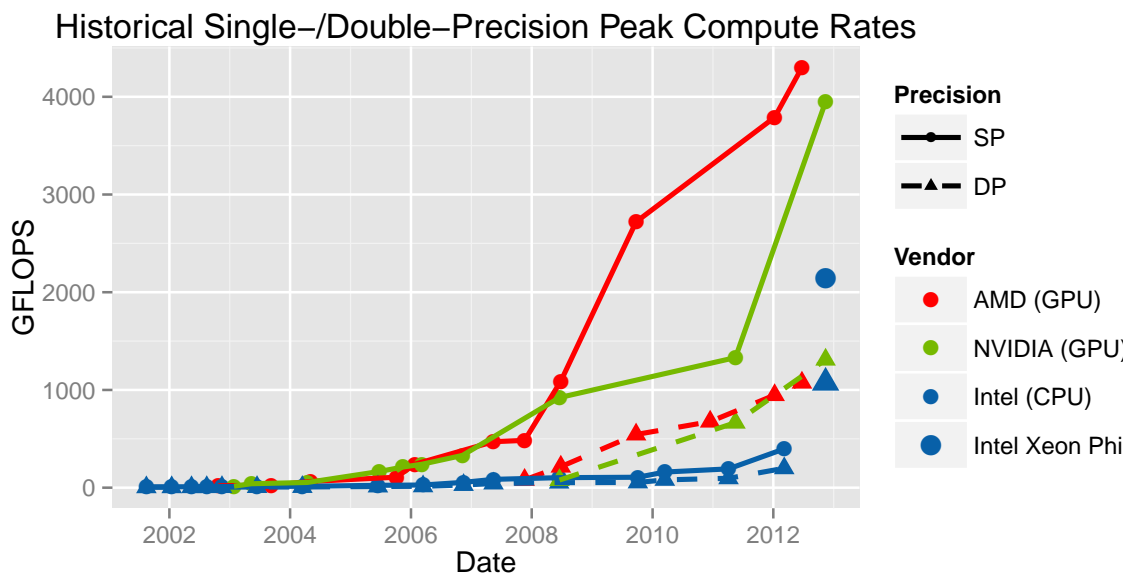
## 2.1 Introduction

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 [Hey et al., 2009]. 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 [Lopes and Ribeiro, 2011c]. Usually obtaining the information represents only a fraction of the time and effort needed to analyze it [Hey et al., 2009]. 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 [Lopes et al., 2010].

Although at present there are plentiful excellent toolkits which provide support for developing ML software in several environments (e.g. Python, R, Lua, Matlab) [King, 2009], 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 [Lopes and Ribeiro, 2011c].

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 [Owens et al., 2008]. 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 [Jang et al., 2008]. 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



**Figure 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).

in the future. This aspect is depicted in Figure 2.1, updated from Owens et al. [Owens et al., 2007], which shows that the GPU peak performance is growing at a much faster pace than the corresponding CPU performance<sup>1</sup>. Typically, the GPU performance is doubled every 12 months while the CPU performance doubles every 18 months [Zhongwen et al., 2005].

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 [Lopes and Ribeiro, 2009a]). Such characteristics trigger the interest of the scientific community who successfully mapped a broad range of computationally demanding problems to the GPU [Owens et al., 2008]. As a result, the GPU represents a credible alternative to traditional microprocessors in the high-performance computer systems of the future [Owens et al., 2008].

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 [Owens et al., 2008]. 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 our view, the GPU represents the most compelling option,

<sup>1</sup> Figure 2.1 is a courtesy of Professor John Owens, from the University of California, Davis, USA.

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 [Steinkraus et al., 2005, Brandstetter and Artusi, 2008]. 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 [Che et al., 2008b]. Moreover, although FPGAs can potentially yield the best performance results [Che et al., 2008b], recently several studies have concluded that GPUs are not only easier to program, but they also tend to outperform FPGAs in scientific computation tasks [Zhang et al., 2009]. 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 [Abramov et al., 2010]. 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 [Garland and Kirk, 2010, Steinkraus et al., 2005, Catanzaro et al., 2008].

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 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 [Stamatopoulos et al., 2012].

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 [Schaa and Kaeli, 2009]. 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 [Schaa and Kaeli, 2009].

The scientific breakthroughs of the future will undoubtedly be powered by advanced computing capabilities that will allow to manipulate and explore massive datasets [Hey et al., 2009]. 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 [Hey et al., 2009].

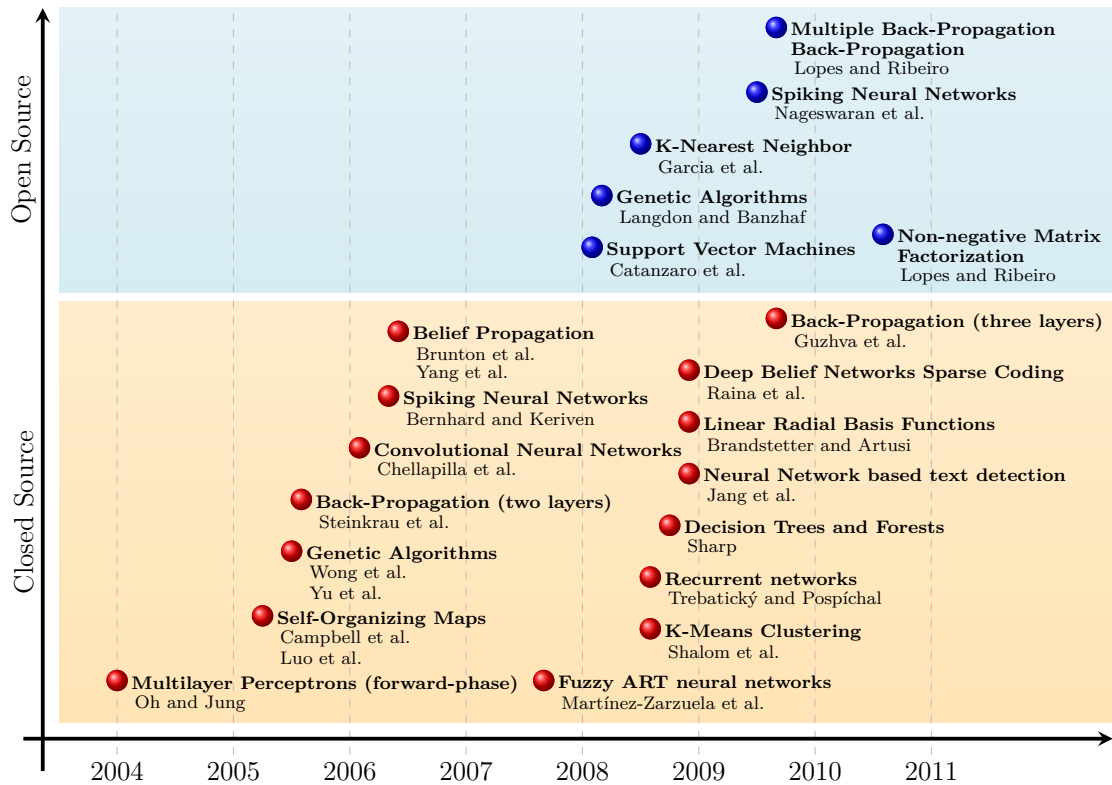
Overtime, a large body of powerful algorithms, suitable for a wide range of applications, has been developed in the field of ML. Unfortunately, the true potential of these methods has not been fully capitalized on, since existing implementations are not openly shared, resulting in software with low usability and weak interoperability [Sonnenburg et al., 2007].

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 [Sonnenburg et al., 2007]. 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 [Sonnenburg et al., 2007].

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 [Sonnenburg et al., 2007].

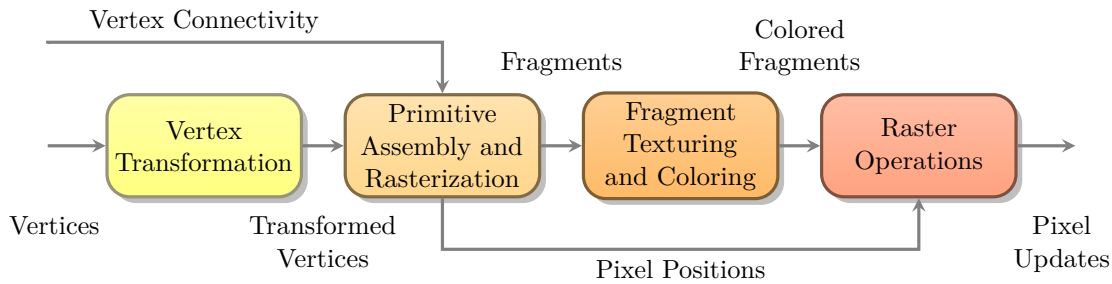
## 2.2 A Review of GPU Parallel Implementations of ML Algorithms

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 [Bohn, 1998, Oh and Jung, 2004, Campbell et al., 2005, Luo et al., 2005, Steinkraus et al., 2005, Wong et al., 2005, Yu et al., 2005, Zhongwen et al., 2005, Bernhard and Keriven, 2006, Brunton et al., 2006, Chellapilla et al., 2006, Yang et al., 2006, Harding and Banzhaf, 2007, Martínez-Zarzuola et al., 2007, Brandstetter and Artusi, 2008, Catanzaro et al., 2008, Do et al., 2008, Garcia et al., 2008, Grauer-Gray et al., 2008, Jang et al., 2008, Lahabar et al., 2008, Langdon and Banzhaf, 2008, Shalom et al., 2008, Sharp, 2008, Trebatický and Pospíchal, 2008, Čerňanský, 2009, Guzhva et al., 2009, Lopes and Ribeiro, 2009b, Nageswaran et al., 2009, Raina et al., 2009, Robilliard et al., 2009, Xu et al., 2009, Lopes and Ribeiro, 2010b].



**Figure 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 necessarily imply being an excellent programmer [Sonnenburg et al., 2007]. Additionally, the GPU programming model is significantly different from the traditional models [Garland and Kirk, 2010, Owens et al., 2008] 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



**Figure 2.3:** Graphics hardware pipeline.

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 [Li et al., 2013, Giannesini and Saux, 2012, Langdon, 2011, Herrero-Lopez, 2011], genetic algorithms [Cavuoti et al., 2014, Cavuoti et al., 2013, Cano et al., 2012, Chitty, 2012], belief propagation [Xiang et al., 2012],  $k$ -means clustering and  $k$ -nearest neighbor ( $k$ -nn) [Jian et al., 2013], particle swarm optimization [Hung and Wang, 2012], ant colony optimization [Cecilia et al., 2013], random forest classifiers [Essen et al., 2012] and sparse Principal Component Analysis (PCA) [Richtárik et al., 2012]. 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 [Owens et al., 2007]. 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 [Fernando and Kilgard, 2003]. 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 [Owens et al., 2007]. 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 [Owens et al., 2007].

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 [Che et al., 2008a]. 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 [NVIDIA, 2009]. 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 [Owens et al., 2007].

## 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 [Lopes and Ribeiro, 2011a].

### 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) [Che et al., 2008a]. 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



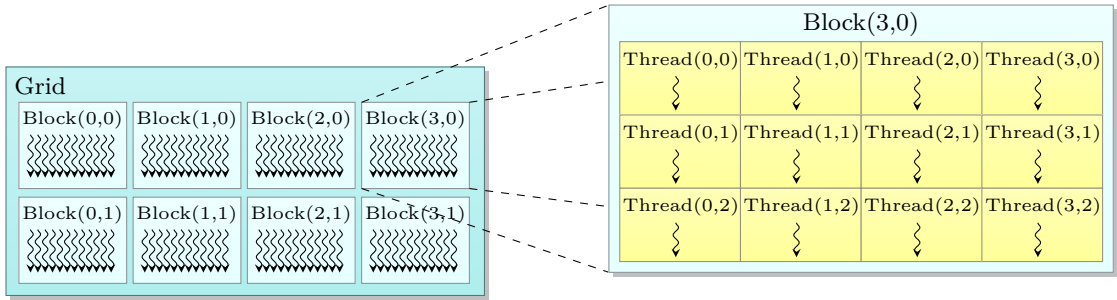
**Table 2.1:** Principal technical specifications according to the CUDA device compute capability.

| Technical Specifications                      | Compute Capability |     |      |             |              |      |
|---|--------------------|-----|------|-------------|--------------|------|
|   | 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 27)        | 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 |              |      |

to the CUDA device compute capability. A complete list of the specifications can be found in the NVIDIA CUDA C programming guide [NVIDIA, 2012b]. Moreover, a list of the devices supporting each compute capability can be found at <http://developer.nvidia.com/cuda-gpus>.

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.

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,



**Figure 2.4:** Example of a kernel grid.

**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 27).    |

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 [Ryoo et al., 2008].

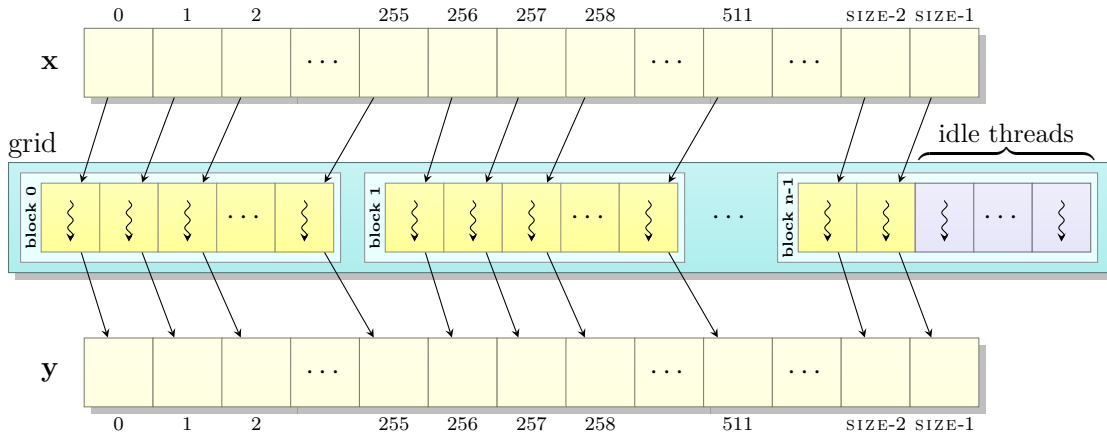
CUDA provides a set of intrinsic variables that kernels can use to identify the actual thread location in the domain, allowing each thread to work on separate parts of a dataset [Che et al., 2008a]. Table 2.2 identifies those built-in variables [NVIDIA, 2012b].

Listing 2.1 presents a simple kernel that computes the square of each element of vector  $\mathbf{x}$ , placing the result in vector  $\mathbf{y}$ . Kernel functions are declared by using the

**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];
}
    
```



**Figure 2.5:** Execution of the square kernel grid blocks (see Listings 2.1 and 2.2).

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 (`<<< ... >>>`). 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 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.

---

**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);

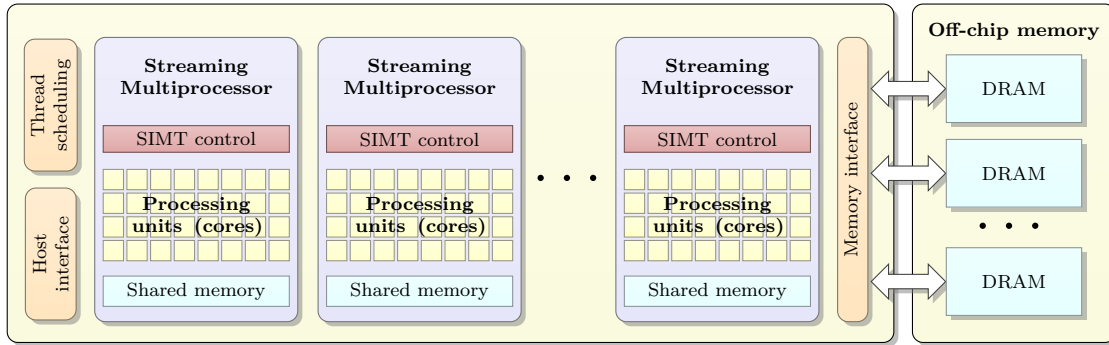
//...
```

---

## 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.

Figure 2.7 shows a detailed diagram of an SM [Halfhill, 2009]. 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 [Halfhill, 2009]. The cores lack their own general-purpose register files, L1



**Figure 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).

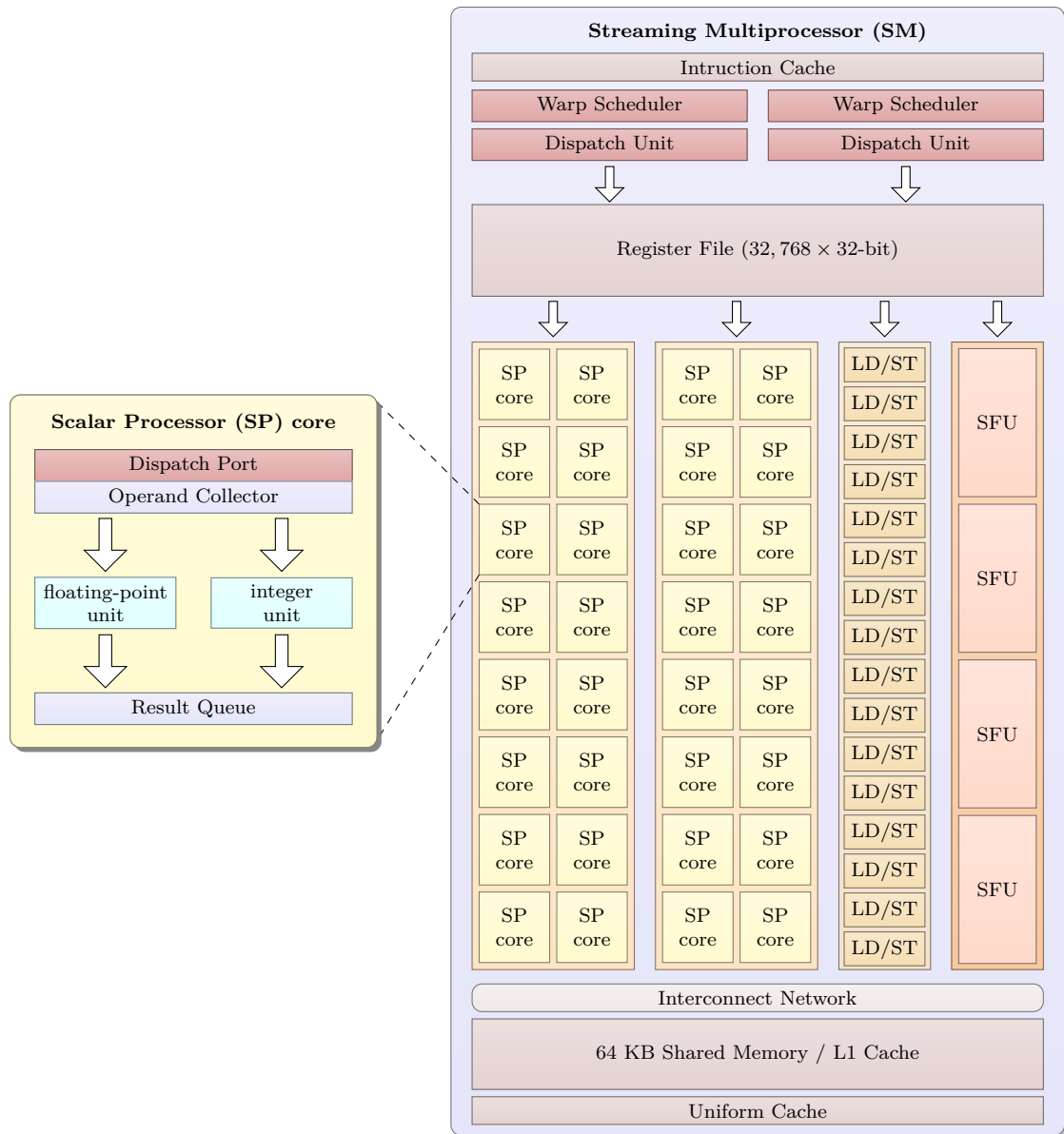
| Compute Capability                 | 1.x | 2.0 | 2.1 | 3.0 |
|------------------------------------|-----|-----|-----|-----|
| Number of cores per multiprocessor | 8   | 32  | 48  | 192 |

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) [Halfhill, 2009].

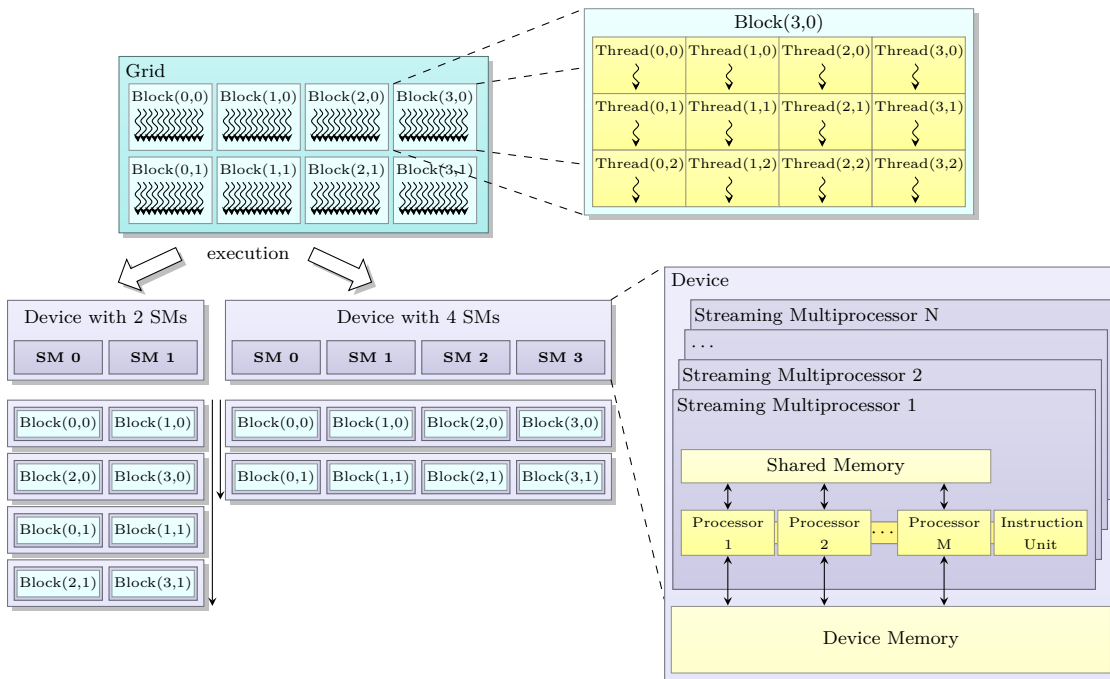
The SMs are optimized for single-precision floating point, since this is enough for traditional 3D graphics applications. In fact, the support for double-precision floating point, on NVIDIA GPUs, was only recently added in order to address the needs of scientific and High-Performance Computing (HPC) applications [NVIDIA, 2009]. Currently double-precision instructions are only supported on devices with compute capability 1.3 or above [NVIDIA, 2012b].

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) [Halfhill, 2009]. 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 [Halfhill, 2009].

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



**Figure 2.7:** Diagram of a Fermi Streaming Multiprocessor (SM).



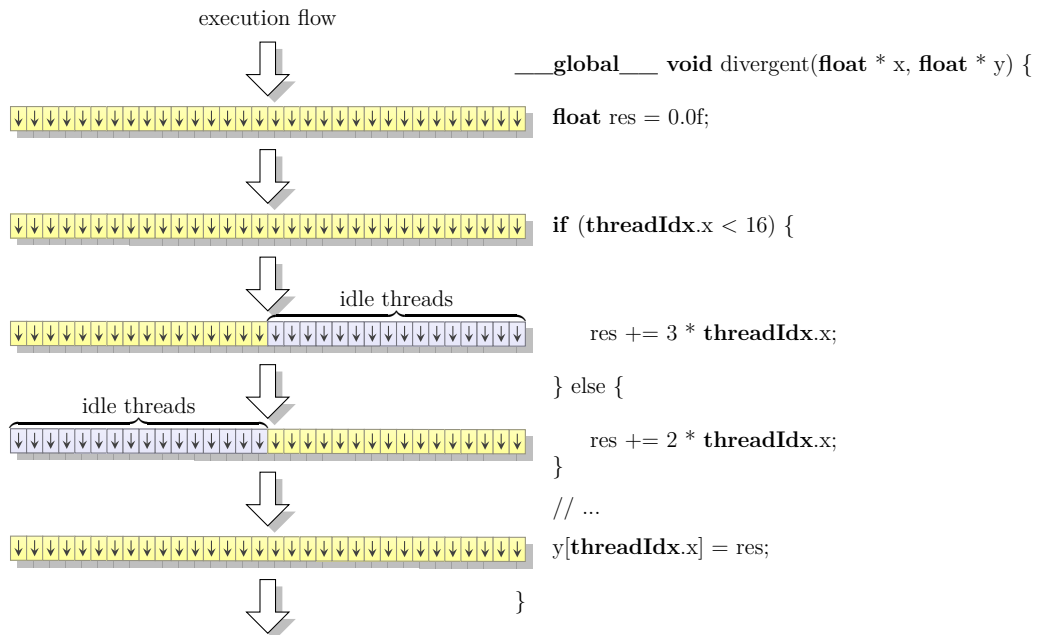
**Figure 2.8:** Execution of a kernel grid on different devices (GPUs).

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 [Halfhill, 2009]. 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.

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 [NVIDIA, 2012b].

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 [NVIDIA, 2012a]. The requirements for obtaining coalesced memory accesses vary according to the device compute capability. Devices with lower computing capabilities impose



**Figure 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.

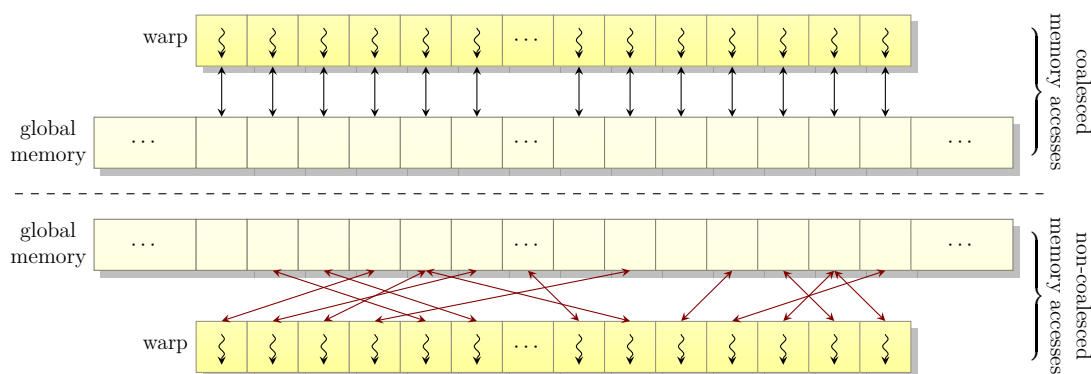
tighter restrictions than more recent ones. Nevertheless, it is usually possible to enforce coalesced memory accesses by guaranteeing that the threads access the memory in a sequential manner. Figure 2.10 illustrates both coalesced and non-coalesced memory access patterns. Non-coalesced patterns require additional memory transactions that ultimately reduce the instruction throughput [NVIDIA, 2012b].

In Ryoo et al. [Ryoo et al., 2008] 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 GPULib architecture

The GPULib 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





**Figure 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.

CUDA kernels that support the execution of ML algorithms on the GPU. Usually, 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 4.1.3).

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 seemingly 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 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-

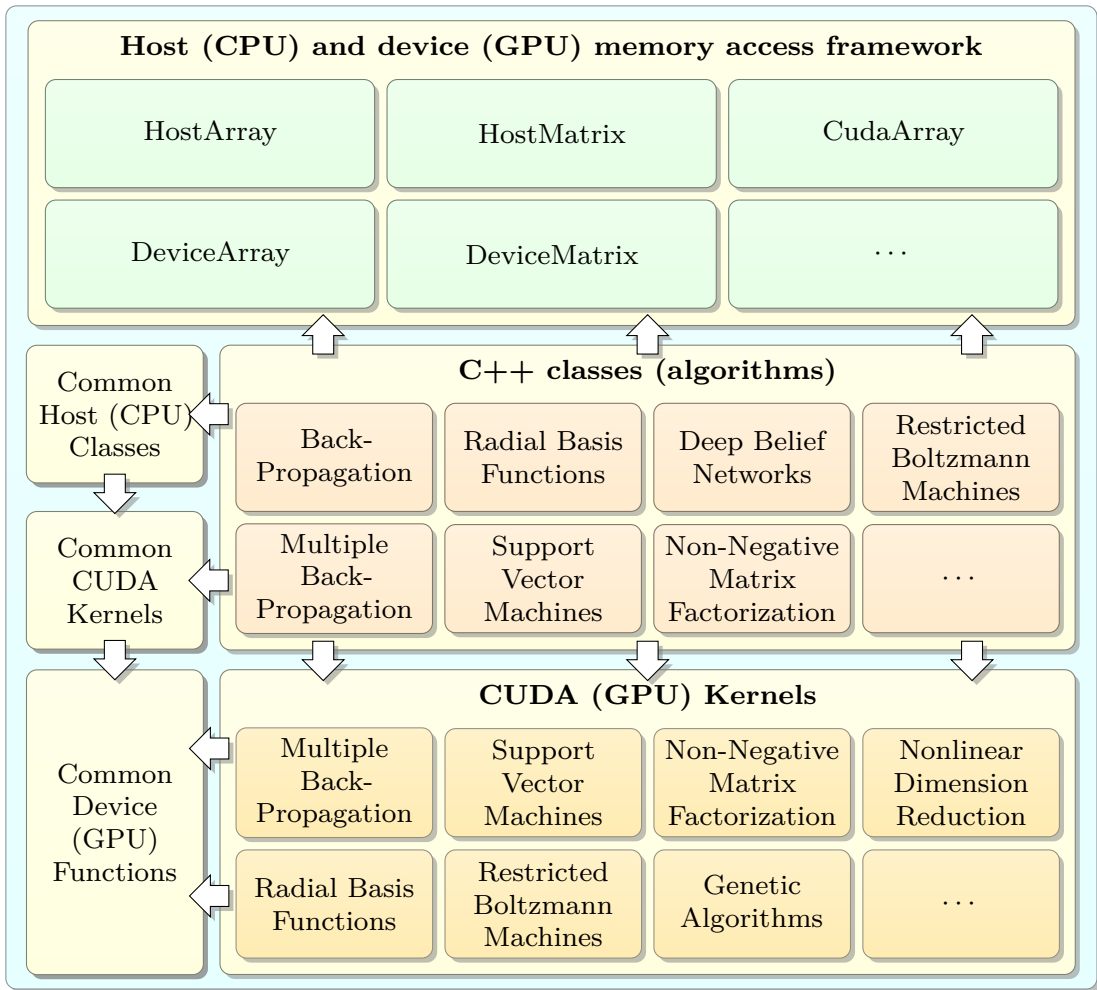


Figure 2.11: Main components of the GPUMLib.

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();

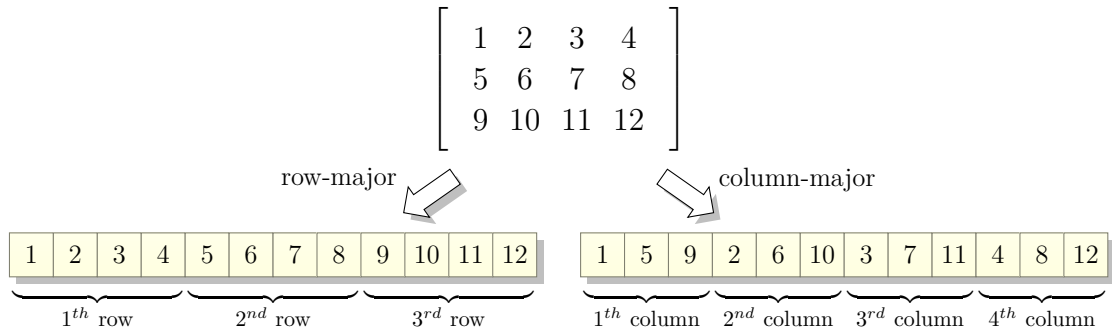
//...
```

---

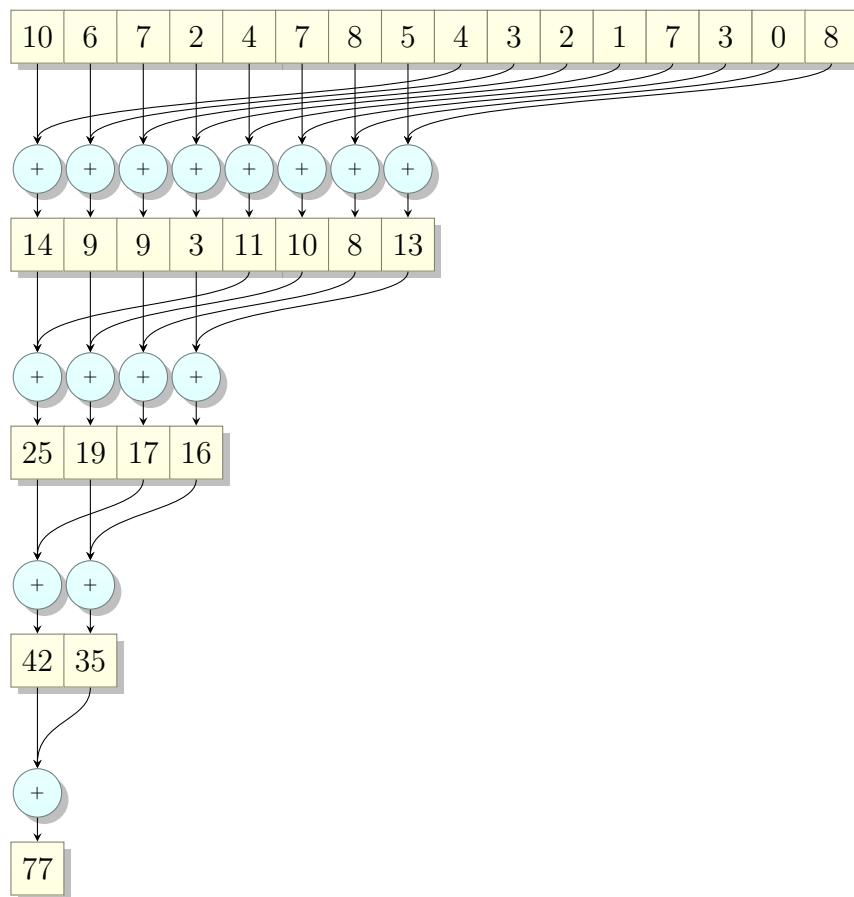
major, the elements of each row are placed together in sequential memory locations while the elements of each column are stored in non-sequential addresses. This implies that kernels can access the elements of each row in a coalesced manner, but they will be unable to do the same for the elements in 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.



**Figure 2.12:** Row-major versus column-major orders.



**Figure 2.13:** Example of a sum reduction.

**Table 2.5:** GPU parallel algorithms implemented in version 0.2.0 of GPUMLib.

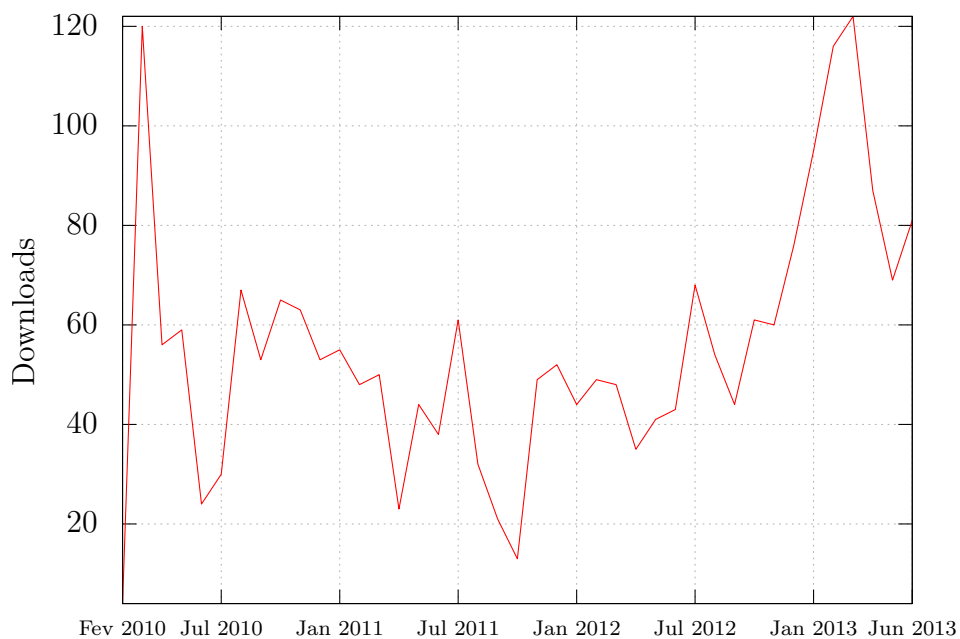
| <b>Algorithm/Architecture</b>           |
|---|
| Back-Propagation (BP)                   |
| Deep Belief Networks (DBNs)             |
| Multiple Back-Propagation (MBP)         |
| Non-Negative Matrix Factorization (NMF) |
| Radial Basis Function (RBF) networks    |
| Restricted Boltzmann Machines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

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 Ricardo Quintas [Quintas, 2010] and the SVMs were developed by João Gonçalves [Gonçalves, 2012] as part of their Master’s Thesis which was supervised by Professor Bernardete Ribeiro and co-supervised by this Thesis author.

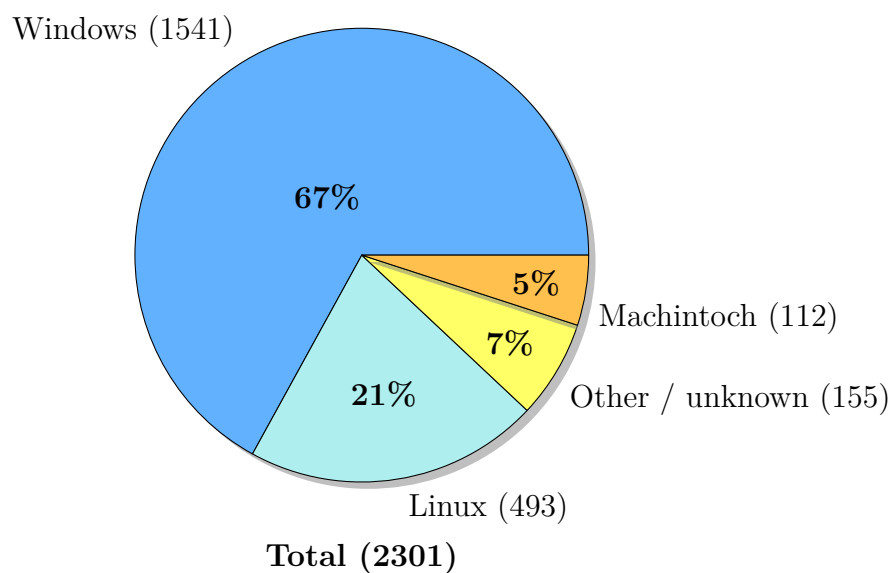
Since good documentation plays a major role in the success of any software library, GPUMLib provides extensive quality documentation and examples to ease its usage and development. Moreover, the library is practical, easy to use and extend, and does not require full understanding of the details inherent to GPU computing. The library is released under the GNU General Public 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.

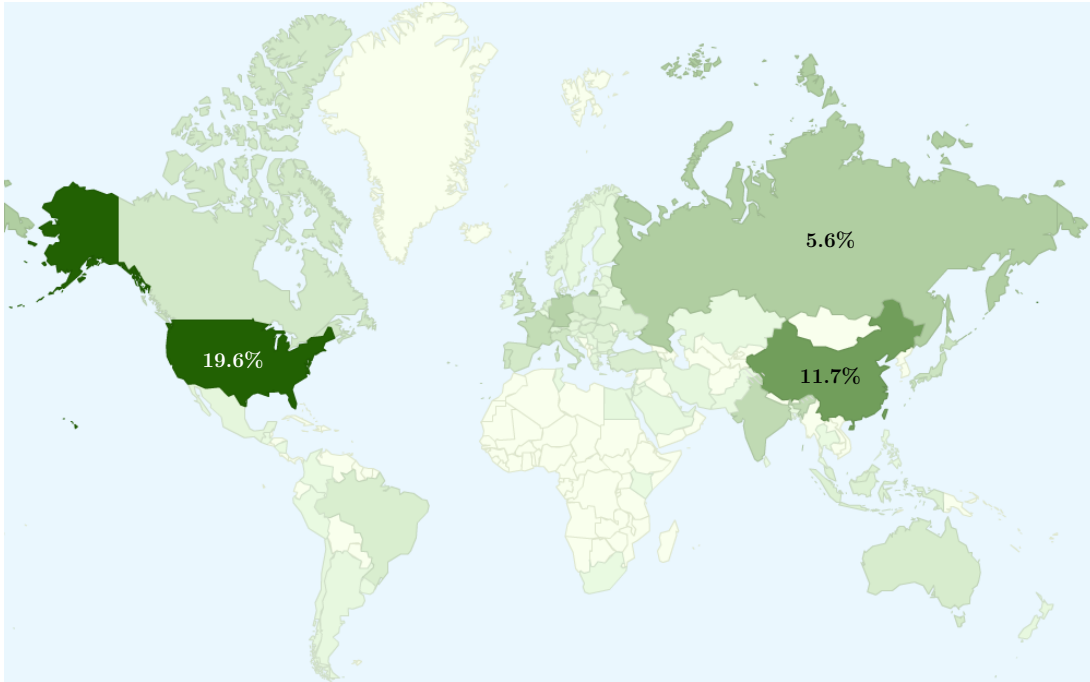
Since its release, GPUMLib has attracted the interest of numerous people (see Figure 2.14), using a wide-range of platforms (see Figure 2.15), benefiting researchers all over the world (see Figure 2.16). Moreover, GPUMLib has received a 5 star award from the soft82.com editors, which according to them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



**Figure 2.14:** Evolution of the number of downloads of GPULib.



**Figure 2.15:** Number of GPULib downloads according to the operating system.



**Figure 2.16:** Number of GPUMLib downloads per country. Regions with a higher number of downloads are represented with darker colors.

## 2.6 Summary

As problems grow increasingly complex and demanding, 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 [Lopes and Ribeiro, 2011c]. 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 alleviate this problem, we have developed a new open-source GPU ML library (GPUMLib) that can efficiently take advantage of the GPU parallel architecture and provide considerable speedups, allowing to easily select the building blocks necessary to create ML software [Lopes et al., 2010, Lopes and Ribeiro, 2011c].

Chapters 4 and 5 detail the GPUMLib parallel implementations of the algorithms developed as part of this Thesis and compare and analyze its results with the (corresponding results of the) standalone counterpart versions.





---

## Experimental Setup and Performance Evaluation

---



---

|   |           |
|---|-----------|
| <b>3.1 Hardware and Software Configurations . . . . .</b> | <b>37</b> |
| <b>3.2 Evaluation Metrics . . . . .</b>                   | <b>38</b> |
| <b>3.3 Validation . . . . .</b>                           | <b>41</b> |
| <b>3.4 Benchmarks . . . . .</b>                           | <b>43</b> |
| <b>3.5 Case Studies . . . . .</b>                         | <b>51</b> |
| <b>3.6 Data Preprocessing . . . . .</b>                   | <b>55</b> |
| <b>3.7 Summary . . . . .</b>                              | <b>57</b> |

---

This Chapter describes the experimental setup configurations and the metrics used for performance evaluation, concerning the experiments carried out within this Thesis framework. It is structured as follows. Section 3.1 details the hardware and software configurations that were used to conduct the experiments. Section 3.2 provides the metrics for evaluating the experiments' results. Section 3.3 discusses the methodologies for validating a model. Sections 3.4 and 3.5 receptively detail the benchmarks and the case studies that were used for conducting the experiments. Section 3.6 describes the data preprocessing techniques that were applied to the datasets. Finally Section 3.7 summarizes the Chapter.

### 3.1 Hardware and Software Configurations

In order to conduct the experiments, three different computer systems were used. Table 3.1 presents their main characteristics and Table 3.2 the principal characteristics of the systems' GPU devices. Since the systems are heterogeneous,

**Table 3.1:** Hardware and Software system main characteristics.

|                           | <b>Main Characteristics</b>  |
|---------------------------|--|
| <b>System 1 (8600 GT)</b> | Intel Core 2 6600 (2.4GHz)<br>NVIDIA GeForce 8600 GT<br>Windows Vista (x64)<br>4GB memory    |
| <b>System 2 (GTX 280)</b> | Intel Core 2 Quad Q 9300 (2.5GHz)<br>NVIDIA GeForce GTX 280<br>Windows 7 (x64)<br>4GB memory |
| <b>System 3 (GTX 460)</b> | Intel Dual-Core i5-2410M (2.7GHz)<br>NVIDIA GeForce GTX 460<br>Windows 7 (x64)<br>8GB memory |

**Table 3.2:** Main characteristics of the NVIDIA GeForce devices used in this work.

|                                  | <b>8600 GT</b> | <b>GTX 280</b> | <b>GTX 460</b> |
|----------------------------------|----------------|----------------|----------------|
| <b>Compute capability</b>        | 1.1            | 1.3            | 2.1            |
| <b>SMs</b>                       | 4              | 30             | 7              |
| <b>Number of cores</b>           | 32             | 240            | 336            |
| <b>Peak performance (GFLOPS)</b> | 113.28         | 933.12         | 940.8          |
| <b>Device memory</b>             | 512MB          | 1GB            | 1GB            |
| <b>Shared Memory per block</b>   | 16KB           | 16KB           | 48KB           |
| <b>Maximum threads per block</b> | 512            | 512            | 1024           |
| <b>Memory bandwidth (GB/sec)</b> | 22.4           | 141.7          | 112.5          |
| <b>Shading clock speed (GHz)</b> | 1.2            | 1.3            | 1.4            |

containing different CPUs and GPUs, in some cases we use the GPU name when referring to a specific system.

## 3.2 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.

### 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 (3.1):

$$speedup = \frac{time_S}{time_P} . \quad (3.1)$$

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 Root Mean Square Error (RMSE), which is given by (3.2):

$$RMSE = \sqrt{\frac{1}{NC} \sum_{i=1}^N \sum_{j=1}^C (Y_{ij} - T_{ij})^2} . \quad (3.2)$$

### Instance Selection Performance

When considering instance selection models, it is important to measure the storage reduction or space savings, which is given by (3.3):

$$storage = 1 - \frac{n}{N} . \quad (3.3)$$

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 3.3 presents the confusion matrix for a two class (binary) problem. A perfect classifier should only present non-zero values in the

**Table 3.3:** Confusion matrix for a binary classification problem.

| Class predicted | Actual class |          |
|-----------------|--------------|----------|
|                 | Positive     | Negative |
| Positive        | $tp$         | $fp$     |
| Negative        | $fn$         | $tn$     |

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 [Sokolova and Lapalme, 2009]. It represents the proportion of the predictions that are correct as given by (3.4):

$$accuracy = \frac{tp + tn}{tp + fp + fn + tn} . \quad (3.4)$$

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 (3.5) and (3.6) for a binary classification problem, are commonly used:

$$precision = \frac{tp}{tp + fp} , \quad (3.5)$$

$$recall = sensitivity = \frac{tp}{tp + fn} . \quad (3.6)$$

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 (3.7) for a binary classification problem:

$$Fmeasure = 2 \times \frac{precision \times recall}{precision + recall} . \quad (3.7)$$

Concerning binary classification problems, the precision, recall and F-measure metrics neglect the classification of negative examples [Sokolova and Lapalme, 2009]. Hence, depending on the problem, other measures such as the sensitivity (true positive rate), see (3.6), and specificity (true negative rate) may be more appropriate. The latter is given by (3.8):

$$specificity = \frac{tn}{fp + tn} . \quad (3.8)$$

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 (3.9) and (3.10).

$$precision_M = \frac{\sum_{c=1}^C \frac{tp_c}{tp_c + fp_c}}{C}, \quad (3.9)$$

$$recall_M = \frac{\sum_{c=1}^C \frac{tp_c}{tp_c + fn_c}}{C}, \quad (3.10)$$

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 (3.7), 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.

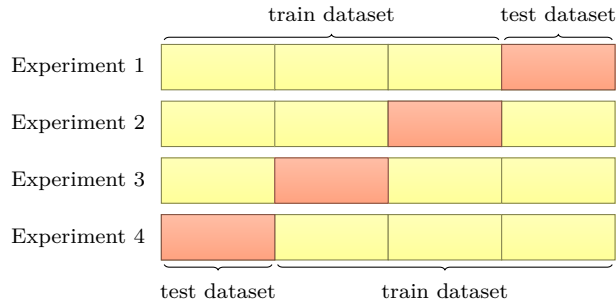
### 3.3 Validation

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 [Alpaydin, 2010, Bishop, 2006]. 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 [Refaeilzadeh et al., 2009]. 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



**Figure 3.1:** Experiments associated with a 4-fold cross-validation procedure.

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 3.1 shows the 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 [Refaeilzadeh et al., 2009].

#### **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 [Refaeilzadeh et al., 2009]. This strategy is used to validate the performance of the Incremental Hypersphere Classifier (IHC) in Section 4.3.

#### **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 [Refaeilzadeh et al., 2009].

### 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 Section 5.1.

### 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.

## 3.4 Benchmarks

Several benchmarks including face and character recognition databases are used to validate the algorithms and their respective models. Table 3.4 presents the main characteristics of the benchmark datasets, after preprocessing (see Section 3.6 for details about the preprocessing techniques used).

Since most of the benchmarks were obtained at the University of California, Irvine (UCI) Machine Learning Repository [Bache and Lichman, 2013] 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 is available at <http://www.c1.cam.ac.uk/research/dtg/attarchive/facedatabase.html> and includes the images of 40 different individuals. For each individual there are 10 distinct images. Hence, the database is composed of 400 images with  $112 \times 92$  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 3.2 presents randomly selected images from this database.

**Table 3.4:** Main characteristics of the benchmark datasets.

| Dataset (Benchmark)                  | Samples ( $N$ ) | Features ( $D$ ) | Classes ( $C$ ) |
|--------------------------------------|-----------------|------------------|-----------------|
| 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              |
| <i>Sinus cardinalis</i>              | 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              |





**Figure 3.2:** 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 3.3.

#### **Electricity Demand (Elec2)**

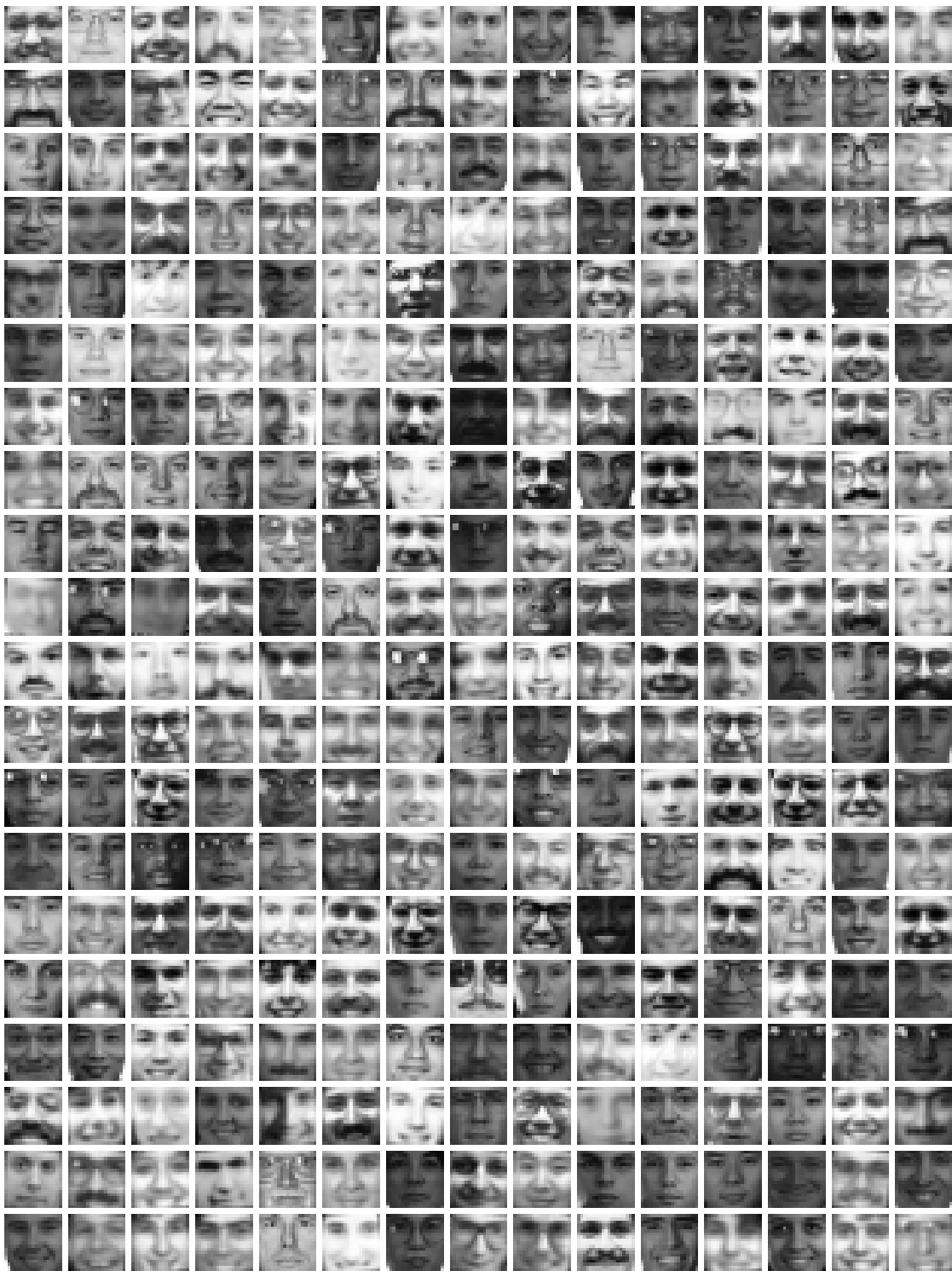
The *Electricity Demand* is a real-world problem that was obtained at [http://www.liaad.up.pt/area/jgama/ales/ales\\_5.html](http://www.liaad.up.pt/area/jgama/ales/ales_5.html). It contains data from the Australian New South Wales (NSW) Electricity Market, where the prices depend both on the demand and supply. The goal is to predict if the price will drop or increase [Gama et al., 2004].

#### **HHreco multi-stroke symbol database**

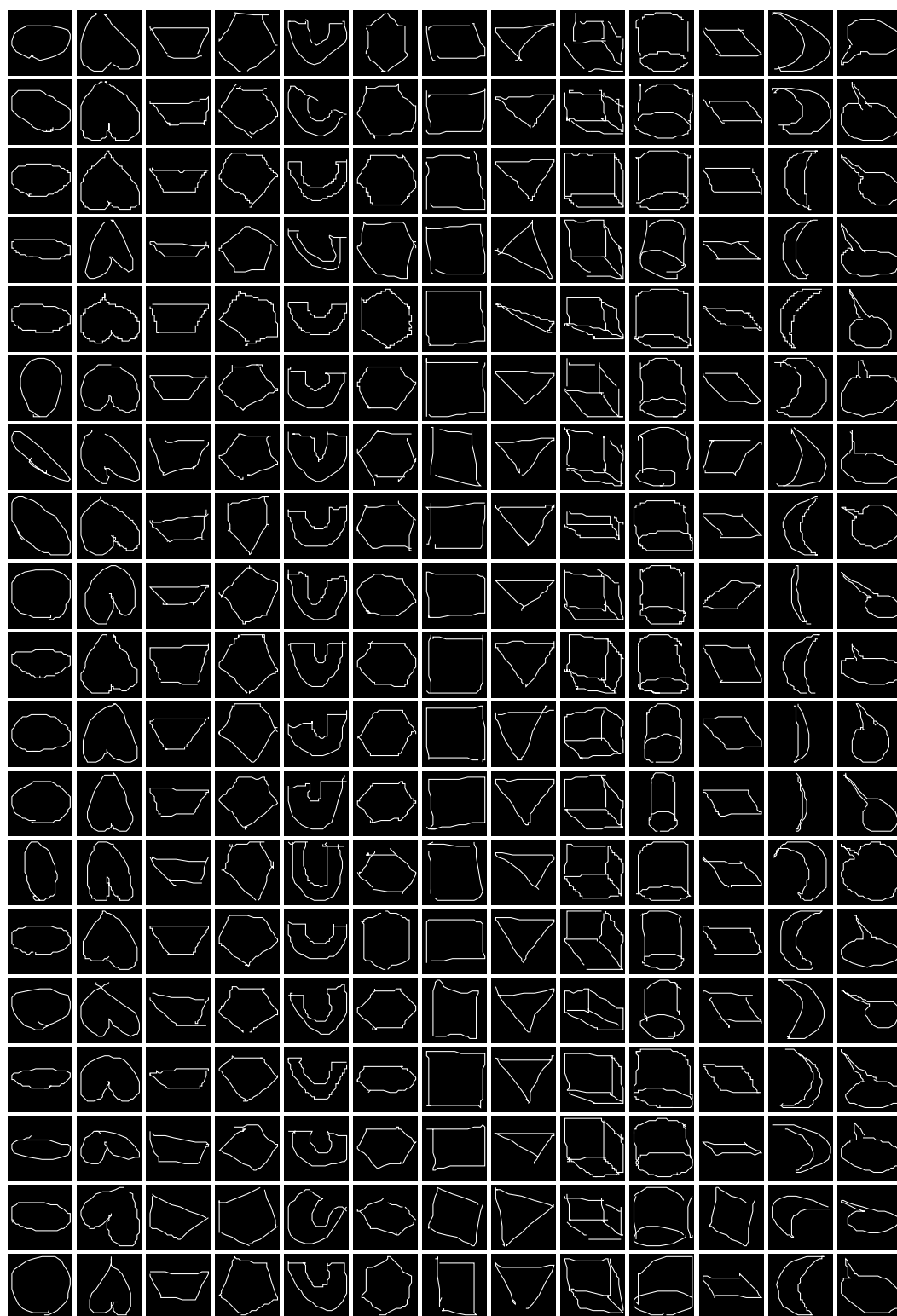
The HHreco multi-stroke symbol database, available at <http://embedded.eecs.berkeley.edu/research/hhreco/>, contains a total of 7,791 samples generated by 19 different persons. Overall, the database contains a total of 13 different symbol classes: ellipse, heart, trapezoid, pentagon, arch, hexagon, square, triangle, cube, cylinder, parallelogram, moon and callout [Hse and Newton, 2004]. 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 5.2.5). Figure 3.4 presents examples of the HHreco images.

#### **KDD Cup 1999 database**

The Knowledge Discovery and Data mining (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 types of attacks: Denial Of Service (DOS), unauthorized access from a remote machine (R2L), unauthorized access to local superuser privileges (U2R) and probing.



**Figure 3.3:** Randomly selected examples of the face images contained in the CBCL training dataset.



**Figure 3.4:** Examples of the HHreco multi-stroke images. Each column contains a symbol while each row contains the images drawn by one of the users.

### 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) [Jowell and the Central Coordinating Team, 2007, Žliobaitė, 2009]. 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 3.5 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:

$$\text{sinc}(x) = \frac{\sin(x)}{x}. \quad (3.11)$$

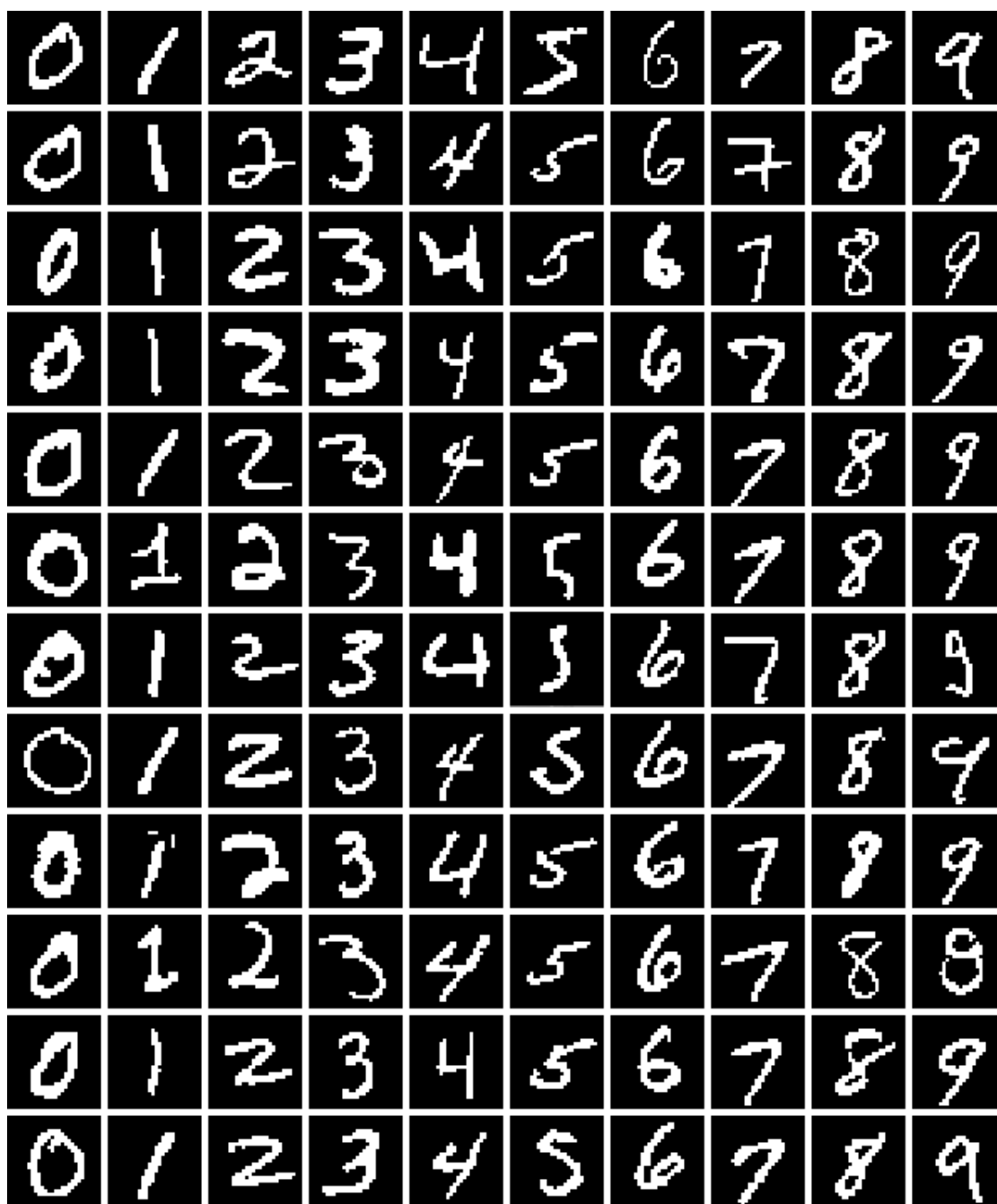
In order to build the training dataset we collected a total of 101 samples from the referred function, uniformly distributed in the interval  $[-10, 10]$ . Note that for  $x = 0$   $\text{sinc}(x)$  is considered to be 1. Figure 3.6 presents a graphical plot of this function.

### Two-Spirals

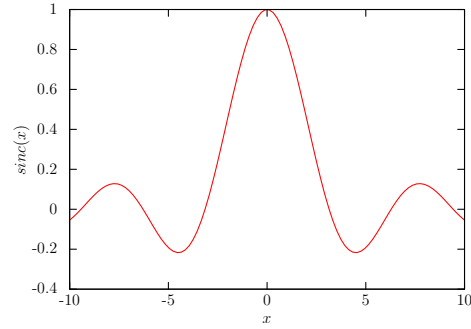
The *two-spirals* benchmark, obtained from the Carnegie Mellon University (CMU) learning benchmark archive, which is available at <http://www.cs.cmu.edu/afs/cs/project/ai-repository/ai/areas/neural/bench/cmu/>, is considered to be an extremely hard problem to solve for algorithms of the BP family [Fahlman and Lebiere, 1990]. 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 3.7.

### Yale Face Database

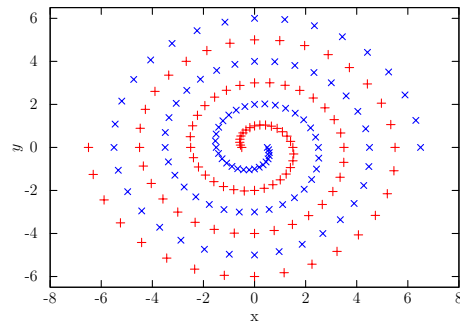
The Yale face database, available at <http://cvc.yale.edu/projects/yalefaces/yalefaces.html>, is comprised of 165 gray-scale images encompassing 15 individuals. Each individual appears in 11 images, each representing a different



**Figure 3.5:** 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.



**Figure 3.6:** *Sinus Cardinalis* function.



**Figure 3.7:** Two spirals dataset.

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  $64 \times 64$  pixels. Figure 3.8 shows the Yale face database images.

## 3.5 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 3.5 presents their main characteristics.

**Table 3.5:** 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               |
| Ventricular arrhythmias | 19,391          | 18               | 2               |



**Figure 3.8:** Yale face images. Each row contains the images of a specific individual and each column a different expression/ configuration.



### 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 [Lopes and Ribeiro, 2011f].

The datasets of this problem were obtained from a large database of French companies, containing information of an ample set of financial ratios spanning 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 3.6) as well as two more features: the number of employees and the turnover. Thus, altogether a total of 89 features were considered [Lopes and Ribeiro, 2011f]. Additional details on the construction of the dataset are given later in Section 4.2.5 (page 102).

### 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 [Morgado et al., 2011].

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 approximately 2% of the genes in all kinds of organisms encode peptidases and their homologues [Rawlings et al., 2010]. Hence, its detection is central to a better understand of their role in a biological system [Lopes et al., 2012a].

For the purpose of peptidase detection, a dataset constructed from the MEROPS [Rawlings et al., 2010] and the Structural Classification Of Proteins (SCOP) [Murzin et al., 1995] 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

**Table 3.6:** Financial ratios selected to create a bankruptcy model.

---

| <b>Financial ratios</b>               |                                      |
|---------------------------------------|--------------------------------------|
| 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 (%)               |                                      |

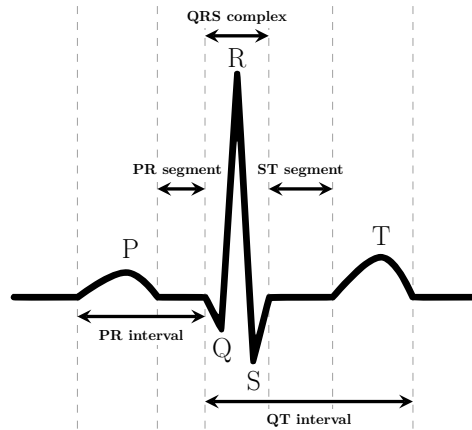
---

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) [Pereira et al., 2011].

The features of the protein primary structure were extracted using text mining techniques [Cheng et al., 2005]. 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’, ‘YGY’ and ‘YGY’. The Word Vector Tool (WVTool) library [Wurst, 2007] 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) [Correia et al., 2011].

### **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 [Marques, 2007] directly from the bio-signal Electrocardiographs (ECGs) data, available at the MIT-BIH Arrhythmia Database (<http://www.physionet.org/physiobank/>).



**Figure 3.9:** Typical ECG diagram of a normal sinus rhythm for a human heart.

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 [WolframAlpha, 2013]. 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 [Ribeiro et al., 2007].

A typical ECG tracing of an ordinary heartbeat consists of a P wave, a QRS complex and a T wave (observe Figure 3.9). 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 [Ribeiro et al., 2007].

Table 3.7 identifies the selected features from the ECG signal. For comparison purposes, we used the same training, test and validation datasets (each one with 19391 samples) that were used in Marques [Marques, 2007] and in Ribeiro et al. [Ribeiro et al., 2007].

## 3.6 Data Preprocessing

Datasets are often disturbed by problems of noise, bias and large variations in variables dynamic range [Lopes and Ribeiro, 1999]. Accordingly, the data preprocessing task assumes particular relevance for designing good generalization performance models [Kotsiantis et al., 2006a]. Typically, in the preprocessing

**Table 3.7:** 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        |

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 [Bishop, 2006].

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 [Cherkassky and Mulier, 2007].

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 (3.12):

$$x = \frac{x' - \min}{\max - \min}(nmax - nmin) + nmin, \quad (3.12)$$

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 [Kotsiantis et al., 2006a]. 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 [Zilu and Guoyi, 2009]. Figures 3.10, 3.11 and 3.12 show the histogram equalization results respectively for the AT&T, CBCL and Yale face images that are depicted in Figures 3.2, 3.3 and 3.8.

## **3.7 Summary**

In this Chapter, we have presented the main hardware and software experimental setup configurations. We have defined a set of metrics for evaluating the algorithms, their parallel implementations, and the resulting models. Moreover, we have specified the validation methodologies as well as the benchmarks and real-world case studies that are used throughout this Thesis to conduct the experiments. In addition, the data preprocessing techniques that are applied to the datasets were also specified.



**Figure 3.10:** AT&T face images after applying a histogram equalization to the original images presented in Figure 3.2.



**Figure 3.11:** CBCL face images after applying a histogram equalization to the original images presented in Figure 3.3.



**Figure 3.12:** Yale face images after applying a histogram equalization to the original images presented in Figure 3.8.



## CHAPTER 4

---

### Supervised algorithms

---

---

|            |   |            |
|------------|---|------------|
| <b>4.1</b> | <b>Multiple Back-Propagation (MBP)</b> . . . . .                | <b>62</b>  |
| 4.1.1      | Back-Propagation (BP) Algorithm . . . . .                       | 63         |
| 4.1.2      | Multiple Back-Propagation (MBP) Algorithm . . . . .             | 69         |
| 4.1.3      | GPU Parallel Implementation . . . . .                           | 75         |
| 4.1.4      | Autonomous Training System (ATS) . . . . .                      | 79         |
| 4.1.5      | Results and Discussion . . . . .                                | 79         |
| <b>4.2</b> | <b>Neural Selective Input Model (NSIM)</b> . . . . .            | <b>91</b>  |
| 4.2.1      | Missing Data Mechanisms . . . . .                               | 93         |
| 4.2.2      | Methods for Handling MVs in Machine Learning . . . . .          | 94         |
| 4.2.3      | NSIM Proposed Approach . . . . .                                | 97         |
| 4.2.4      | GPU Parallel Implementation . . . . .                           | 99         |
| 4.2.5      | Results and Discussion . . . . .                                | 99         |
| <b>4.3</b> | <b>Incremental Hypersphere Classifier (IHC)</b> . . . . .       | <b>104</b> |
| 4.3.1      | Proposed Incremental Hypersphere Classifier Algorithm . . . . . | 106        |
| 4.3.2      | Results and Discussion . . . . .                                | 110        |
| <b>4.4</b> | <b>Summary</b> . . . . .  | <b>121</b> |

---

Supervised learning is by far the most extensively used ML learning paradigm [Murphy, 2012]. Algorithms belonging to this category attempt to estimate an unknown mapping function by creating models that fit the training data samples, which include a set of previously observed inputs,  $\mathbf{x}$ , and their corresponding

targets,  $\mathbf{t}$  [Cherkassky and Mulier, 2007]. In other words, the algorithms generate a dependency model between the input,  $\mathbf{x} \in \mathbb{R}^D$ , and the output,  $\mathbf{y} \in \mathbb{R}^C$ , variables, using a set of associative correspondences,  $\mathbf{x} \mapsto \mathbf{t}$ , that the resulting model is expected to map.

Within the supervised framework, two different learning approaches can be considered: batch and incremental. In batch learning, the algorithms have full access to the complete training dataset, whenever they need. Conversely, incremental algorithms must update their models with each new observation and thus are suitable for handling non-stationary data streams [Drugowitsch, 2008]. In this context, the order in which the samples are presented to the algorithms becomes relevant, imposing a bias and making the learning task more complex and difficult. Notwithstanding this, the potential for extracting real-time information from large and dynamic data repositories (and the inherent competitive advantages) outweighs the drawback of obtaining models with lower generalization performance (as compared to state-of-the-art batch models).

Considering the goal of handling large volumes of data, in this Chapter we address both a batch and incremental supervised learning algorithms. Accordingly, Section 4.1 presents a GPU implementation of the Back-Propagation (BP) and Multiple Back-Propagation (MBP) algorithms. Moreover, an Autonomous Training System (ATS) that takes advantage of the aforementioned GPU implementations to find high-quality NN-based solutions, without human intervention, is also presented. In addition, Section 4.2 presents a novel Neural Selective Input Model (NSIM) that empowers NNs with the ability to handle directly Missing Values (MVs), which are common in large datasets. Section 4.3 presents a new incremental algorithm (Incremental Hypersphere Classifier (IHC)) that can also be used to select a representative subset of the original dataset. Finally, Section 4.4 concludes this Chapter and addresses directions for future work.

### 4.1 Multiple Back-Propagation (MBP)

Despite being motivated by the parallel processing capabilities of the human brain, artificial NNs (referred in this Thesis simply by NNs) have little in common with their biological counterparts [Piękniewski and Rybicki, 2004, Duch and Jankowski, 1999]. 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 [Kumar et al., 2013, Hui, 2011, Tang et al., 2007, Samarasinghe, 2007, Vonk et al., 1995, Widrow et al., 1994]. Unfortunately, building an NN solution is a computationally 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 [Lopes and Ribeiro, 2011a, Lopes and Ribeiro, 2009a]. Hence, creating GPU implementations of these algorithms is highly desirable.

### 4.1.1 Back-Propagation (BP) Algorithm

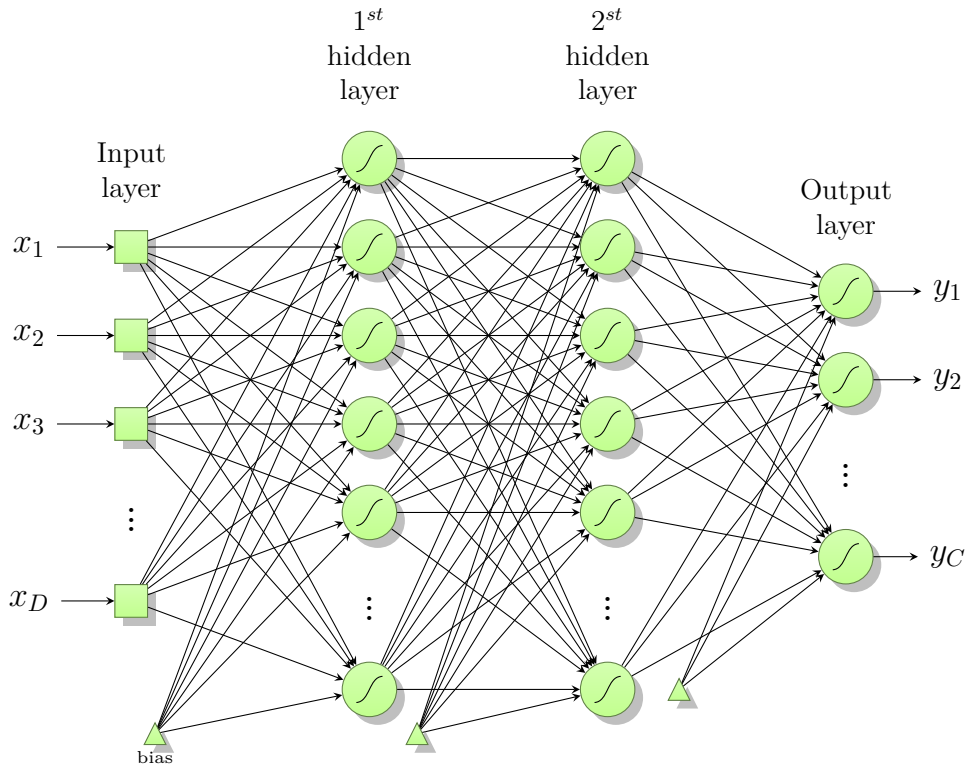
The BP is one of the most well-known and extensively used ML algorithms. In fact, it is so successful that over 90% of the real-world commercial and industrial NN applications use this algorithm [Munakata, 2008, Yuming and Yuanyuan, 2012].

#### Feed-Forward (FF) networks

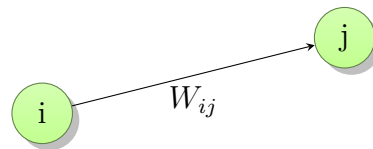
The BP is a supervised learning algorithm for training Feed-Forward (FF) NNs. These networks, also called Multi-Layer Perceptrons (MLPs), or BP networks (when trained with the BP algorithm), are comprised of several interconnected processing units (neurons) organized in layers, such that the information 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 4.1 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 4.2. 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 (i.e.  $x_i = y_i$ ), 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 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 be the weight of an extra connection, whose input signal remains constantly set to 1. This



**Figure 4.1:** Three-layer feed-forward network.

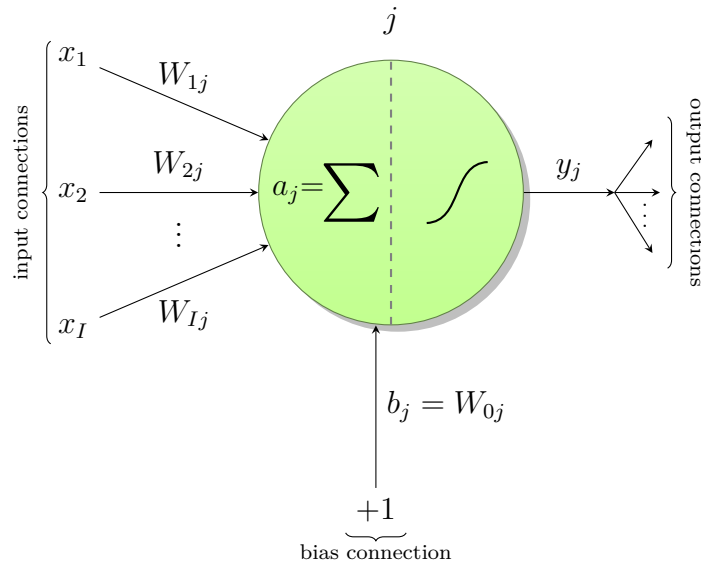


**Figure 4.2:** Connection between two neurons.

allows the BP algorithm to adjust the bias weight together with the remaining weights [Bishop, 2006, Piękniewski and Rybicki, 2004]. Accordingly, assuming that the neuron  $j$  contains  $I$  input connections, its activation,  $a_j$ , is given by (4.1):

$$a_j = \sum_{i=0}^I W_{ij} y_i . \quad (4.1)$$

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 4.3 illustrates this process.



**Figure 4.3:** Neuron architecture.

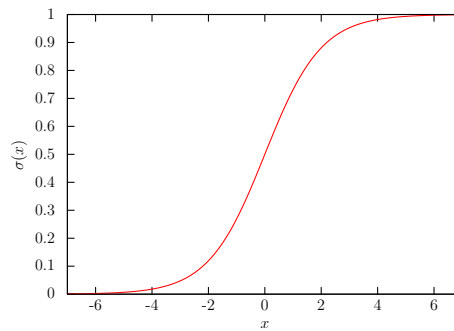
Accordingly, the output,  $y_j$ , of neuron  $j$  is given by (4.2):

$$y_j = \phi(a_j) = \phi\left(\sum_{i=0}^I W_{ij}y_i\right), \quad (4.2)$$

where  $\phi(x)$  is the neuron activation function. Typically, the (logistic) sigmoid function, given by (4.3), is used for this purpose:

$$\sigma(x) = \frac{1}{(1 + e^{-x})}. \quad (4.3)$$

The sigmoid is a non-linear and non-decreasing function, whose output is clamped between 0 and -1. Figure 4.4 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 [Shenouda, 2006, Piękniewski and Rybicki, 2004, Duch and Jankowski, 1999] as it reshapes the geometry of the transformations generated by the networks with implications in the training speed and generalization capabilities [Piękniewski and Rybicki, 2004]. Moreover, the activation function is strongly correlated with the number of adaptive parameters required to model complex decision borders [Duch and Jankowski, 1999]. Therefore its choice must be carefully 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 derivative is a very fast and straightforward process [Shenouda, 2006]. In addition,



**Figure 4.4:** Sigmoid function.

it is generally believed that the activity of biological neurons is regulated by a sigmoidal transfer function [Duch and Jankowski, 1999].

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 [Hornik et al., 1989, Cybenko, 1989, Funahashi, 1989].

### 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 [Bishop, 2006], 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<sup>1</sup>:

$$E = \frac{1}{2} \sum_{o=1}^C (t_o - y_o)^2, \quad (4.4)$$

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 minimize 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  $\eta$  as a learning rate factor, the weight change,  $\Delta W_{ij}$ , is given

---

<sup>1</sup>Online training mode is considered.

by (4.5):

$$\Delta W_{ij} = -\eta \frac{\partial E}{\partial W_{ij}} . \quad (4.5)$$

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 (4.6):

$$\begin{aligned} \Delta W_{ij} &= -\eta \frac{\partial E}{\partial a_j} \frac{\partial a_j}{\partial W_{ij}} \\ &= -\eta \frac{\partial E}{\partial a_j} y_i . \end{aligned} \quad (4.6)$$

By defining the local gradient  $\delta_j$  as in (4.7):

$$\delta_j = -\frac{\partial E}{\partial a_j} , \quad (4.7)$$

we can write (4.6) as (4.8):

$$\Delta W_{ij} = \eta \delta_j y_i . \quad (4.8)$$

Using the chain rule once again, (4.7) becomes (4.9):

$$\begin{aligned} \delta_j &= -\frac{\partial E}{\partial y_j} \frac{\partial y_j}{\partial a_j} \\ &= -\frac{\partial E}{\partial y_j} \phi'(a_j) . \end{aligned} \quad (4.9)$$

Thus, for an output neuron,  $j = o$ , the local gradient is given by (4.10):

$$\delta_o = (t_o - y_o) \phi'(a_o) . \quad (4.10)$$

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, considering a hidden neuron,  $j = h$ , and assuming the next layer is the output layer, the local gradient,  $\delta_h$ , can be obtained by applying the chain rule [Bishop, 2006]:

$$\begin{aligned} \delta_h &= -\phi'(a_h) \frac{\partial E}{\partial y_h} \\ &= -\phi'(a_h) \sum_{o=1}^C \frac{\partial E}{\partial a_o} \frac{\partial a_o}{\partial y_h} \\ &= -\phi'(a_h) \sum_{o=1}^C \frac{\partial E}{\partial a_o} \frac{\partial}{\partial y_h} \sum_{i=0}^I W_{io} y_i \\ &= -\phi'(a_h) \sum_{o=1}^C \frac{\partial E}{\partial a_o} W_{ho} . \end{aligned} \quad (4.11)$$

and using (4.7) we can finally write (4.11) as (4.12):

$$\delta_h = \phi'(a_h) \sum_{o=1}^C \delta_o W_{ho} . \quad (4.12)$$

Together (4.8), (4.10) and (4.12) allow to recursively update all the weights in the network. In practice, however, a momentum term,  $0 \leq \alpha < 1$ , is usually included in (4.8) in order to improve the algorithm's convergence. Accordingly, instead of (4.8) we can use (4.13):

$$\Delta W_{ij} = \eta \delta_j y_i + \alpha \Delta W_{ij}^{(\text{old})} . \quad (4.13)$$

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 (4.5). 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 [Haykin, 1998].

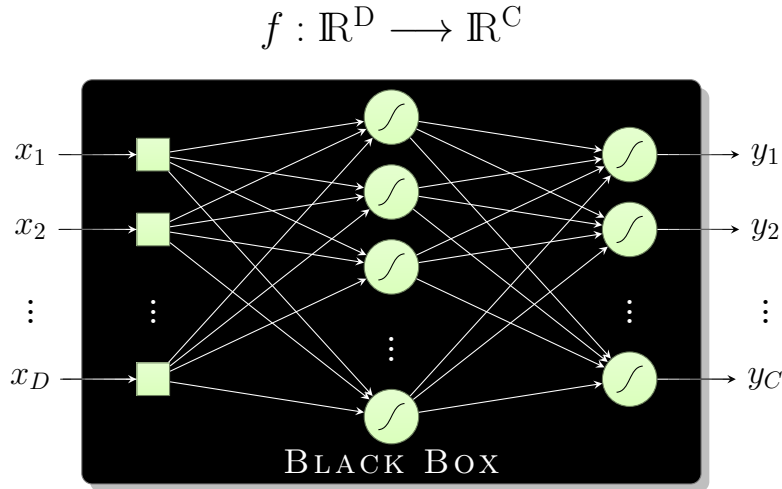
Another simple acceleration technique, although memory consuming, is the adaptive step size technique, which consists of using an individual learning rate (step size) parameter,  $\eta_{ij}$ , for each weight connection,  $W_{ij}$ , instead of a global learning rate. At each iteration, the step sizes  $\eta_{ij}$  are adjusted according to the successive signs of the gradient components, using (4.14) [Almeida, 1997]:

$$\eta_{ij} = \begin{cases} u\eta_{ij}^{(\text{old})} & \text{if } (\frac{\partial E}{\partial W_{ij}})(\frac{\partial E}{\partial W_{ij}})^{(\text{old})} > 0 \\ d\eta_{ij}^{(\text{old})} & \text{if } (\frac{\partial E}{\partial W_{ij}})(\frac{\partial E}{\partial W_{ij}})^{(\text{old})} < 0 \end{cases} \quad (4.14)$$

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 [Almeida, 1997].

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 [Almeida, 1997]. To avoid both situations, it is possible to implement a robustness method that is triggered when the error grows behind a predefined threshold (e.g. 0.1%). The above-mentioned method basically consists of [Almeida, 1997]: (i) setting the weights back to the values they had in the epoch with the lowest error achieved so





**Figure 4.5:** A neural network viewed as a black box system that maps  $D$  inputs into  $C$  outputs.

far; (ii) reducing the step size parameters by a pre-defined factor,  $0 < r < 1$ , (e.g. 0.5) and (iii) set the momentum memory to zero.

The combined procedure (adaptive step size, robustness method and momentum) results in a very effective training algorithm that works well in practice [Almeida, 1997].

### 4.1.2 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 mapping function,  $f : \mathbb{R}^D \longrightarrow \mathbb{R}^C$ , that attempts to fit the observed (training) data (see Figure 4.5).

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 [Džeroski et al., 2009, Wang, 2008, Tahir and Smith, 2010]. 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 [Yuksel et al., 2012, Džeroski et al., 2009]. 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 [Yuksel et al., 2012].

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.

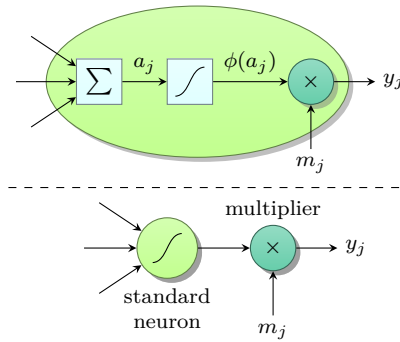
Nevertheless, it is possible to partition the input space without the proviso of explicit information. For example, the RBF networks perform an implicit partition of the input space by assigning localized neurons that respond only to the samples that are in the vicinity of its center. From the point 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) [Kanwisher, 2010].

### 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.

In order to specify the contribution of a given neuron,  $j$ , to the network output, we incorporate an importance factor,  $m_j$ , in the neuron equation that defines its relevance for the sample (*stimulus*) being presented to the network. Such neurons are designated by neurons with selective actuation [Lopes and Ribeiro, 2003, Lopes



**Figure 4.6:** Selective actuation neuron architecture.

and Ribeiro, 2001] and the equation governing its output is given by (4.15):

$$y_j = m_j \phi(a_j) = m_j \phi\left(\sum_{i=0}^I W_{ij} y_i\right). \quad (4.15)$$

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 [Lopes and Ribeiro, 2003, Lopes and Ribeiro, 2001].

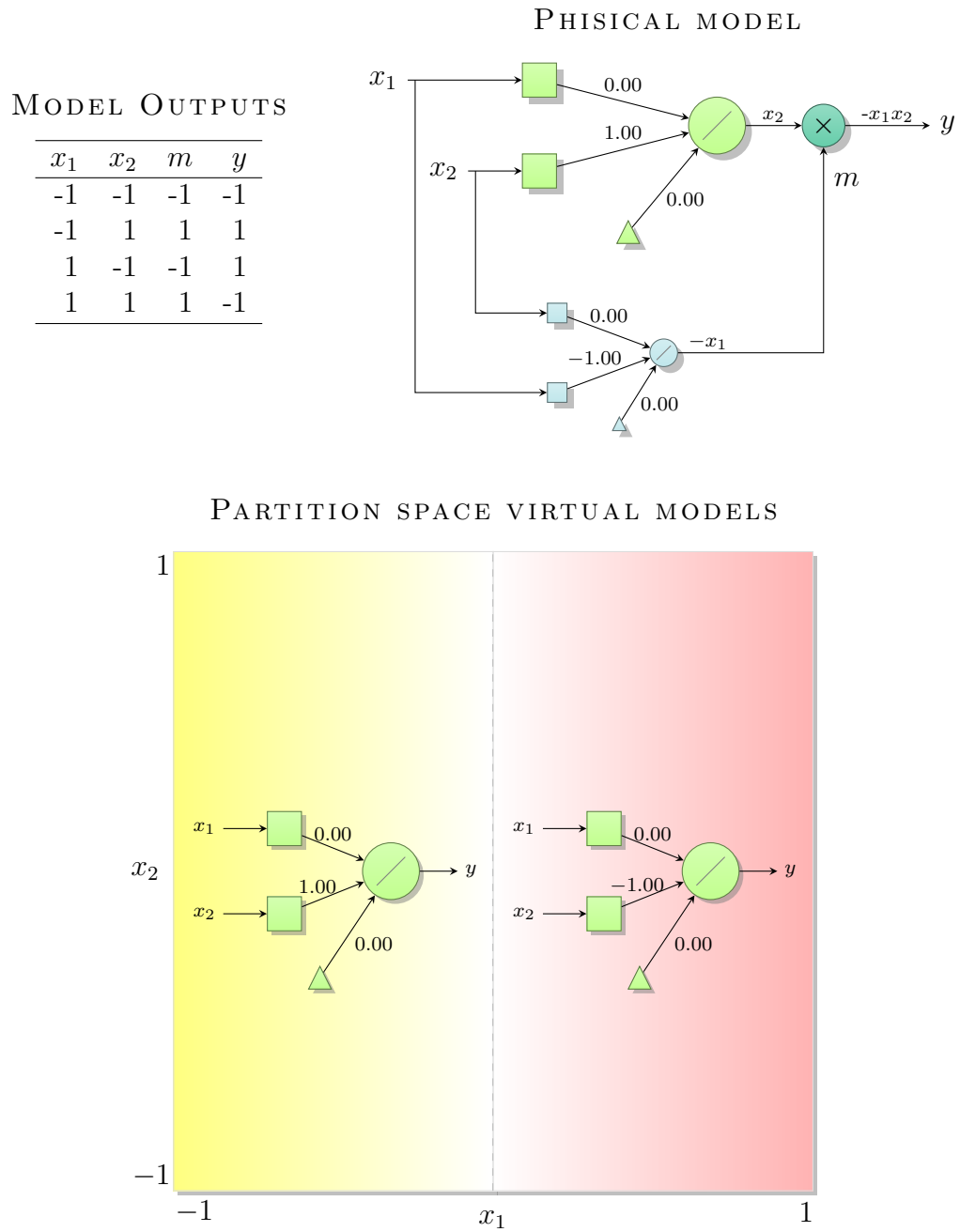
Notice that if we consider all the importance factors  $m_j$  to be constant and equal to one, i.e., if all neurons are equally important to the network regardless of the pattern being presented, then (4.15) becomes identical to the standard neuron output equation (4.2).

Figure 4.6 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.

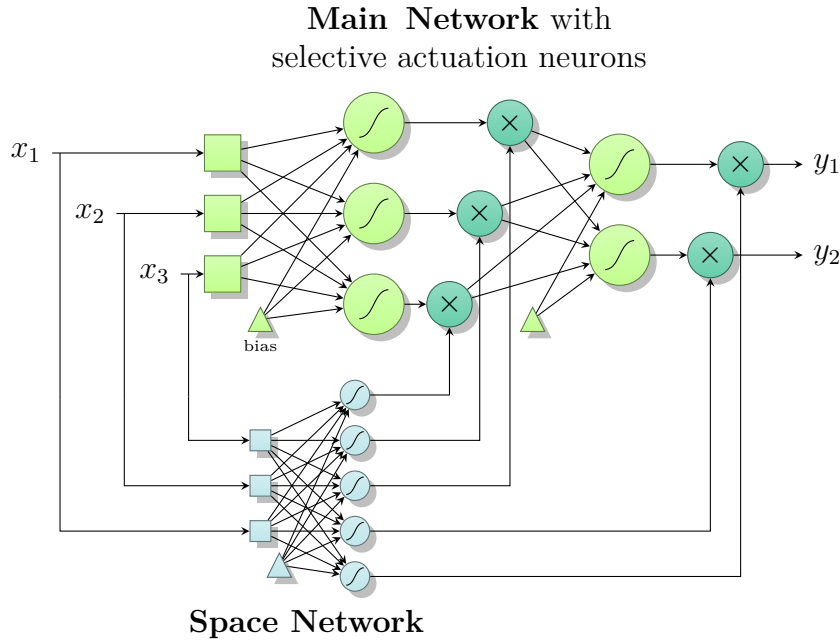
A neuron with selective actuation is inherently a non-linear device that is capable of solving the XOR problem, even when its constituent neurons use a linear activation function (see Figure 4.7).

### 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 [Lopes and Ribeiro, 2003, Lopes and Ribeiro, 2001].



**Figure 4.7:** Architecture of a selective actuation neuron, with linear activation functions, which solves the XOR problem.



**Figure 4.8:** Example of a multiple feed-forward network.

The resulting network is called a Multiple Feed-Forward (MFF) or an MBP network. Figure 4.8 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 neuron with selective actuation integrating those models, some data points can be interpreted as being more important than others [Lopes and Ribeiro, 2003, Lopes and Ribeiro, 2001].

### 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 [Lopes and Ribeiro, 2003, Lopes and Ribeiro, 2001].

In the MBP algorithm, the main network weights are adjusted according to the gradient descent method, using (4.8) as in the BP algorithm. However, due to the introduction of the importance factor, the local gradients for the output neurons  $\delta_o$  and for the hidden neurons  $\delta_h$  are now respectively given by (4.16) and (4.17):

$$\delta_o = (t_o - y_o)m_o\phi'(a_o), \quad (4.16)$$

$$\delta_h = m_h \phi'(a_h) \sum_{o=1}^C \delta_o W_{ho} . \quad (4.17)$$

Jointly, (4.8), (4.16) and (4.17) 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 (4.10) and (4.12)). Thus, MBP can be considered as a generalization of the BP algorithm [Lopes and Ribeiro, 2003, Lopes and Ribeiro, 2001].

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 (4.18):

$$\Delta m_j = - \frac{\partial E}{\partial m_j} . \quad (4.18)$$

Considering the importance factor of an output neuron  $o = j$  (of the main network), (4.18) becomes (4.19):

$$\Delta m_o = (t_o - y_o) \phi(a_o) , \quad (4.19)$$

Regarding the importance factor of a hidden neuron  $h = j$  (of the main network), (4.18) can be written as (4.20):

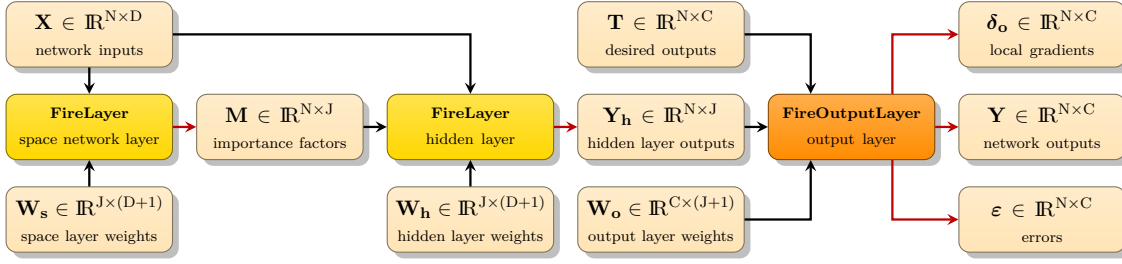
$$\begin{aligned} \Delta m_h &= - \sum_{o=1}^C \frac{\partial E}{\partial a_o} \frac{\partial a_o}{\partial m_h} \\ &= \sum_{o=1}^C \delta_o \frac{\partial}{\partial m_h} \sum_{i=0}^I W_{io} y_i \\ &= \sum_{o=1}^C \delta_o \frac{\partial}{\partial m_h} \sum_{i=0}^I W_{io} m_i \phi(a_i) \end{aligned} \quad (4.20)$$

and finally we obtain (4.21):

$$\Delta m_h = \sum_{o=1}^C \delta_o W_{ho} \phi(a_h) . \quad (4.21)$$

Employing (4.19) and (4.21), 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 [Lopes and Ribeiro, 2003, Lopes and Ribeiro, 2001] and has yielded good results in several applications, such as financial data analysis [Bucur and Florea, 2011], electricity consumption forecasting [Granmo, 2012], analysis of market orientation [Silva et al., 2009] and character recognition [Chacko et al., 2010].



**Figure 4.9:** Model of the kernels executed (in each epoch) to complete the forward phase of an MBP network.

### 4.1.3 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 pages 68 and 69), 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.

#### 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 4.9 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.

Regarding the batch training mode, there are two sources of parallelism: First the outputs of the neurons 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 [Ryoo et al., 2008] (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 4.1 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 the `FireOutputLayer` kernel. However, we noticed that part of the data needed to calculate the local gradients (of the output layer) and the RMSE of the network was already in the SM registries and shared memory (see Figure 4.9 and listing 4.1) 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,  $\boldsymbol{\delta}_o \in \mathbb{R}^{N \times C}$ , the local gradients of the space network neurons,  $\boldsymbol{\delta}_s \in \mathbb{R}^{N \times C}$  (associated with the selective actuation neurons in the output layer) and the (square of the) neurons' errors,  $\boldsymbol{\varepsilon} \in \mathbb{R}^{N \times C}$ , where each element (corresponding to sample  $n$  and output  $o$ ) is given by  $\varepsilon_{no} = (T_{no} - Y_{no})^2$ .

### Robust Learning Phase

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



Listing 4.1: 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();

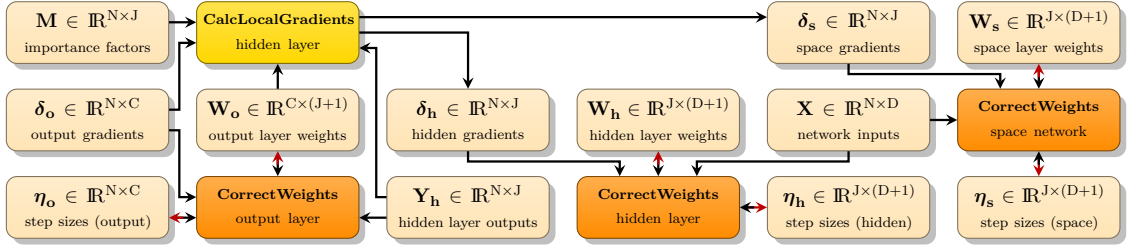
    int es = N_INPUTS_INCL_BIAS;
    for(int s = (es >> 1); es > 1; s = (es >> 1)) {
        int nextNumberElemSum = s;
        if (es & 1) nextNumberElemSum++;
        if (INPUT < s) xw[c] += xw[c + nextNumberElemSum];
        es = nextNumberElemSum;
        __syncthreads();
    }
}

__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;
    }
}

```



**Figure 4.10:** Model of the kernels executed (in each epoch) in the back-propagation phase of an MBP network.

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 are restored, (ii) the step sizes multiplied (reduced) by the robustness factor,  $r$ , and (iii) the momentum memories are set to zero.

### 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,  $\delta_h \in \mathbb{R}^{N \times J}$ . If there are neurons with selective actuation, the corresponding local gradients,  $\delta_s \in \mathbb{R}^{N \times J}$ , of the space network are also calculated. The task of the **CorrectWeights** kernel consists of adjusting the weights,  $W_h \in \mathbb{R}^{J \times (D+1)}$ , and the corresponding step sizes,  $\eta_h \in \mathbb{R}^{J \times (D+1)}$ , 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 4.10 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 considered in Figure 4.9.

#### 4.1.4 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. 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 training. 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 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 4.1.5) and constitutes a working basis on top of which new improvements can be build.

#### 4.1.5 Results and Discussion

##### Experimental Setup

The experimental setup was conducted using the CUDA implementation, described earlier in Section 4.1.3, and the Multiple Back-Propagation software: a highly optimized application, developed in C++, for training NNs [Lopes and Ribeiro, 2009c]. 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/>.

**Algorithm 1** Autonomous Training System.

---

```
1: Input:  $d_{train}$  ▷ Training dataset.
2: Input:  $d_{test}$  ▷ Test dataset.
3: Input:  $nets_{train}$  ▷ Number of networks to train.
4: Input:  $h_{ini}$  ▷ Initial number of hidden neurons.
5:  $fpn_{best} \leftarrow \infty$ 
6:  $nets_{trained} \leftarrow 0$ 
7:  $inc \leftarrow 0$ 
8:  $h \leftarrow h_{ini}$ 
9:  $h_{best} \leftarrow h$ 
10:  $direction \leftarrow down$ 
11: repeat
12:    $network \leftarrow$  new network with  $h$  hidden neurons
13:   Train( $network$ ,  $d_{train}$ )
14:    $nets_{trained} \leftarrow nets_{trained} + 1$ 
15:    $fp \leftarrow$  FalsePositives( $network$ ,  $d_{test}$ )
16:    $fn \leftarrow$  FalseNegatives( $network$ ,  $d_{test}$ )
17:    $fpn \leftarrow fp + fn$ 
18:   Log  $network$ ,  $fp$ ,  $fn$ , RMSE( $network$ ,  $d_{test}$ )
19:   if  $fpn \leq fpn_{best}$  then
20:      $fpn_{best} \leftarrow fpn$ 
21:      $h_{best} \leftarrow h$ 
22:      $inc \leftarrow inc + 1$ 
23:     SaveNetwork( $network$ )
24:   end if
25:   if  $h < h_{best}$  then
26:     if  $direction = down$  then
27:        $direction \leftarrow up$ 
28:       if  $inc > 1$  then  $inc \leftarrow inc - 1$ 
29:     end if
30:   else if  $h > h_{best}$  then
31:     if  $direction = up$  then
32:        $direction \leftarrow down$ 
33:       if  $inc > 1$  then  $inc \leftarrow inc - 1$ 
34:     end if
35:   end if
36:   if  $direction = up$  then
37:      $h \leftarrow h + inc$ 
38:   else if  $h > 1$  then
39:      $h \leftarrow h - inc$ 
40:   else
41:      $direction \leftarrow up$ 
42:   end if
43: until  $nets_{trained} \geq nets_{train}$ 
```

---

**Table 4.1:** Main characteristics of the training datasets used in the MBP experimental setup.

| Dataset (Benchmark)     | Samples ( $N$ ) | Features ( $D$ ) | Classes ( $C$ ) |
|-------------------------|-----------------|------------------|-----------------|
| <i>Sinus cardinalis</i> | 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               |

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 3.1 and 3.2, on page 38, 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 Section 3.4, but we reproduce here, in Table 4.1, the main characteristics of their training datasets. Note that unlike in Table 3.4, which shows the overall number of samples, the number presented here corresponds only to the training samples, which are the ones effectively used to train the networks.

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 [Lopes and Ribeiro, 2003]. Hence, in the latter benchmark, the networks had also 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,  $\eta$ , and the momentum terms,  $\alpha$ , 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.

## 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

**Table 4.2:** Speedups ( $\times$ ) for the *sinus cardinalis* problem.

| <b>Topology</b> | <b>Hidden units (<math>J</math>)</b> | <b>System 1 (8600 GT)</b> | <b>System 2 (GTX 280)</b> |
|-----------------|--------------------------------------|---------------------------|---------------------------|
| BP              | 7                                    | $3.98 \pm 0.19$           | $5.48 \pm 0.21$           |
|                 | 9                                    | $4.66 \pm 0.16$           | $7.15 \pm 0.13$           |
|                 | 11                                   | $5.44 \pm 0.13$           | $8.43 \pm 0.15$           |
| MBP             | 5                                    | $4.37 \pm 0.10$           | $6.08 \pm 0.13$           |
|                 | 7                                    | $5.73 \pm 0.07$           | $7.99 \pm 0.10$           |
|                 | 9                                    | $6.77 \pm 0.09$           | $10.24 \pm 0.12$          |

**Table 4.3:** Speedups ( $\times$ ) for the *two-spirals* problem.

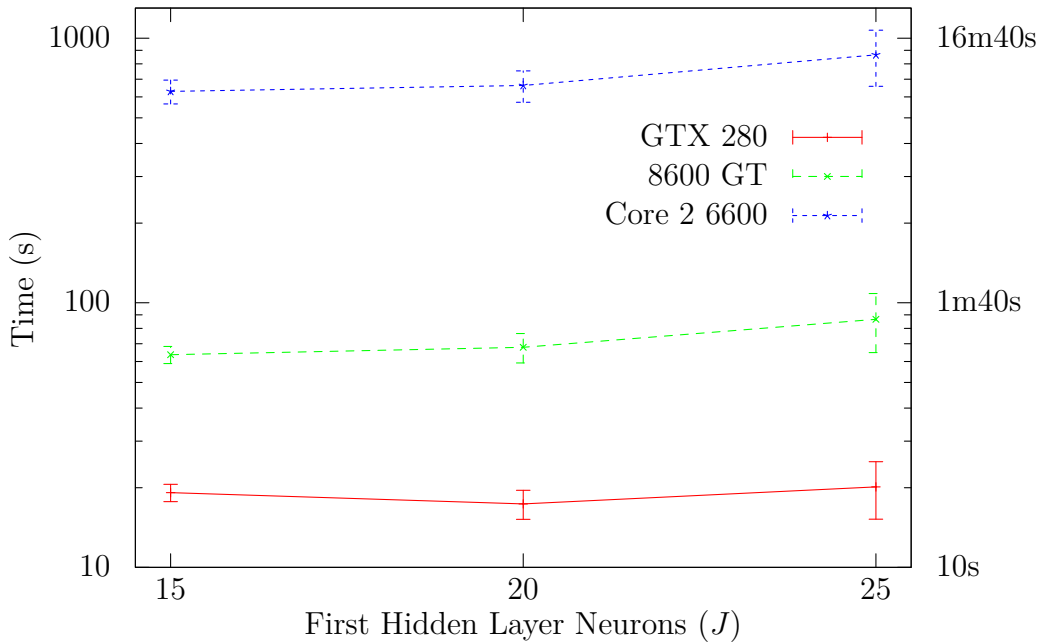
| <b>Topology</b> | <b>Hidden units (<math>J</math>)</b> | <b>System 1 (8600 GT)</b> | <b>System 2 (GTX 280)</b> |
|-----------------|--------------------------------------|---------------------------|---------------------------|
| BP              | 25                                   | $7.68 \pm 1.01$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

and a decrement  $d$  of 0.95 (the robust training technique was not used). Tables 4.2 and 4.3 present the average GPU speedups and the corresponding standard deviations respectively for the *sinus cardinalis* and for *two-spirals* benchmarks.

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 4.11 visually shows the performance improvements of the GPU over the CPU.

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

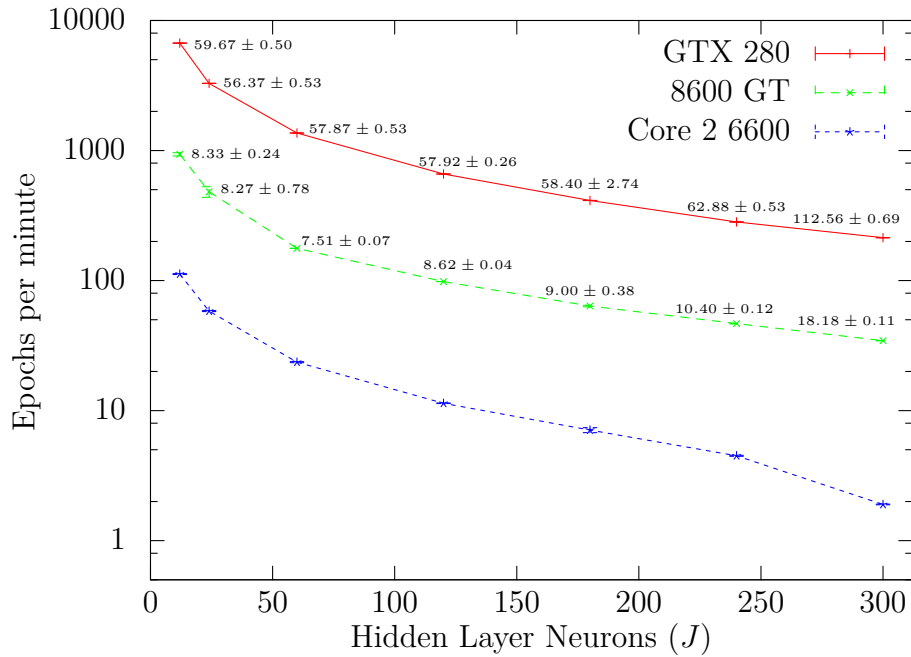


**Figure 4.11:** *Two-spirals* training time (MBP algorithm).

a significant speedup (up to  $10\times$  in this case). Moreover, even when using an old model device (8600 GT), it is possible to obtain significant speedups (almost  $7\times$  in the *sinus cardinalis* and up to  $10\times$  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



**Figure 4.12:** Number of epochs per minute using the BP algorithm for the *forest cover* problem. The GPU speedups are shown near the corresponding lines.

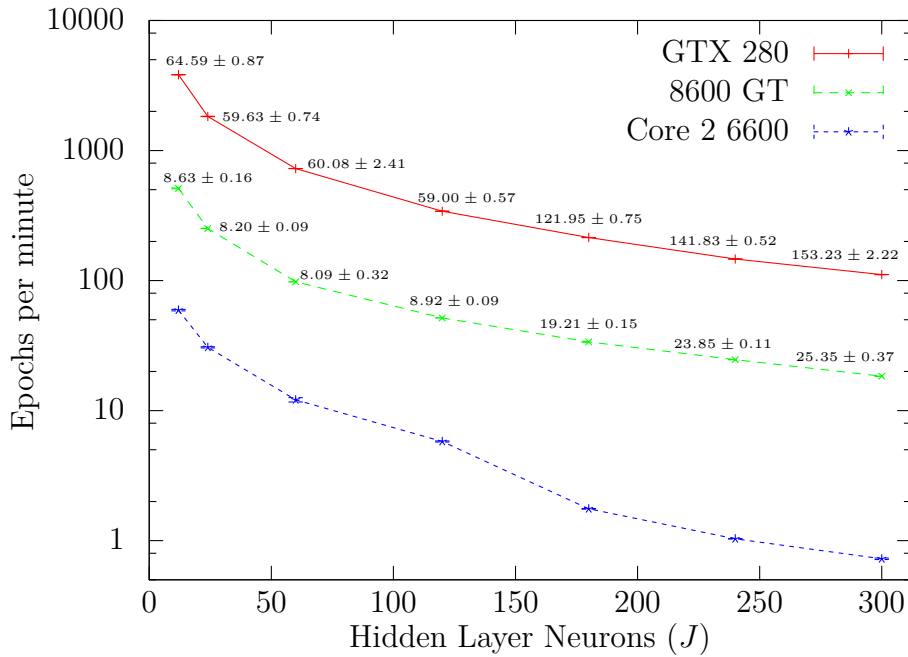
to use the number of epochs trained per minute, instead of the time required to reach a predefined RMSE. Moreover, in the remaining tests, 30 networks for each configuration were trained to establish confidence bounds and ensure statistical significance using standard deviation.

In the *forest cover* benchmark, we selected 7 out of the 14 different hidden neurons configurations, chosen in Blackard and Dean [Blackard and Dean, 1999]. 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 4.12 and 4.13 show the number of epochs trained per minute for the *forest cover* problem, using respectively the BP and MBP algorithms.

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 4.14 and 4.15 present the number of epochs trained per minute for the *poker hand* problem, using respectively the BP and MBP algorithms. The settings





**Figure 4.13:** Number of epochs per minute using the MBP algorithm for the *forest cover* problem. The GPU speedups are shown near the corresponding lines.

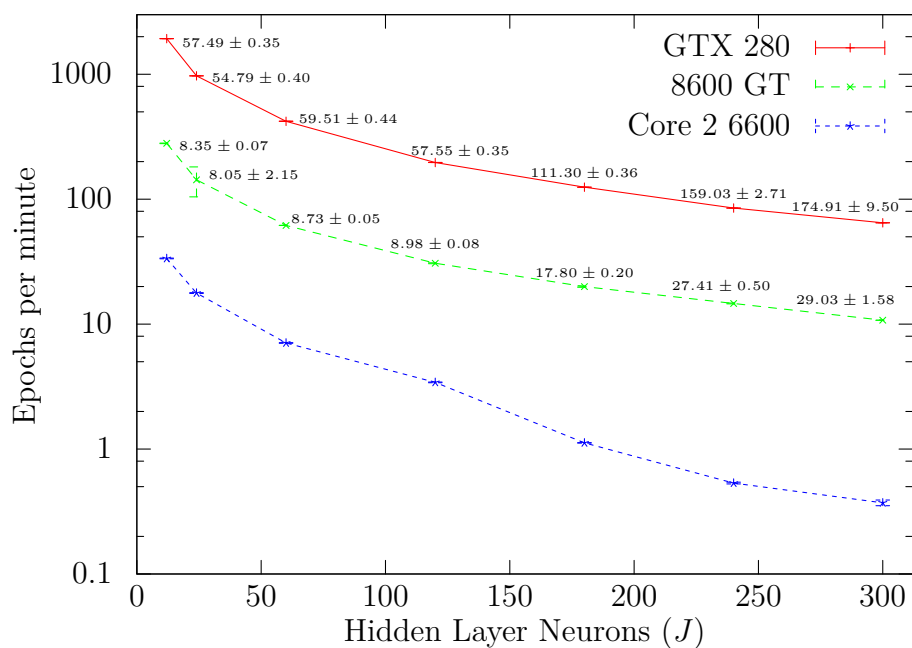
were the same as in the *forest cover* problem. Comparatively to the former, the *poker hand* benchmark has more connections (per layer), which translate into a larger number of CUDA threads and higher speedups (up to 30× on a 8600 GT and 178× on a GTX 280 device). Moreover, training a single epoch for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

### Case study: Ventricular Arrhythmias (VAs)

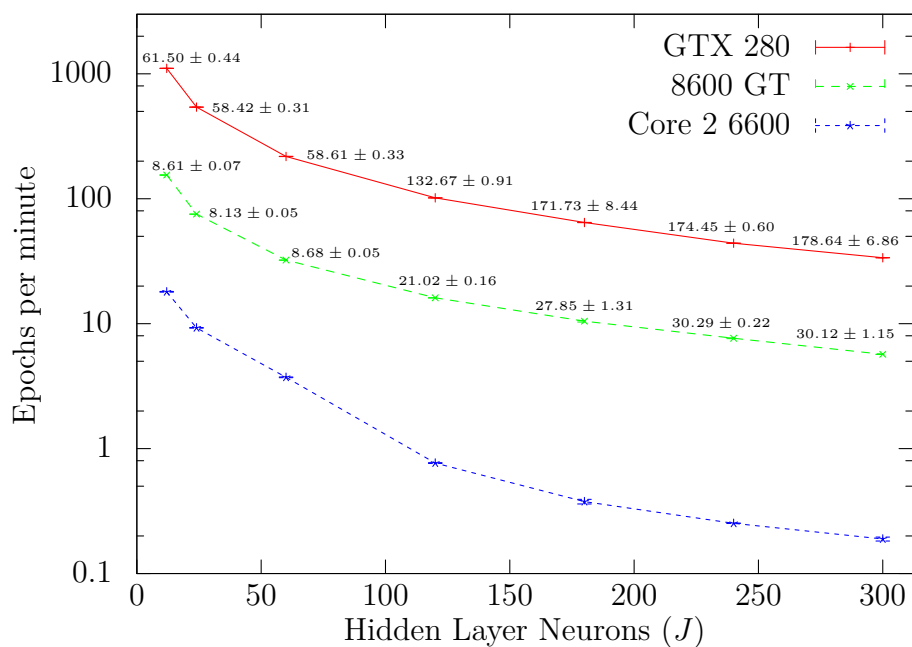
Concerning the real-world, VAs problem (see page 54), Figures 4.16 and 4.17 show the number of epochs trained per minute according to the hardware used respectively for the BP and MBP algorithms.

Figures 4.18 and 4.19 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.

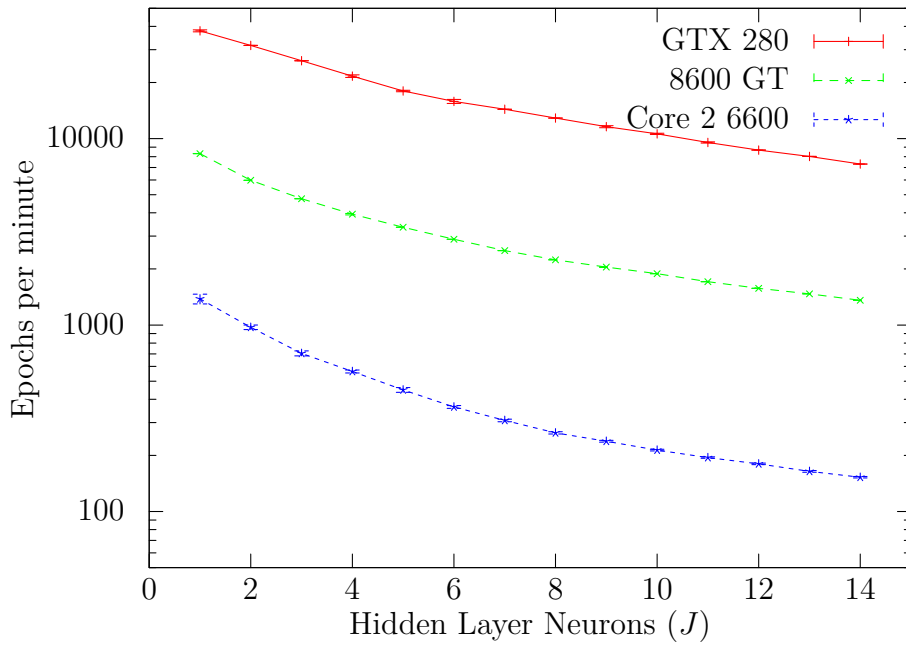
Since currently our implementation does not support using a validation set, preliminary tests were conducted in order to determine when to stop training. Subsequently, using the information collected, we have trained several networks, during one million epochs, varying the number of hidden neurons. It is worth



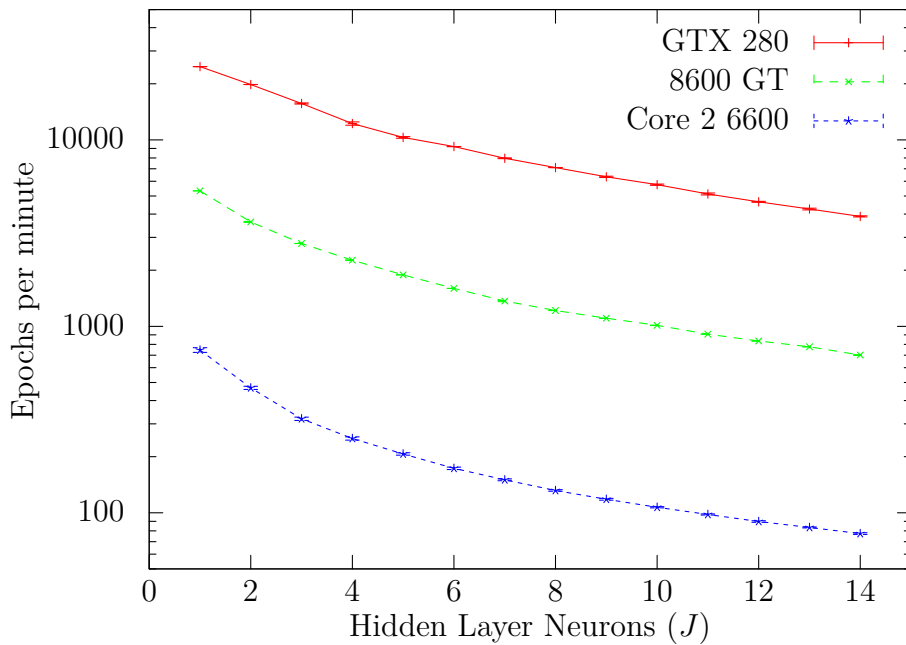
**Figure 4.14:** Number of epochs per minute using the BP algorithm for the *poker* problem. The GPU speedups are shown near the corresponding lines.



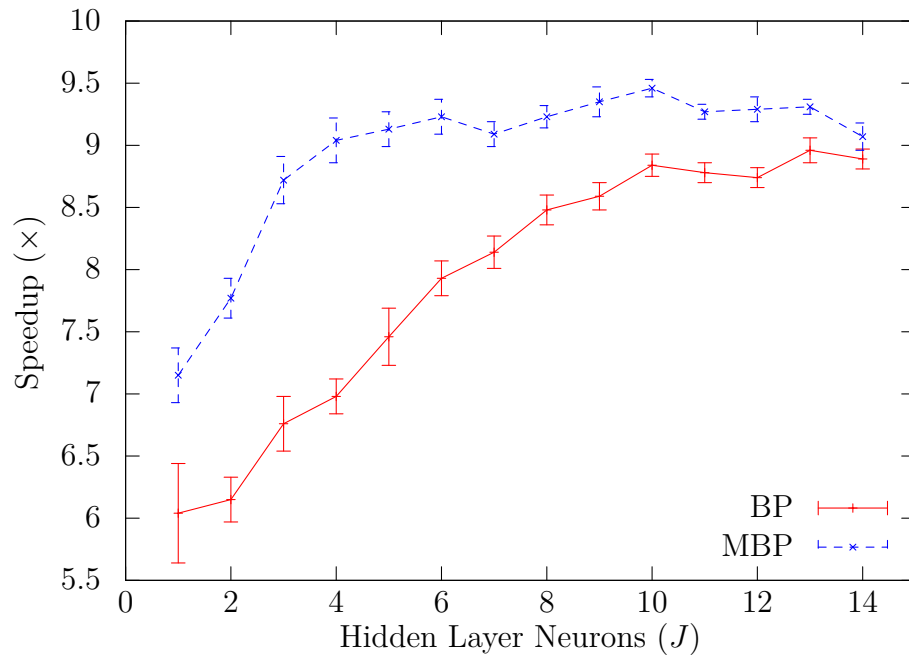
**Figure 4.15:** Number of epochs per minute using the MBP algorithm for the *poker* problem. The GPU speedups are shown near the corresponding lines.



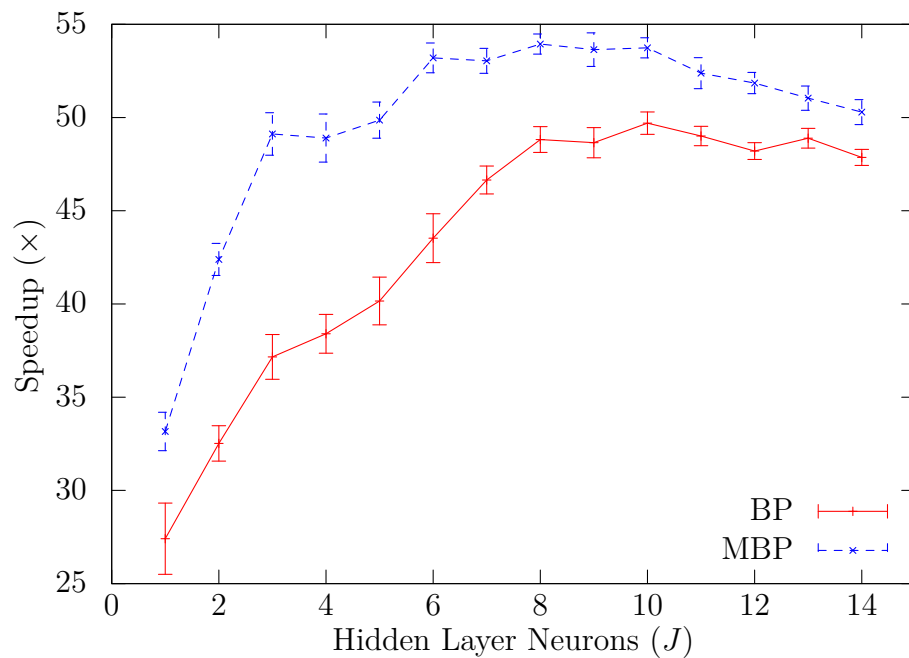
**Figure 4.16:** Number of epochs per minute for the Ventricular Arrhythmias case study, using the BP algorithm.



**Figure 4.17:** Number of epochs per minute for the Ventricular Arrhythmias case study, using the MBP algorithm.



**Figure 4.18:** Speedup ( $\times$ ) obtained for the Ventricular Arrhythmias case study, using an 8600 GT.



**Figure 4.19:** Speedup ( $\times$ ) obtained for the Ventricular Arrhythmias case study, using a GTX 280.

**Table 4.4:** Performance results (%) for the ventricular arrhythmias problem.

| Metrics     | BP ( $J = 14$ ) |       |            | MBP ( $J = 13$ ) |       |            |
|-------------|-----------------|-------|------------|------------------|-------|------------|
|             | 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      |

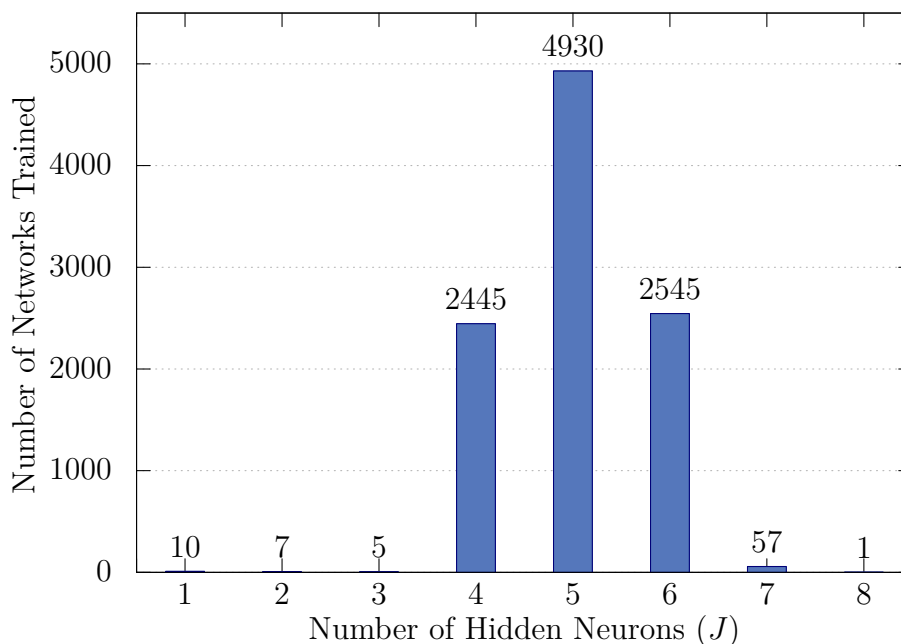
mentioning that during the preliminary tests a few NNs were trained up to 3 million epochs, requiring almost 9 hours of training with a GTX 280 (computer system 2). We estimate that the CPU would require almost three weeks to train a single network (of these) [Lopes and Ribeiro, 2009a].

Table 4.4 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), 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. [Ribeiro et al., 2007] and in Marques [Marques, 2007], would not be obtained without the speedups provided by the GPU [Lopes and Ribeiro, 2009a].

## 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 4.20 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 [Lopes and Ribeiro, 2001]. Additionally, it is worth mentioning that the best network found in Lopes and Ribeiro [Lopes and Ribeiro, 2001] was also an MBP network with 5 hidden neurons.



**Figure 4.20:** Networks trained, according to the number of 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 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 crucial parameters in an NN model. Basically, 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 extreme solutions, both leading to poor generalization cases, can be obtained: (*i*) a solution whereby the network model has 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, reducing the number of free parameters results in more rigid models having 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 and results in models that present the best predictive capabilities [Bishop, 2006]. 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 5.1.5 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.

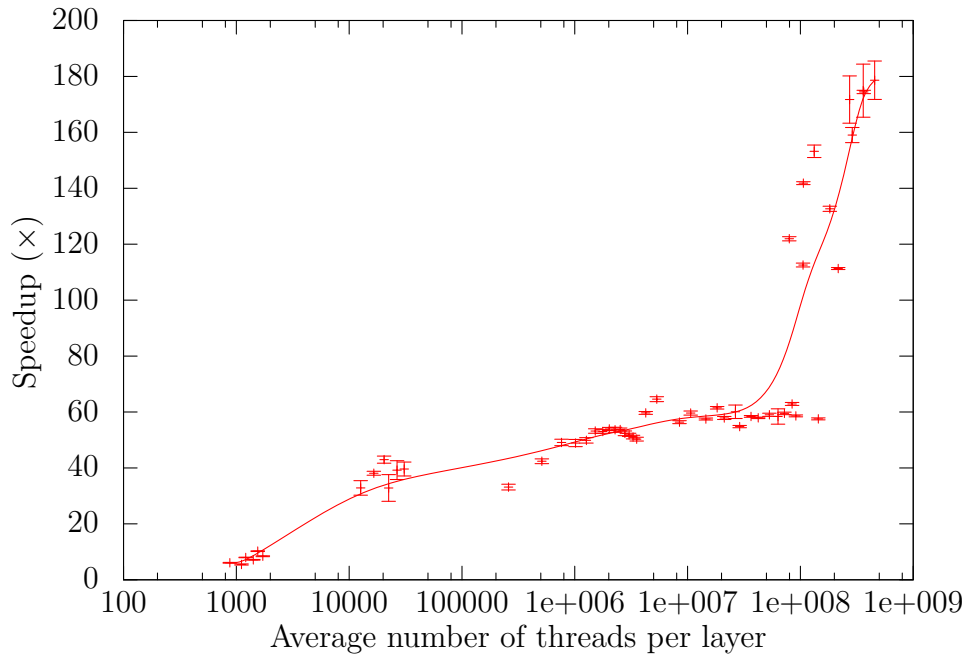
## Discussion

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 4.21 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.

## 4.2 Neural Selective Input Model (NSIM)

Incomplete data is an unavoidable problem for most real-world databases, which often contain missing data [Kotsiantis et al., 2006b, Karhunen, 2011]. In particular, in domains such as gene expression microarray experiments or clinical medicine, databases routinely miss pieces of information [Tuikkala et al., 2008, Markey et al., 2006]. Missing Values (MVs) can exist either by design (e.g. a survey questionnaire



**Figure 4.21:** Speedups ( $\times$ ) versus processing threads.

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 [García-Laencina et al., 2010, Bramer, 2007, Nelwamondo et al., 2007].

Independently of the causes associated to the existence of MVs, the fact is that most scientific data procedures are not designed to handle them [Schafer and Graham, 2002]. 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 [García-Laencina et al., 2010].

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 [Lopes and Ribeiro, 2011f]. 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 [Tuikkala et al., 2008].

### 4.2.1 Missing Data Mechanisms

The presence of MVs in data observations is one of the most frequent problems that must be faced when building ML systems [López-Molina et al., 2008]. 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:

$$K_{ij} = \begin{cases} 1 & \text{if } X_{ij} \text{ is observed} \\ 0 & \text{if } X_{ij} \text{ is missing} \end{cases} . \quad (4.22)$$

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,  $\mathbf{X}_{\text{obs}}$ , containing the samples for which all the variables (features) values are known, and into the unknown input matrix,  $\mathbf{X}_{\text{miss}}$  containing the samples that have variables with MVs:

$$\mathbf{X} \equiv \begin{bmatrix} \mathbf{X}_{\text{obs}} \\ \mathbf{X}_{\text{miss}} \end{bmatrix} , \quad (4.23)$$

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

$$p(\mathbf{K} \mid \mathbf{X}, \xi) = p(\mathbf{K} \mid \mathbf{X}_{\text{obs}}, \mathbf{X}_{\text{miss}}, \xi) , \quad (4.24)$$

where  $\xi$  denotes the unknown parameters which define the missing data mechanism [García-Laencina et al., 2010, Mockus, 2008].

Little and Rubin [Little and Rubin, 2002] define three types of missing data mechanisms according to their causes: Missing At Random (MAR), Missing Completely At Random (MCAR) and Not Missing At Random (NMAR).

#### 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 [García-Laencina et al., 2010]. In such cases we can define the conditional distribution for the missing data as (4.25):

$$p(\mathbf{K} \mid \mathbf{X}_{\text{obs}}, \mathbf{X}_{\text{miss}}, \xi) = p(\mathbf{K} \mid \mathbf{X}_{\text{obs}}, \xi) . \quad (4.25)$$

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 [Mockus, 2008]; a sensor occasionally fails due to power outages, preventing the data acquisition process from taking place [García-Laencina et al., 2010]. 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 [Mockus, 2008]. Similarly, the power outages in the second case explain why the sensor data is missing.

### 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 (4.26):

$$p(\mathbf{K} \mid \mathbf{X}_{\text{obs}}, \mathbf{X}_{\text{miss}}, \xi) = p(\mathbf{K} \mid \xi) . \quad (4.26)$$

Examples of this mechanism include the following [García-Laencina et al., 2010]: a biological sample is accidentally contaminated by the researcher collecting the data; a page from a questionnaire is unintentionally lost.

### Not Missing At Random (NMAR)

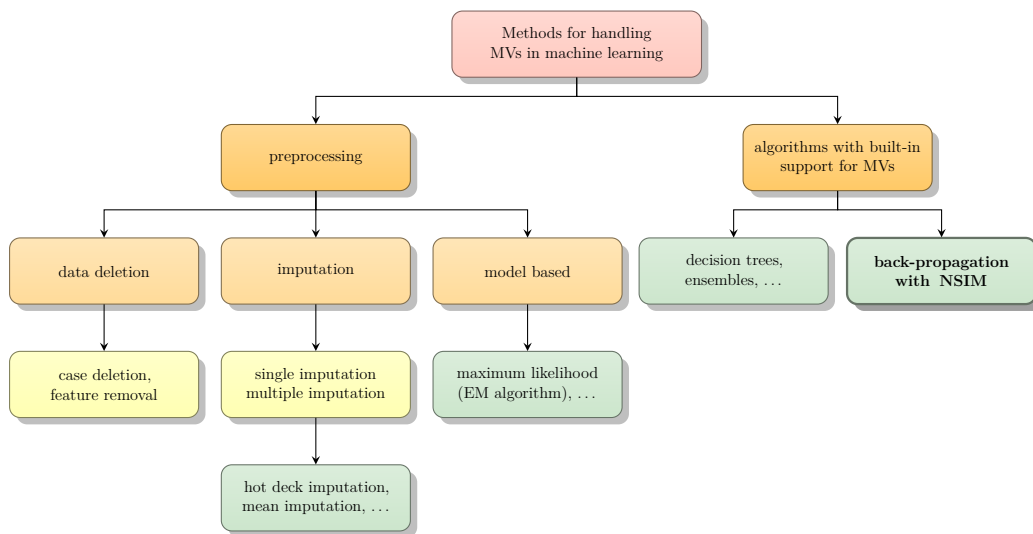
The alternative for data MAR or MCAR is to consider that the data is NMAR. This is the case when the pattern of data *missingness* depends on the missing variables themselves. A typical example of data NMAR is in the case of a personal survey involving private questions, whose nature will most likely 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 [Mockus, 2008].

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 [García-Laencina et al., 2010].

## 4.2.2 Methods for Handling MVs in Machine Learning

According to Laencina et al. [García-Laencina et al., 2010], 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 4.22 presents an overview of ML procedures to handle MVs.

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 [García-Laencina et al., 2010, Tuikkala et al., 2008, López-Molina et al., 2008, Mockus, 2008, Lim and Zainuddin, 2008, Ayuyev et al., 2009, Kotsiantis et al., 2006b]. A review of the methods and techniques to deal with this problem,



**Figure 4.22:** Overview of the types of techniques for handling MVs in ML.

including a comparison of some well-known approaches, can be found in Laencina et al. [García-Laencina et al., 2010].

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 [Bramer, 2007]. However, it is not applicable when the MVs cover a large fraction of the instances, or when their presence in essential attributes is large [Ayuyev et al., 2009, Bramer, 2007]. 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 [López-Molina et al., 2008]. 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 [Bramer, 2007]: 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 [López-Molina et al., 2008]. 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 [Bramer, 2007]. 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) [García-Laencina et al., 2010, Tuikkala et al., 2008, Lim and Zainuddin, 2008, Little and Rubin, 2002]. 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 [Serpen, 2005, Alavala, 2008]. 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 [Alavala, 2008, Wang, 2005]. 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 [López-Molina et al., 2008]. Furthermore, statistically, the variability or correlation estimations can be strongly biased [López-Molina et al., 2008].

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 [López-Molina et al., 2008]. 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. However, 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 [Tuikkala et al., 2008].

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 [Lopes and Ribeiro, 2012a].

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 [García-Laencina et al., 2010]. For example, in the real-world bankruptcy problem, presented later in Section 4.2.5, distressed companies tend to omit much more financial information than healthy ones [Lopes and Ribeiro, 2011f]. 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 [Lopes and Ribeiro, 2011f, Lopes and Ribeiro, 2012a].

Algorithms with built-in support for handling MVs offer numerous advantages over (more) conventional ones: (*i*) the amount of time spent in the preprocessing phase is reduced; (*ii*) 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); (*iii*) noise and outliers are not inadvertently injected in the data and the uncertainty associated to the missing data is preserved; (*iv*) 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 [Lopes and Ribeiro, 2012a].

### 4.2.3 NSIM Proposed Approach

The building blocks of the proposed NSIM are the selective actuation neurons, described earlier in Section 4.1.2 (see page 70).

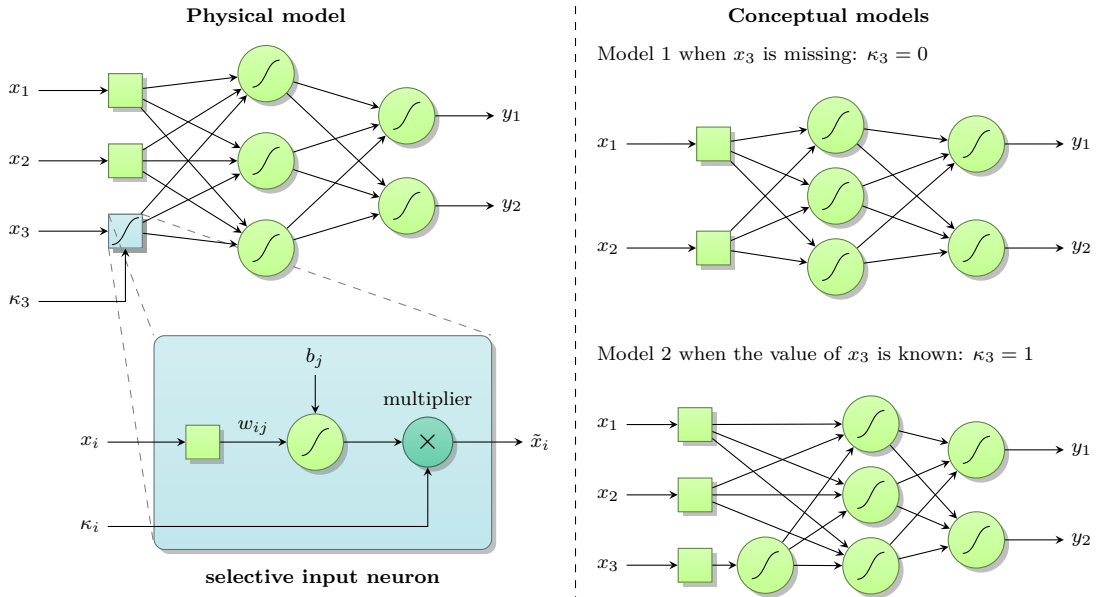
Let us consider that each row of the response matrix,  $\mathbf{K}$ , contains the response indicator vector of a specific sample,  $\boldsymbol{\kappa}_i \in \{0, 1\}^D$ . Furthermore, to simplify the notation, let  $\boldsymbol{\kappa} = [\kappa_1, \kappa_2, \dots, \kappa_D]$  denote the response indicator vector of a generic sample where  $\kappa_i$  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  $\kappa_i$  as shown in (4.27):

$$\tilde{x}_i = f(x_i, \kappa_i) . \quad (4.27)$$

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  $\kappa_i$ . Consequently, (4.27) can be replaced with (4.28), by taking advantage of (4.15):

$$\tilde{x}_i = \kappa_i \phi(W_{ij}x_i + b_j) . \quad (4.28)$$

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  $\kappa_i$  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 4.23 shows the physical model (NSIM) of a network containing a selective input and the two conceptual models inherent to



**Figure 4.23:** Physical and conceptual models of a network with a selective input ( $j = 3$ ).

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) [Lopes and Ribeiro, 2012a].

As we said before, an NN acts as an adaptive non-linear mapping function,  $f : \mathbb{R}^D \rightarrow \mathbb{R}^C$ . Consequently, the goal of the training procedure consists of adjusting the free parameters (weights) of the network ( $W_1, W_2, \dots, W_P$ ), 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,  $f_1(W_1, W_2, \dots, W_s)$  and  $f_2(W_1, W_2, \dots, W_s, W_{s+1}, \dots, W_P)$ , that share  $s$  parameters. For the network presented in Figure 4.23,  $s$  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 [Lopes and Ribeiro, 2010d].

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 (4.8), (4.16) and (4.17)). Thus, only the relevant (conceptual) models will be adjusted [Lopes and Ribeiro, 2010d].

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 [Lopes and Ribeiro, 2012a].

#### 4.2.4 GPU Parallel Implementation

The GPU implementation of the NSIM extends the BP and MBP CUDA parallel implementation previously presented in Section 4.1.3 (see page 75). 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 4.2, 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 79).

#### 4.2.5 Results and Discussion

##### 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 4.5 presents the main characteristics of the databases and an overview of the proportion

Listing 4.2: 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;
}

```

---

**Table 4.5:** 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.

| Database        | $N$  | $D$ | $C$ | Proportion<br>of MVs (%) | Features<br>with MVs | MVs per feature<br>(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                              |

and distribution of the MVs in each database, after preprocessing. Additional details on the datasets can be found in Section 3.4.



**Table 4.6:** Macro-average F-Measure performance (%) according to the methods used to handle the MVs and the algorithms used to train the NNs.

| Database        | NSIM                 |                      | Single Imputation    |                      | Multiple Imputation  |                      |
|-----------------|----------------------|----------------------|----------------------|----------------------|----------------------|----------------------|
|                 | BP                   | MBP                  | BP                   | MBP                  | BP                   | MBP                  |
| Annealing       | 97.13 ± 02.69        | <b>97.77 ± 02.55</b> | 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        | <b>61.07 ± 14.72</b> |
| Breast cancer   | 95.05 ± 01.59        | 95.42 ± 01.69        | 95.38 ± 01.21        | <b>95.53 ± 01.67</b> | 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        | <b>94.83 ± 01.79</b> | 94.70 ± 01.46        |
| hepatitis       | 70.10 ± 06.23        | 73.55 ± 05.99        | 72.63 ± 07.96        | 72.20 ± 07.53        | <b>75.89 ± 10.35</b> | 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        | <b>87.37 ± 02.30</b> |
| Japanese credit | <b>81.98 ± 02.45</b> | 81.50 ± 02.81        | 81.43 ± 02.53        | 81.07 ± 02.25        | 81.27 ± 01.93        | 81.59 ± 02.45        |
| Mammographic    | <b>81.62 ± 01.74</b> | 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        | <b>99.98 ± 00.02</b> | <b>99.98 ± 00.01</b> | 99.97 ± 00.02        | <b>99.98 ± 00.01</b> |
| Soybean         | 93.43 ± 00.68        | <b>94.34 ± 00.94</b> | 91.63 ± 01.26        | 92.87 ± 00.85        | 92.54 ± 00.68        | 93.48 ± 00.66        |

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 [Ayuyev et al., 2009]. For single imputation the version 3.7.5 of the Weka software package was used [Hall et al., 2009]. For multiple imputation, the NORM (Multiple imputation of incomplete multivariate data under a normal model) software was used [Schafer, 1999]. 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 4.5, 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 [Nelwamondo et al., 2007].

## Benchmark Results

Table 4.6 presents the macro-average F-Measure performance (%) according to the algorithms used to train the NNs and the methods chosen for handling the MVs.

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. These results 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 [Ayuyev et al., 2009].

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 in their NN systems. Moreover, the time and memory constraints necessary for the imputation process to take place would in many cases render such systems useless.

### **Case study: Financial Distress Prediction**

This study involves the bankruptcy problem described earlier in Section 3.5 (see page 53). 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

**Table 4.7:** Results of the NSIM for the bankruptcy problem.

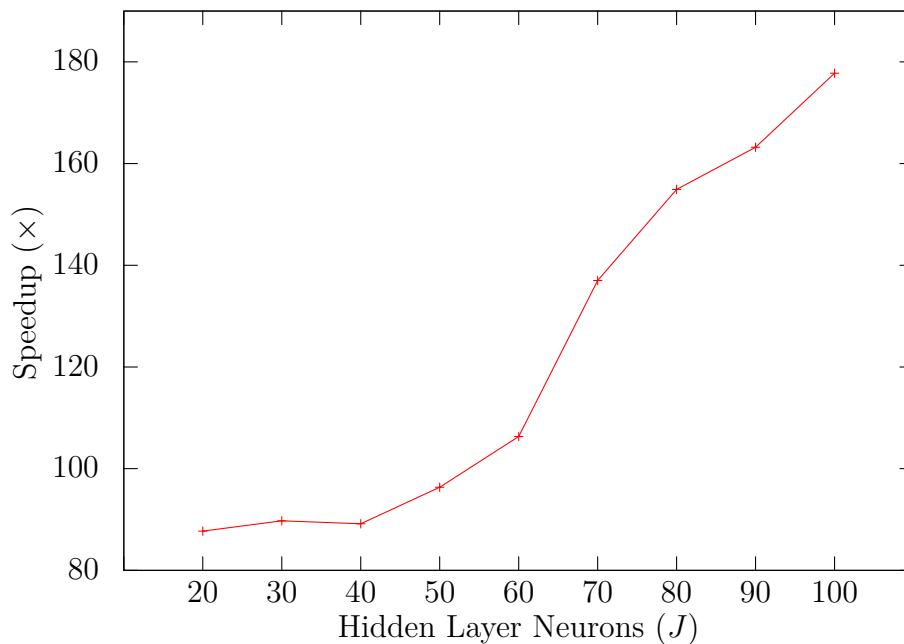
| <b>Metric</b> | <b>Results (%)</b> |
|---------------|--------------------|
| Accuracy      | $95.70 \pm 1.42$   |
| Sensitivity   | $95.60 \pm 1.61$   |
| Specificity   | $95.80 \pm 1.83$   |
| Precision     | $95.82 \pm 1.77$   |
| Recall        | $95.60 \pm 1.61$   |
| F-measure     | $95.70 \pm 1.35$   |

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). Thus, 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 [Vieira et al., 2009, Ribeiro et al., 2010], 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 4.7 presents the results of the NSIM, with the MBP algorithm, using a 10-fold cross-validation. These excel by far the results previously obtained [Vieira et al., 2009, Ribeiro et al., 2010] when imputation techniques were used and demonstrated the validity and usefulness of the NSIM in a real-world setting.

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



**Figure 4.24:** GPU speedups obtained for the bankruptcy problem.

4.24 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.

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 when relevant information is missing.

### 4.3 Incremental Hypersphere Classifier (IHC)

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 [Gama et al., 2009]. 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) [Zhou, 2003] and predominantly focus on one-shot data analysis from homogeneous and stationary data [Gama et al., 2009]. 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 [García-Pedrajas et al., 2010].

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 during the whole learning process [Jain et al., 2006]. 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 [Lopes and Ribeiro, 2011d, Lopes and Ribeiro, 2011e].

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 [Reinartz, 2002].

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 [Olvera-López et al., 2010, Kotsiantis et al., 2006a]. 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. [Olvera-López et al., 2010].

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 [Olvera-López et al., 2010].

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 [Jain et al., 2006].

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 [Pappa and Freitas, 2010, Bibi and Stamelos, 2006].

### 4.3.1 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  $\rho_i$ , given by (4.29):

$$\rho_i = \frac{\min(\|\mathbf{x}_i - \mathbf{x}_j\|)}{2}, \text{ for all } j \text{ where } y_j \neq y_i \quad (4.29)$$

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

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 ( $\rho_i$  and  $\rho_j$ ) 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  $\rho_i$  and  $\rho_j$  can have without overlapping their hypersphere's. Figure 4.25(a) shows the regions of influence for a chosen toy problem.

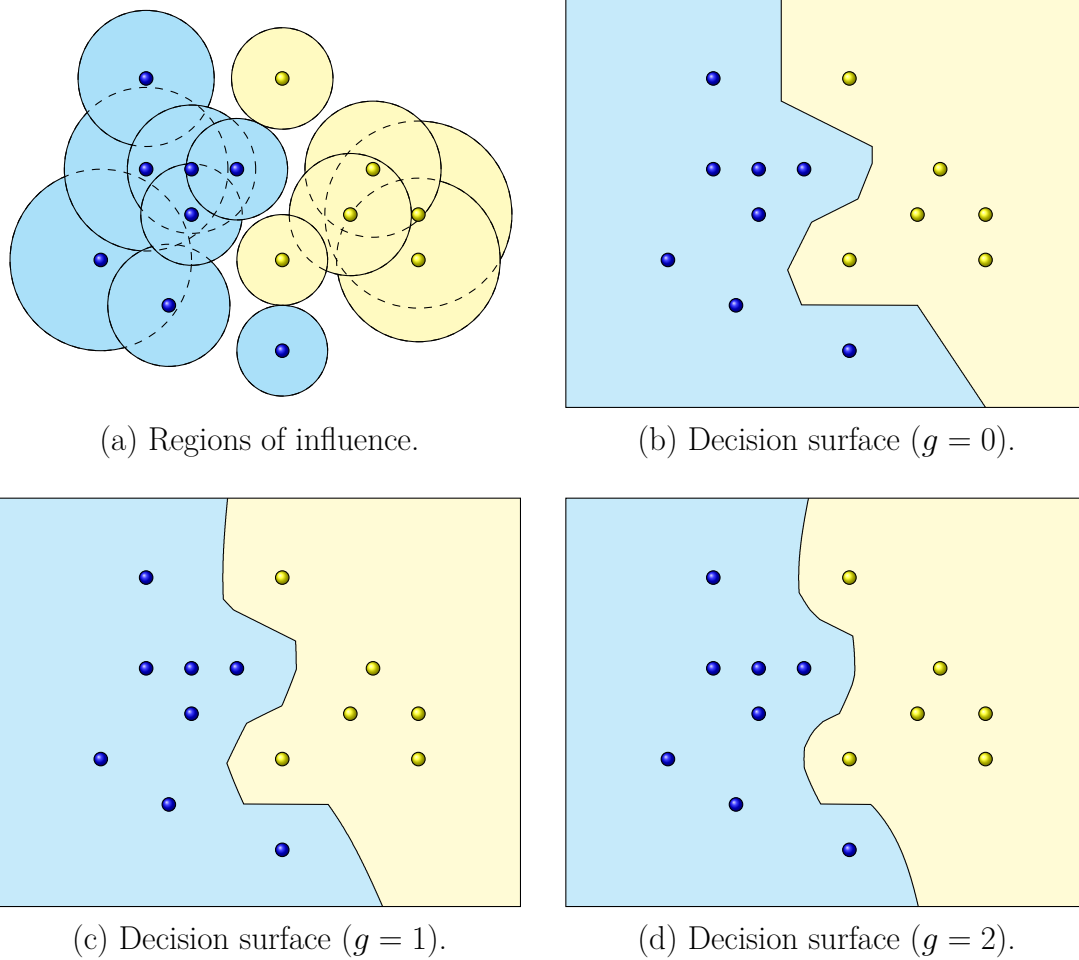
New data points are classified according to the class of the nearest region of influence (not the nearest sample). Let  $\mathbf{x}_k$  represent an input vector whose class  $y_k$  is unknown. Then, sample  $k$  belongs to class  $y_i$  ( $y_k = y_i$ ) provided that:

$$\|\mathbf{x}_i - \mathbf{x}_k\| - g a_i \rho_i \leq \|\mathbf{x}_j - \mathbf{x}_k\| - g a_j \rho_j, \text{ for all } j \neq i \quad (4.30)$$

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.

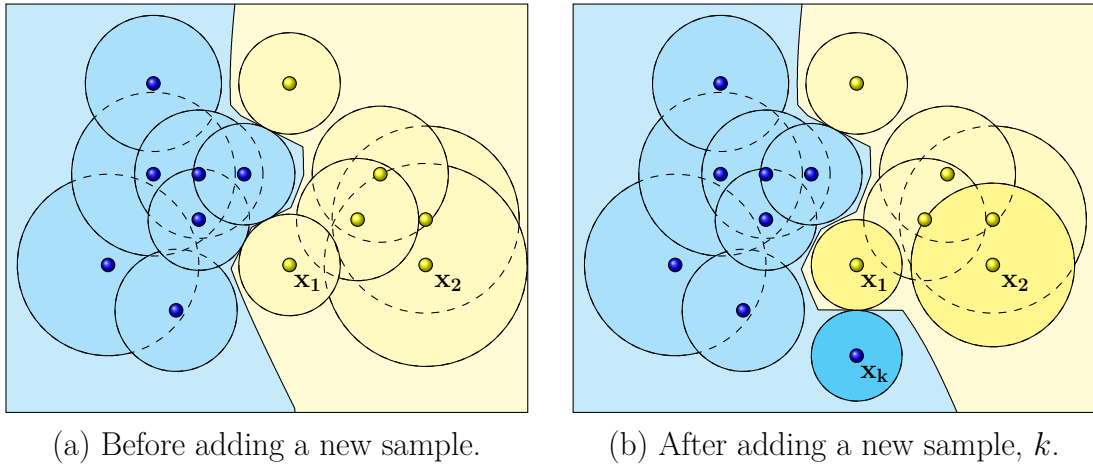
Figures 4.25(b), 4.25(c) and 4.25(d) show the decision surface generated by the IHC algorithm for a toy problem, considering different gravity,  $g$ , values. Note that for  $g = 0$  the decision rule of the IHC is exactly the same as the one of the 1-nearest neighbor (see (4.30)). A detailed description of the  $k$ -nn can be found in Clarke et al. [Clarke et al., 2009]. It is interesting to point out that (for  $g > 0$ ) the



**Figure 4.25:** Application of the IHC algorithm to a toy problem.

IHC algorithm generates smoother decision surfaces (see Figures 4.25(b), 4.25(c) and 4.25(d)).

Note that the farthest from the decision border an hypersphere is, the larger its radius will be (see Figure 4.25(a)). This provides a simple method for determining the relevance of a given sample: samples with smaller radius ( $\rho$ ) 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.



**Figure 4.26:** Regions of influence and decision surfaces generated by IHC for a toy problem ( $g = 1$ ).

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 (4.30), 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 [Wilson and Martinez, 2000] or alternatively in Aha et al. [Aha et al., 1991].

Accordingly, as in the IB3 algorithm, confidence intervals are determined both for the instance accuracy, not including its own classification, unlike in (4.30), 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 [Wilson and Martinez, 2000, Aha et al., 1991].

A major advantage of the IHC algorithm relies on the possibility of building models incrementally on a sample by sample basis. Figure 4.26 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  $\mathbf{x}_2$ ).

Algorithm 2 describes the main steps required to incorporate a new sample,  $k$ , on the IHC model. 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  $O(2Dn)$ .



---

**Algorithm 2** Incremental Hypersphere Classifier (IHC) algorithm.

---

```

1: Input:  $\mathbf{x}_k$                                 ▷ Input vector of the new sample  $k$ .
2: Input:  $y_k$                                     ▷ Class of the new sample  $k$ .
3:  $\rho_k \leftarrow \infty$                             ▷ Radius of sample  $k$ 
4:  $d \leftarrow \infty$                                 ▷ Distance to the nearest sample (using  $\|\mathbf{x}_i - \mathbf{x}_k\| - g a_i \rho_i$ )
5:  $n \leftarrow \text{null}$                                 ▷ Nearest sample (using  $\|\mathbf{x}_i - \mathbf{x}_k\| - g a_i \rho_i$ )
6:  $tp_k \leftarrow 1$                                 ▷ True positives (classified by sample  $k$ )
7:  $fp_k \leftarrow 0$                                 ▷ False positives ( $a_k = \frac{tp_k}{tp_k + fp_k}$ )
8: for  $class \leftarrow 1, \dots, C$  do
9:     for all sample  $i \in \text{memory}[class]$  do
10:        if  $\|\mathbf{x}_i - \mathbf{x}_k\| - g a_i \rho_i < d$  then
11:             $d \leftarrow \|\mathbf{x}_i - \mathbf{x}_k\| - g a_i \rho_i$ 
12:             $n \leftarrow i$ 
13:        end if
14:        if  $class \neq y_k$  then
15:            if  $\frac{\|\mathbf{x}_i - \mathbf{x}_k\|}{2} < \rho_k$  then  $\rho_k \leftarrow \frac{\|\mathbf{x}_i - \mathbf{x}_k\|}{2}$ 
16:            if  $\frac{\|\mathbf{x}_i - \mathbf{x}_k\|}{2} < \rho_k$  then  $\rho_i \leftarrow \frac{\|\mathbf{x}_i - \mathbf{x}_k\|}{2}$ 
17:        end if
18:    end for
19: end for
20: if  $\text{memory}[y_k]$  is full and  $n \neq \text{null}$  then
21:    if  $\rho_n > \rho_k$  then
22:        Remove sample  $n$  from  $\text{memory}[y_k]$ 
23:         $d \leftarrow \infty$ 
24:         $j \leftarrow \text{null}$                                 ▷ Nearest sample of sample  $n$ 
25:        for  $class \leftarrow 1, \dots, C$  do
26:            for all sample  $i \in \text{memory}[class]$  do
27:                if  $\|\mathbf{x}_i - \mathbf{x}_n\| - g a_i \rho_i < d$  then
28:                     $d \leftarrow \|\mathbf{x}_i - \mathbf{x}_n\| - g a_i \rho_i$ 
29:                     $j \leftarrow i$ 
30:                end if
31:            end for
32:        end for
33:        if  $j \neq \text{null}$  then
34:            if  $y_n = y_j$  then  $tp_j \leftarrow tp_j + 1$  else  $fp_j \leftarrow fp_j + 1$ 
35:        end if
36:    else
37:        if  $y_n = y_k$  then  $tp_n \leftarrow tp_n + 1$  else  $fp_n \leftarrow fp_n + 1$ 
38:    end if
39: end if
40: if  $\text{memory}[y_k]$  is not full then Add sample  $k$  to  $\text{memory}[y_k]$ 

```

---

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.

### 4.3.2 Results and Discussion

#### 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 [Tahir and Smith, 2010]. 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 (*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, a real-world case study (Protein membership prediction), in which we combine the IHC and SVM algorithms, is addressed. Note that for this experiment, we specifically chose 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.

The SVM [Vapnik, 1995] is an algorithm that obtains sparse solutions in the sense that the models' decision surface is defined by a subset of the training data points (support vectors). The training data is projected into a higher-dimensional space by means of a kernel where the optimal hyperplane obtained linearly separates the classes with the largest margin. This methodology has been highly successful in solving both classification and regression problems, building discriminative models that combine high-accuracy with good generalization properties [Bishop, 2006]. In particular, SVMs have been widely applied in classification of biological data including sub-sequence cellular prediction and protein sequence classification [Hua and Sun, 2001, Rätsch et al., 2006, She et al., 2003]. The SVM models are implemented successfully in the well-known Library for Support Vector Machines (LIBSVM) software [Chang and Lin, 2011].

All the experiments were performed using computer system 2 (see Table 3.1, page 38).

## Benchmark Results

Table 4.8 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.

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

**Table 4.8:** 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 $\pm$ 0.41                   | <b>96.07 <math>\pm</math> 0.30</b> | <b>96.45 <math>\pm</math> 0.36</b> |
| Ecoli                | 336  | 7   | 8   | 66.04 $\pm$ 0.82                   | <b>67.51 <math>\pm</math> 0.72</b> | <b>68.03 <math>\pm</math> 0.78</b> |
| German credit data   | 1000 | 59  | 2   | <b>64.38 <math>\pm</math> 0.96</b> | 63.98 $\pm$ 0.95                   | 63.55 $\pm$ 0.95                   |
| Glass identification | 214  | 9   | 6   | 68.77 $\pm$ 1.63                   | <b>70.30 <math>\pm</math> 2.20</b> | <b>69.81 <math>\pm</math> 2.23</b> |
| Haberman’s survival  | 306  | 3   | 2   | 55.53 $\pm$ 2.04                   | 55.26 $\pm$ 2.35                   | <b>56.36 <math>\pm</math> 1.92</b> |
| Heart - Statlog      | 270  | 20  | 2   | 75.30 $\pm$ 1.60                   | <b>75.92 <math>\pm</math> 1.28</b> | <b>76.19 <math>\pm</math> 1.27</b> |
| Ionosphere           | 351  | 34  | 2   | 85.90 $\pm$ 0.69                   | <b>90.98 <math>\pm</math> 0.54</b> | <b>92.55 <math>\pm</math> 0.47</b> |
| Iris                 | 150  | 4   | 3   | 95.70 $\pm$ 0.69                   | <b>95.71 <math>\pm</math> 0.61</b> | <b>96.04 <math>\pm</math> 0.64</b> |
| Pima diabetes        | 768  | 8   | 2   | 66.95 $\pm$ 1.06                   | <b>68.41 <math>\pm</math> 1.00</b> | <b>70.09 <math>\pm</math> 0.97</b> |
| Sonar                | 208  | 60  | 2   | 85.60 $\pm$ 1.76                   | <b>85.63 <math>\pm</math> 1.79</b> | <b>87.03 <math>\pm</math> 1.50</b> |
| Tic-Tac-Toe          | 958  | 9   | 2   | 49.47 $\pm$ 0.47                   | <b>73.43 <math>\pm</math> 0.54</b> | <b>81.21 <math>\pm</math> 0.83</b> |
| Vehicle              | 946  | 18  | 4   | 69.35 $\pm$ 0.76                   | <b>69.46 <math>\pm</math> 0.71</b> | 68.78 $\pm$ 0.93                   |
| Wine                 | 178  | 13  | 3   | 95.90 $\pm$ 0.51                   | <b>96.80 <math>\pm</math> 0.44</b> | <b>96.93 <math>\pm</math> 0.64</b> |
| Yeast                | 1484 | 8   | 10  | 56.32 $\pm$ 1.04                   | <b>57.73 <math>\pm</math> 1.12</b> | <b>58.75 <math>\pm</math> 0.86</b> |

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 4.9 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 [García-Pedrajas et al., 2010], presenting low storage requirements and high accuracy results [Wilson and Martinez, 2000]. 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*

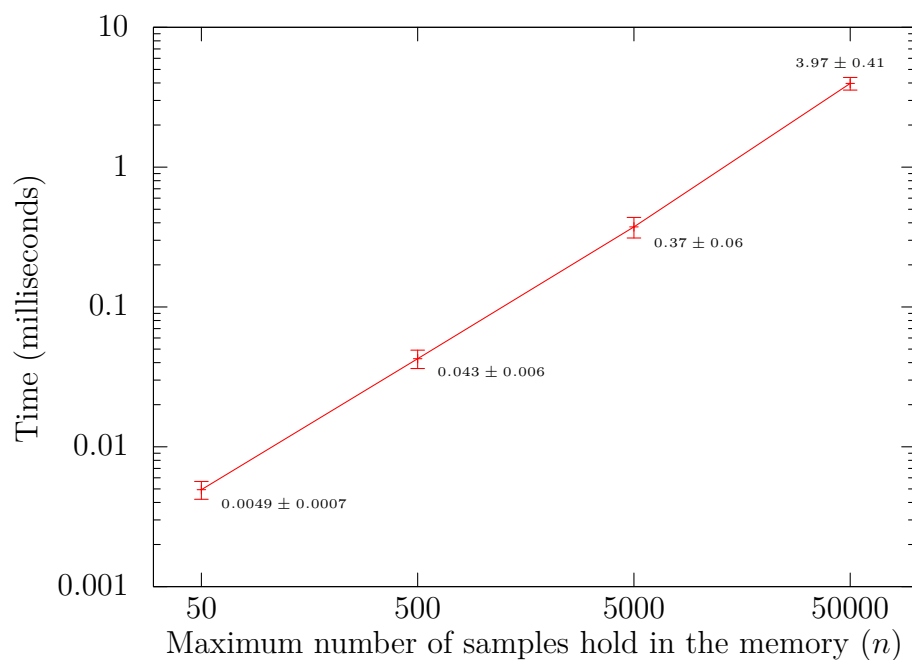
**Table 4.9:** Classification performance, macro-average F-measure (%), and storage reduction (%) of the IHC and IB3 algorithms for the UCI benchmark experiments.

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

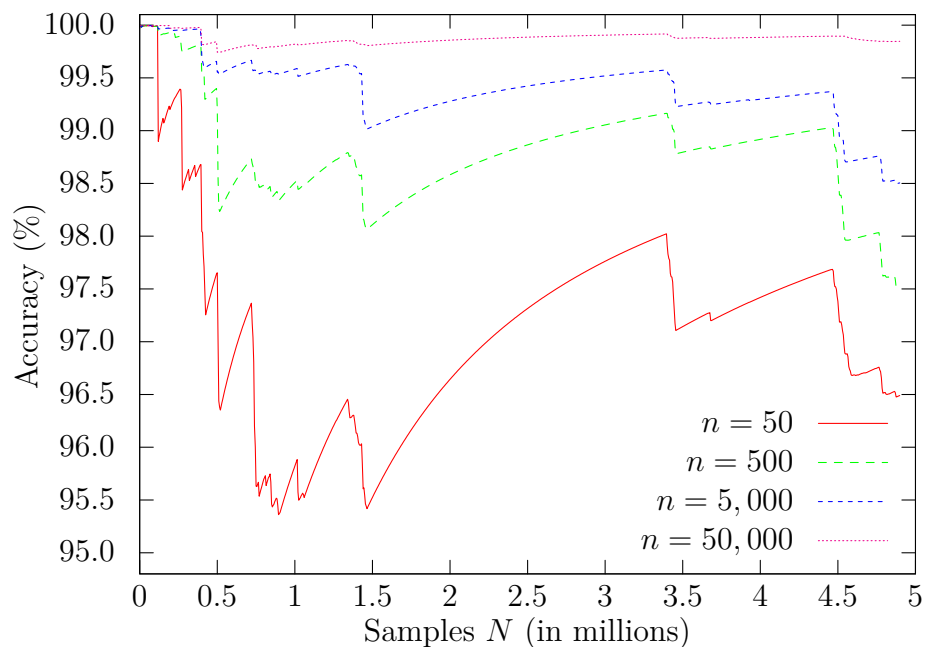
*survival*, *sonar* and *yeast* benchmarks where the F-Measure is improved respectively by 15.80%, 9.25%, 12.36% and 9.69%.

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 4.9 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 earlier in Section 3.4 (see page 46). 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 4.27 and 4.28 show respectively the time required to update the model and the accuracy according to the memory used by the algorithm.



**Figure 4.27:** Average time required to update the IHC model (after presenting a new sample) for the *KDD Cup 1999* dataset.

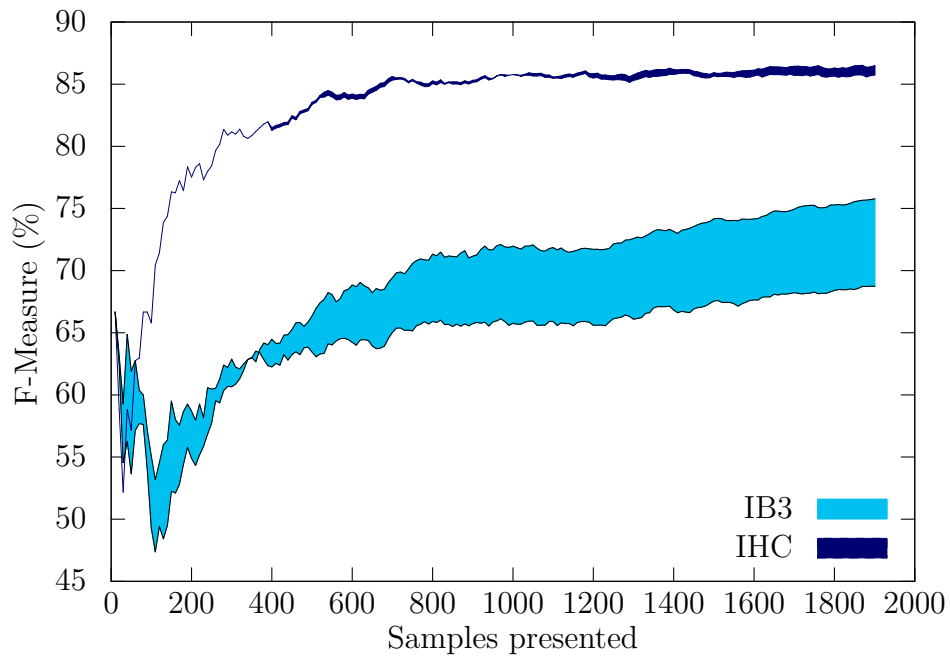


**Figure 4.28:** Accuracy of the IHC model for the *KDD Cup 1999* dataset.

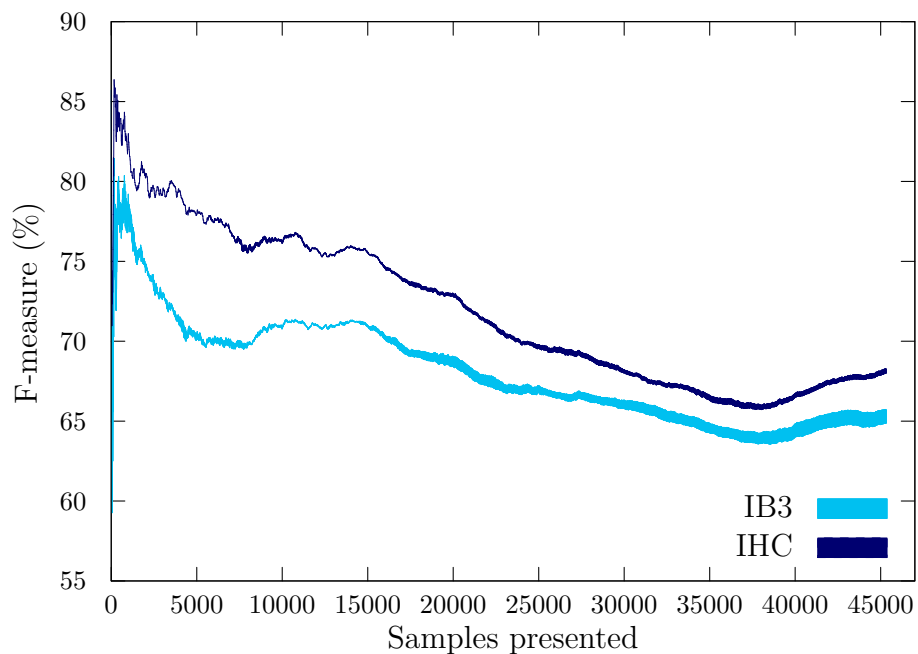
As expected the time necessary to update the model grows linearly with the amount of memory used, making the IHC a highly scalable algorithm. Updating the model requires approximately 4 milliseconds for  $n = 50,000$ , using the referred system. This demonstrates real-time model adaptability and knowledge 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 4.28). 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 (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 (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 4.28). 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 3.4 (see pages 46 and 49). Figures 4.29 and 4.30 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 [Beringer and Hüllermeier, 2007], 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 4.29). 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.



**Figure 4.29:** Evolution of the F-Measure for the Internet usage dataset.



**Figure 4.30:** Evolution of the F-Measure for the electricity dataset.



### Case study: Protein Membership Prediction

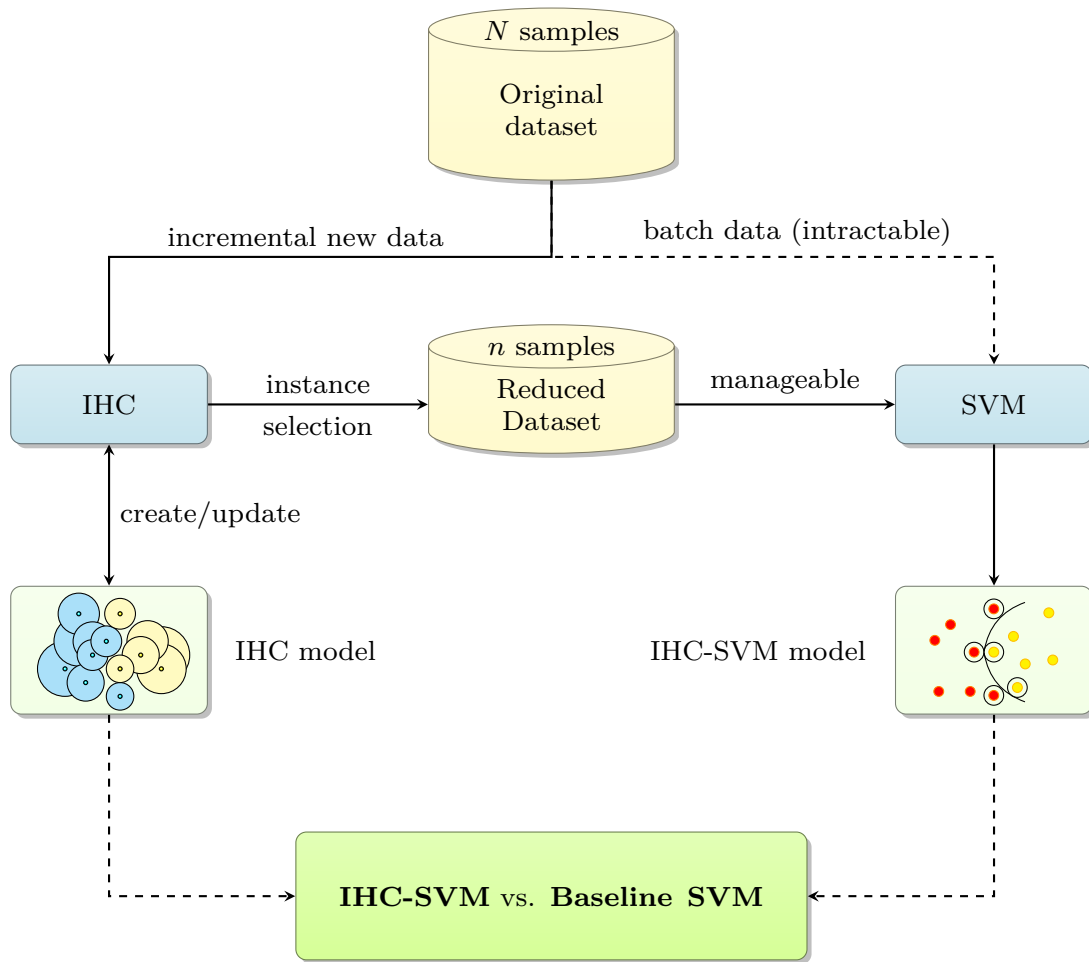
Due to the ever increasing biological databases, a fast response for protein classification prompts the need to expedite models' adaptation. In this context, two main approaches can be considered [Masud et al., 2010]. One approach consists of using a single model that is dynamically updated as new data 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 [Masud et al., 2010].

Concerning the protein membership prediction case study, described earlier in Section 3.5, (see page 53), 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 4.31 presents the combined learning framework, which works as follows: As new data becomes 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 4.32 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 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



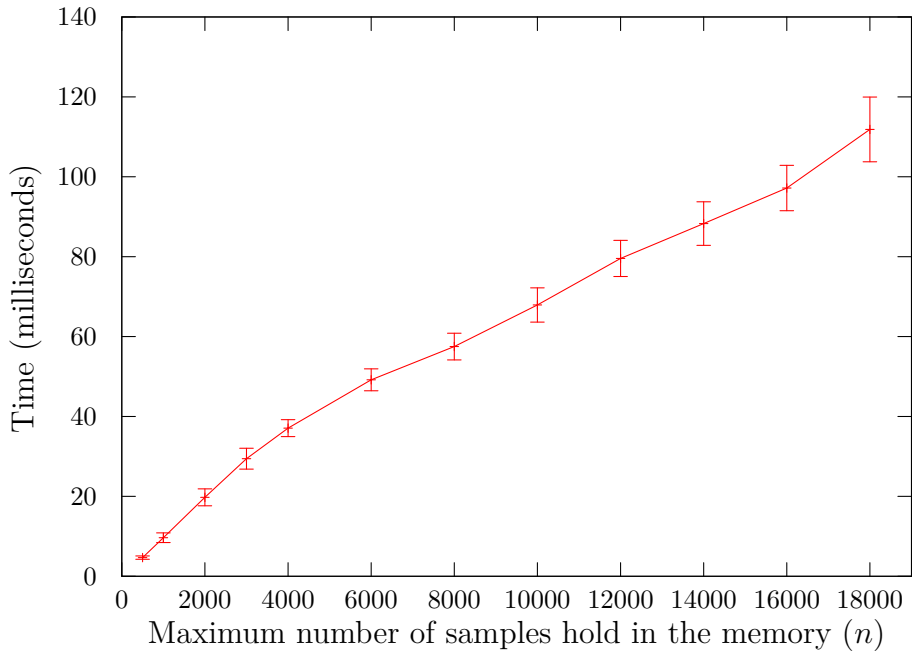
**Figure 4.31:** IHC-SVM learning framework.

F-measure for the test dataset of 95.91%. The same configurations were used to train the SVMs in the proposed (IHC-SVM) approach.

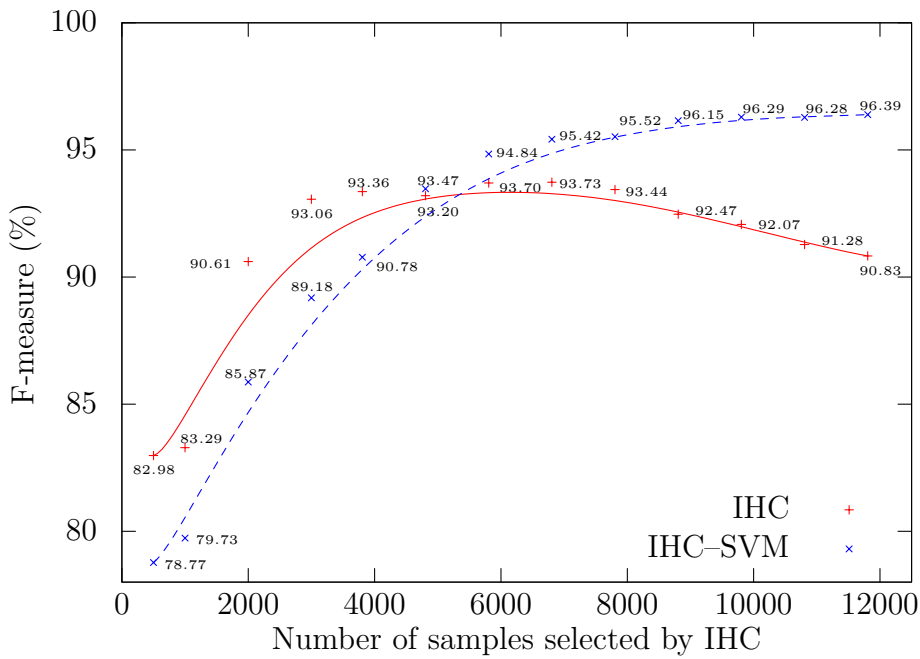
We tested both the IHC and the IHC-SVM approach for the concerned problem, using parameters  $g = 1$  and  $g = 2$  which have demonstrated to yield good results in Lopes and Ribeiro [Lopes and Ribeiro, 2011d]. Based on the information collected, we have set  $g = 2$ . Figure 4.33 shows the macro-average 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 in <http://193.137.78.18/ihc/>.

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



**Figure 4.32:** Average time required to update the IHC model (with a new sample) for the protein membership prediction case study.



**Figure 4.33:** IHC and IHC-SVM macro-average F-measure performance for the protein membership prediction case study.

**Table 4.10:** IHC-SVM storage reduction and classification improvement over the baseline (SVM).

| Number of samples<br>selected by IHC | Storage<br>reduction (%) | F-Measure<br>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                    | ↑ <b>0.24</b>                |
| 9,806                                | 42.87                    | ↑ <b>0.37</b>                |
| 10,806                               | 37.04                    | ↑ <b>0.37</b>                |
| 11,806                               | 31.22                    | ↑ <b>0.48</b>                |

data (under-fitting), since we will only be able to store a fraction of the samples that define the decision frontier (see Figure 4.33). 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 4.10 shows the gains of the proposed incremental-batch approach (IHC-SVM) over the baseline batch approach (SVM).

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 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.

## 4.4 Summary

In this Chapter, we have addressed two different strategies for handling large volumes of data. More specifically, we have shown that GPU parallel implementations of traditional batch algorithms can reduce considerably the time necessary to create models, therefore providing the means to convert otherwise intractable problems into manageable ones. Moreover, a new incremental learning algorithm (IHC) supporting real-time models' adaptation is also presented.

While NNs have proven to be suitable for solving many challenging problems, their long training times prevent them from being used in applications involving large datasets. In this context, we have proposed a GPU parallel implementation of the BP and MBP algorithms using the CUDA programming model. The experiments conducted demonstrate that the training time is considerably reduced (see Figures 4.11 to 4.19), thus allowing the deployment of computationally demanding NN applications.

A very important component of this work resides on the well-designed kernels built specifically to optimize the occupancy and maximize throughput on the deployed platforms. Our results relate both to classification performance and processing time and were achieved using five benchmarks and one real-world problem. This can be verified by comparing the GPU and CPU runs on the datasets, which uphold a considerable reduction in the NNs training time (see Figures 4.11 to 4.19).

The speedups obtained, ranging from  $5\times$  to  $180\times$  on computer system 2, are related to the complexity of the problem (see Tables 4.2 and 4.3 and Figures 4.12 to 4.15, 4.18, 4.19 and 4.21). Typically, the more complex the problem is (i.e. the more inputs, neurons and samples to process) the greater the speedups (over sequential approaches) obtained (see Figure 4.21). Moreover, we provide evidence that old model devices, such as the 8600 GT, can still provide significant speedups even when facing small problems (see Tables 4.2 and 4.3). This has a profound impact on real-world problems, as is the case of the VAs problem where we were able to reduce the work of weeks to a matter of hours and obtain improved quality models.

Although the GPU reduces significantly the drawback of the long training times of NNs, making their use more attractive, building a viable NN solution still requires a great deal of effort. Thus, we have also presented an ATS that is capable of automatically finding high-quality NNs-based solutions. The proposed system takes full advantage of the GPU potential, searching actively for better solutions without human intervention.

We have also presented a solution which integrates an NSIM into the NNs models, empowering them with the capacity to handle MVs. To our best knowledge this is

the first method that allows NNs, otherwise considered to be highly sensitive to MVs [Ye, 2003], to cope directly with this ubiquitous problem without requiring data to be preprocessed. Thus, NNs turn out to be positioned as an excellent alternative to other algorithms capable of dealing directly with the MVP, e.g. decision trees.

Through the use of selective inputs, the proposed approach accounts for the creation of different conceptual models, while maintaining a single physical model. The NSIM works as if we divide the original training dataset into several subsets of data containing all the combinations of complete features, one for each set of maximal independent data without MVs, in order to create an ensemble of NNs. However, by using a single physical model, the training time of the NSIM is that of a single model (a fraction of the time that would be required to create the ensemble).

The proposed solution presents several advantages as compared to traditional methods for handling MVs, making this a first-class method for dealing with this crucial problem: *(i)* it reduces the burden and the amount of time associated to 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 MAR or MCAR assumptions to hold, since only the known data is used actively to build the models; *(iv)* unlike preprocessing methods which may inject outliers into the data and cause undesirable bias, the NSIM uses the best conceptual model depending exclusively on the available data; *(v)* the NSIM may allow to infer and take advantage of any informative knowledge associated with the MVs; *(vi)* it presents 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; *(viii)* its classification performance, considering the MBP algorithm, is similar to state-of-the-art multiple imputation methods and the tests conducted show that the NSIM performs better than multiple imputation methods when the proportion of MVs is significant (more than 5% in our tests) or the prevalence of MVs affects a large number of features. This is validated in a real-world problem of bankruptcy prediction that attests to the quality and usefulness of the proposed method.

In a different line of work, we have presented a new incremental and highly-scalable algorithm (IHC) with multi-class support that can accommodate memory and computational restrictions, 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 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 streams (see Figures 4.27 and 4.32). Furthermore, the experiments demonstrate its ability to

update the models incrementally and handle concept drifts, while maintaining superior classification performance (see Figures 4.28 to 4.30). Additionally, the resulting models are interpretable, in the sense that we can provide the “nearest” sample that was used to reach the decision, making this algorithm useful even in domains where interpretability is a key factor. This is important because humans are reluctant to base their decisions on complex (black box) systems, preferring models that can provide some degree of information to support their findings [Cherkassky and Mulier, 2007].

Moreover, due to its capacity to minimize the impact of noisy samples, eventually removing them from memory, the algorithm can handle concept drift scenarios. Finally, since it keeps the instances (that are believed to be lying) on the decision frontier, it can also be an optimal choice for selecting a representative subset of the data for more sophisticated algorithms. In this context, we have presented a learning framework approach (IHC-SVM) for predicting protein membership which is able to deal with the dynamic everyday changes of real-world biological databases. It has been demonstrated that under certain conditions the IHC-SVM presents a better performance than the baseline SVM, using sequences of proteins built from well-known peptidase repositories (see Table 4.10). Moreover, there is evidence that using IHC as the first step to determine the relevance of each sample and deciding which ones to retain and which ones to discard, is an efficient procedure for the incremental learning framework. The SVM training on the reduced data set builds a model that yields high accuracy which is desirable to handle the dynamic (and pervasive) biological databases.

### **Future Work**

Future work will concentrate on enhancing the ATS, by using topology-modifying algorithms (e.g. constructive and pruning methods [Nicoletti et al., 2009, Zapranis and Refenes, 1999]). Moreover, the GPU implementation of other supervised algorithms and their eventual inclusion in the ATS is also envisioned.

In addition, we will exploit the possibility of adapting the NSIM to other types of NNs and ML algorithms.

Concerning the IHC, we will extend this study to further experiments involving larger dynamic datasets. Moreover, another line of work consists of evaluating the impact of using different values of  $g$  for distinct classes as well as adjusting them automatically.





---

 Unsupervised and Semi-supervised algorithms
 

---



---

|            |  |            |
|------------|--|------------|
| <b>5.1</b> | <b>Non-Negative Matrix Factorization (NMF)</b> . . . . . | <b>127</b> |
| 5.1.1      | NMF Algorithm . . . . .                                  | 128        |
| 5.1.2      | Combining NMF with other ML Algorithms . . . . .         | 131        |
| 5.1.3      | Semi-Supervised NMF (SSNMF) . . . . .                    | 131        |
| 5.1.4      | GPU Parallel Implementation . . . . .                    | 134        |
| 5.1.5      | Results and Discussion . . . . .                         | 139        |
| <b>5.2</b> | <b>Deep Belief Networks (DBNs)</b> . . . . .             | <b>153</b> |
| 5.2.1      | Restricted Boltzmann Machines (RBMs) . . . . .           | 156        |
| 5.2.2      | Deep Belief Networks Architecture . . . . .              | 162        |
| 5.2.3      | Adaptive Step Size Technique . . . . .                   | 163        |
| 5.2.4      | GPU parallel implementation . . . . .                    | 163        |
| 5.2.5      | Results and Discussion . . . . .                         | 170        |
| <b>5.3</b> | <b>Summary</b> . . . . .                                 | <b>182</b> |

---

Unlike supervised learning algorithms, which learn a mapping model from a given set of inputs to a predefined set of outputs, depending on their corresponding target values, unsupervised algorithms cannot use any error criterion that is based on targets or in any other external information [Alpaydin, 2010, Marsland, 2009]. Instead, they must rely exclusively on the information encompassed within the (observed) input data,  $\mathbf{X}$ . Rationally, for a given problem, the input space is shaped (structured) in such a way than certain patterns are more likely to occur than others [Marsland, 2009]. For example, in the MNIST problem (see Section

3.4, page 49) we can expect the dataset images (input features) to present an innate set of characteristics that are inherently different from the ones found in other problems (e.g. face recognition). Therefore, unsupervised algorithms can exploit the regularities found within the training dataset, grouping together samples that present some degree of similarity among them [Marsland, 2009]. Ideally, the resulting models would be able to capture the latent variables for concepts of interest, directly from the unlabeled raw input data.

In a complementary perspective, we may view unsupervised learning algorithms as a tool to uncover the hidden structure of the data in order to learn representations that are potentially more discriminative than the original input data and thus more suitable for supervised ML algorithms [Ranzato et al., 2007].

Many unsupervised methods create a representation of the data that is constrained in order to have specific and desirable properties (e.g. distributed representation, low-dimensionality, sparsity), from which the original input data can be reconstructed [Ranzato et al., 2007]. Other methods focus on approximating the data density by stochastically reconstructing the input data from the created representation [Ranzato et al., 2007]. Typically good representations preserve the information that is useful for the desired task (e.g. detection, recognition, prediction, visualization) while discarding noise and other irrelevant data variabilities [Ranzato et al., 2007].

In this Chapter we present two different unsupervised learning approaches and their corresponding GPU parallel implementations. First, in Section 5.1 we address the work related with the NMF algorithm, which includes a new semi-supervised method, designated by SSNMF. The NMF unsupervised learning approach consists of projecting the input data into a lower dimensional manifold, thus reducing the number of features of a dataset, while retaining the essential information in order to reconstruct the original data. Section 5.1 is structured as follows. Section 5.1.1 describes the NMF algorithm. Section 5.1.2 explains how to combine NMF with other ML algorithms. Section 5.1.3 describes the SSNMF algorithm, which reduces the computational cost while improving the accuracy of NMF-based models. Section 5.1.4 details the GPU parallel implementation of the NMF algorithm. Finally, Section 5.1.5 presents and analyzes the results obtained in several experiments, concerning the face recognition domain, both for the NMF GPU implementation and for the SSNMF method.

Second, Section 5.2 addresses the work related with the DBNs, which includes an adaptive step size technique for accelerating the training convergence. A DBN is an energy-based deep architecture generative model, which aims to maximize the likelihood of the training data. By contrast with the NMF approach, the building blocks (RBMs) of a DBN allow for overcomplete representations. Section 5.2 is organized as follows. Sections 5.2.1 and 5.2.2 detail respectively the RBMs and the DBNs. Section 5.2.3 presents the proposed adaptive step size technique. Section 5.2.4 focuses on the GPU parallel implementation of the RBMs and DBNs. Lastly, Section 5.2.5 asserts the validity of both approaches (the adaptive step

size technique and the GPU implementation) to speedup the training process and analyzes the effects of varying the number of layers and neurons in a DBN. To this end, the MNIST database of hand-written digits (see Section 3.4, page 49) and the HHreco multi-stroke symbol database (see Section 3.4, page 46) were used.

Finally, Section 5.3 concludes this Chapter and points out directions for future lines of work.

## 5.1 Non-Negative Matrix Factorization (NMF)

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) [Verleysen et al., 2009]. 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 [Verleysen et al., 2009].

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-dimensional data [Verleysen, 2003].

The number of data samples ( $N$ ) required to estimate a function of several variables grows exponentially with the number of dimensions ( $D$ ) [Verleysen et al., 2009]. 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 [Verleysen et al., 2009]. 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  $D$  [Verleysen, 2003]. Therefore, according to Verleysen, any learning task should begin by an attempt to reduce the dimension of the data, since this is a key issue in high-dimensional learning [Verleysen, 2003]. 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) [Verleysen et al., 2009, Verleysen, 2003]. 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 [Zilu and Guoyi, 2009, Lee and Seung, 1999], 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 [Gillis and Glineur, 2010, Li et al., 2010, Ribeiro et al., 2009, Zilu and Guoyi, 2009]. 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 [Verleysen, 2003], therefore allowing for the improvement of the accuracy of the models. Third, the computational cost associated with the problem is usually reduced [Garg and Murty, 2009] (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 [Verleysen, 2003]. 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.

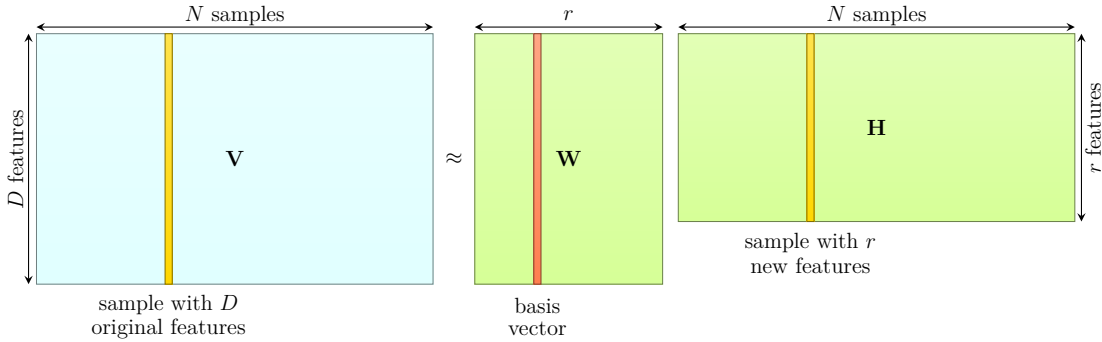
### 5.1.1 NMF Algorithm

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):

$$\mathbf{V} \approx \mathbf{WH}. \quad (5.1)$$

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 [Xu et al., 2003].

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  $\mathbf{V}$  [Li



**Figure 5.1:** NMF factorization.

et al., 2010]. 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 5.1 illustrates this idea.

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) [Li et al., 2010]. Moreover, the localized nature of the extracted features is compatible with the intuitive notion of combining parts to form a whole [Zilu and Guoyi, 2009]. 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 [Lee and Seung, 1999]. 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 [Gillis and Glineur, 2010].

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  $\mathbf{X} \in \mathbb{R}_+^{N \times D}$ .

### Cost Functions

In order to measure the quality of the approximation defined in (5.1) 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 (5.2) and the (generalized) Kullback-Leibler divergence, given by (5.3):

$$\|\mathbf{V} - \mathbf{WH}\|^2 = \sum_{ij} ((\mathbf{V})_{ij} - (\mathbf{WH})_{ij})^2. \quad (5.2)$$

$$D(\mathbf{V} \parallel \mathbf{WH}) = \sum_{ij} \left( (\mathbf{V})_{ij} \log \frac{(\mathbf{V})_{ij}}{(\mathbf{WH})_{ij}} - (\mathbf{V})_{ij} + (\mathbf{WH})_{ij} \right). \quad (5.3)$$

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 (5.2) and (5.3) 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 [Lee and Seung, 2000].

### Multiplicative Update Rules

Considering the multiplicative rules and the Euclidean distance metric, the updates specified in (5.4) and (5.5) can be used iteratively, until a good approximation of  $\mathbf{V}$  is found:

$$(\mathbf{H})_{a\mu} \leftarrow (\mathbf{H})_{a\mu} \frac{(\mathbf{W}^\top \mathbf{V})_{a\mu}}{(\mathbf{W}^\top \mathbf{WH})_{a\mu}}, \quad (5.4)$$

$$(\mathbf{W})_{ia} \leftarrow (\mathbf{W})_{ia} \frac{(\mathbf{VH}^\top)_{ia}}{(\mathbf{WHH}^\top)_{ia}}. \quad (5.5)$$

Similarly, (5.6) and (5.7) can be used for the divergence metric and the multiplicative update rules:

$$(\mathbf{H})_{a\mu} \leftarrow (\mathbf{H})_{a\mu} \frac{\sum_i (\mathbf{W})_{ia} (\mathbf{V})_{i\mu} / (\mathbf{WH})_{i\mu}}{\sum_k (\mathbf{W})_{ka}}, \quad (5.6)$$

$$(\mathbf{W})_{ia} \leftarrow (\mathbf{W})_{ia} \frac{\sum_\mu (\mathbf{H})_{a\mu} (\mathbf{V})_{i\mu} / (\mathbf{WH})_{i\mu}}{\sum_v (\mathbf{H})_{av}}. \quad (5.7)$$

### Additive Update Rules

An alternative to the multiplicative update rules can be obtained by using the gradient descent technique. In such a case (5.8) and (5.9) can be applied iteratively, for the Euclidean distance, until a good approximation of  $\mathbf{V}$  is found:

$$(\mathbf{H})_{a\mu} \leftarrow \max(0, (\mathbf{H})_{a\mu} + \eta_{a\mu} [(\mathbf{W}^\top \mathbf{V})_{a\mu} - (\mathbf{W}^\top \mathbf{WH})_{a\mu}]), \eta_{a\mu} = \frac{(\mathbf{H})_{a\mu}}{(\mathbf{W}^\top \mathbf{WH})_{a\mu}}, \quad (5.8)$$

$$(\mathbf{W})_{ia} \leftarrow \max(0, (\mathbf{W})_{ia} + \gamma_{ia} [(\mathbf{VH}^\top)_{ia} - (\mathbf{WHH}^\top)_{ia}]), \gamma_{ia} = \frac{(\mathbf{W})_{ia}}{(\mathbf{WHH}^\top)_{ia}}. \quad (5.9)$$

Similarly (5.10) and (5.11) can be used for the divergence:

$$(\mathbf{H})_{a\mu} \leftarrow \max(0, (\mathbf{H})_{a\mu} + \eta_{a\mu} \left[ \sum_i (\mathbf{W})_{ia} \frac{(\mathbf{V})_{i\mu}}{(\mathbf{WH})_{i\mu}} - \sum_i (\mathbf{W})_{ia} \right]), \eta_{a\mu} = \frac{(\mathbf{H})_{a\mu}}{\sum_i (\mathbf{W})_{ia}}, \quad (5.10)$$

$$(\mathbf{W})_{ia} \leftarrow \max(0, (\mathbf{W})_{ia} + \gamma_{ia} \left[ \sum_j (\mathbf{H})_{aj} \frac{(\mathbf{V})_{ij}}{(\mathbf{WH})_{ij}} - \sum_j (\mathbf{H})_{aj} \right]), \gamma_{ia} = \frac{(\mathbf{W})_{ia}}{\sum_j (\mathbf{H})_{aj}}. \quad (5.11)$$

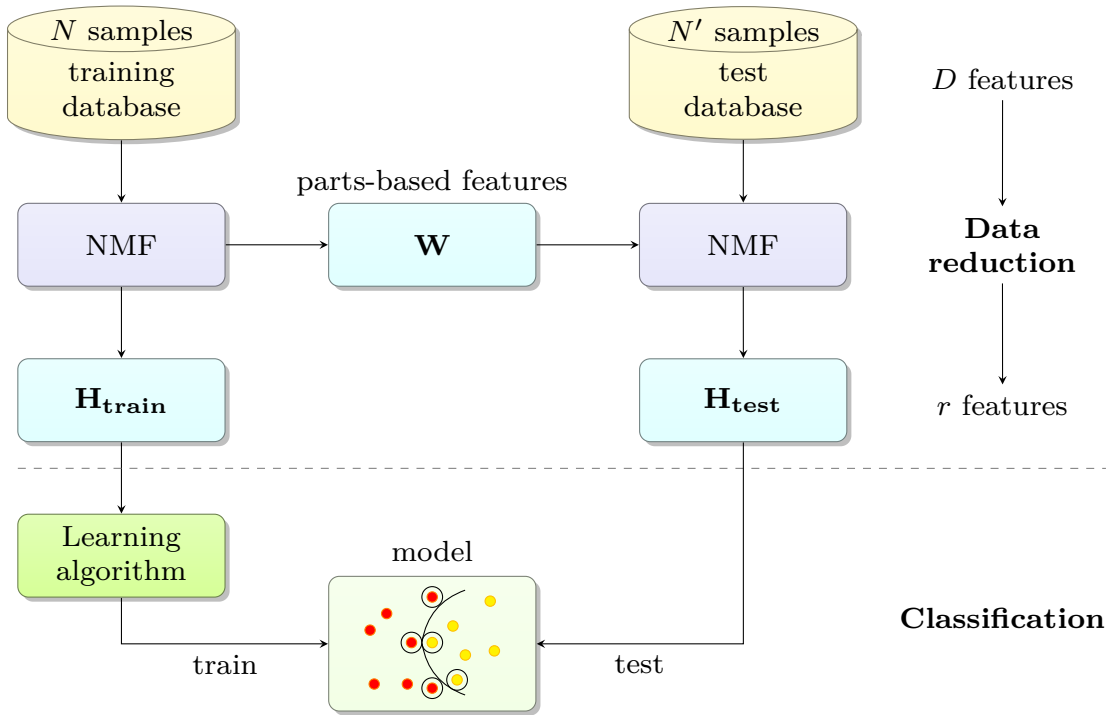
### 5.1.2 Combining NMF with other ML Algorithms

In order to combine NMF with other algorithms we use the procedure described in Ribeiro et al. [Ribeiro et al., 2009]. Accordingly, Figure 5.2 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  $\mathbf{H}_{\text{train}} \in \mathbb{R}_+^{r \times N}$  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  $\mathbf{H}_{\text{train}}$  (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  $\mathbf{H}_{\text{test}} \in \mathbb{R}_+^{r \times N'}$ , 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  $\mathbf{H}_{\text{test}}$  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.

Note that each time new data is gathered to be used by the resulting classifier, a brand-new  $\mathbf{H}$  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  $\mathbf{W}$  remains invariant, computing  $\mathbf{H}$  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 5.1.4, is a fundamental step towards softening this problem.

### 5.1.3 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,  $\mathbf{W}$ , will most likely be shared by objects of all the classes. To overcome this problem, we propose to partition

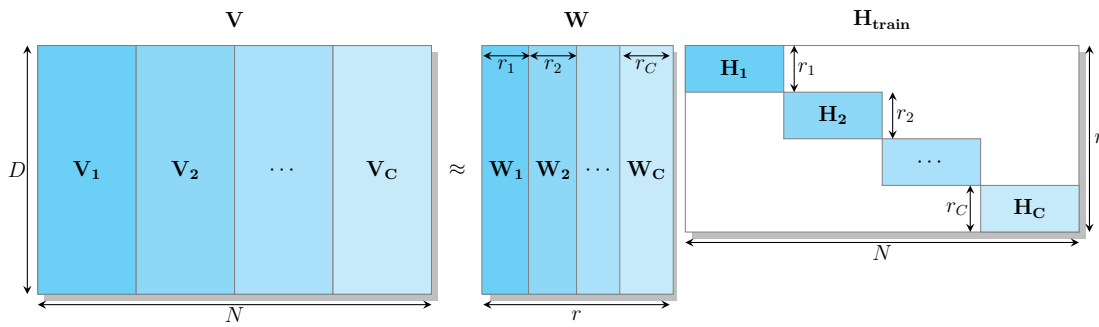


**Figure 5.2:** Combining NMF with other learning algorithms.

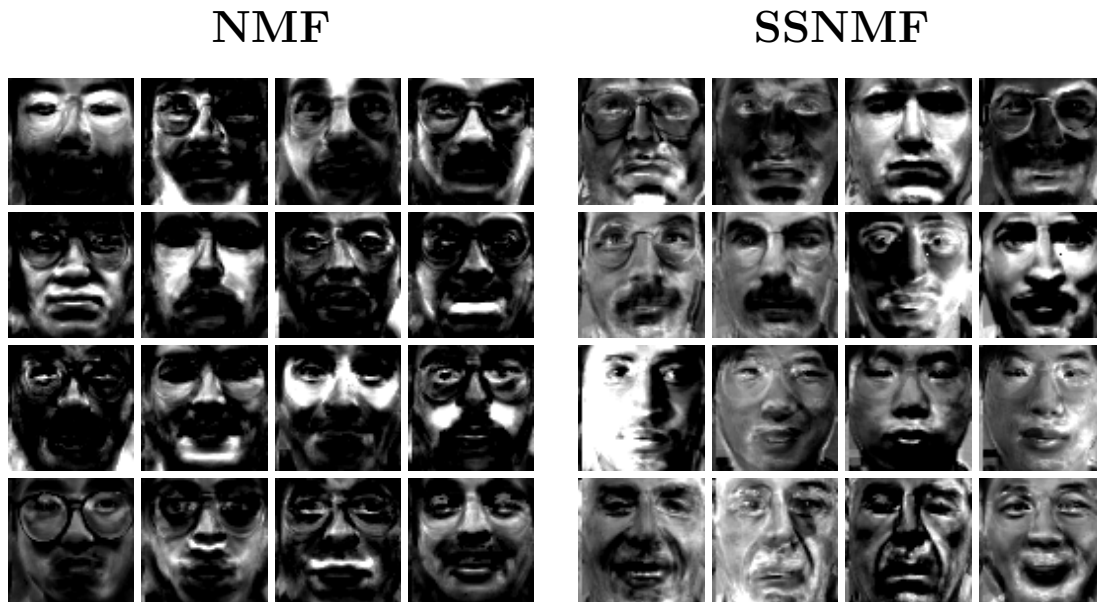
the matrix,  $\mathbf{V}$ , containing the original inputs (features) of the training data into sub-matrices,  $\mathbf{V}_1, \mathbf{V}_2, \dots, \mathbf{V}_C$ , each containing the samples' inputs of one of the  $C$  different classes. The NMF algorithm is then independently applied to each one of the sub-matrices, using a smaller number of basis vectors  $r_1, r_2, \dots, r_C$  such that  $r = \sum_{i=1}^C r_i$ . This results in the creation of  $C$  parts-based feature matrices,  $\mathbf{W}_1, \mathbf{W}_2, \dots, \mathbf{W}_C$ , and  $C$  codification matrices,  $\mathbf{H}_1, \mathbf{H}_2, \dots, \mathbf{H}_C$ , which are then combined to create a global  $\mathbf{W}$  matrix and a global  $\mathbf{H}_{\text{train}}$  matrix. Figure 5.3 represents this process, where the white areas of the  $\mathbf{H}_{\text{train}}$  matrix correspond to zero value elements [Lopes and Ribeiro, 2011b].

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 5.4 shows the typical basis vectors generated by the NMF and SSNMF methods for the Yale face database (described earlier in Section 3.4, page 49). 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 3.8, page 52). Finally, it is worth mentioning that the SSNMF method applies only to the computation of





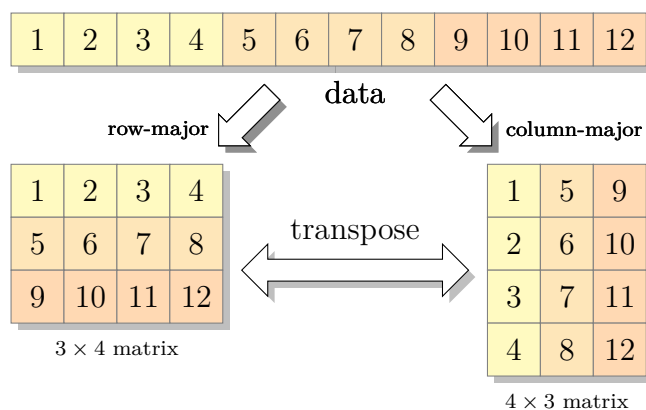
**Figure 5.3:** Generation and combination of the individual class matrices. The white areas of the  $H_{\text{train}}$  matrix correspond to zero value elements.



**Figure 5.4:** Typical basis vectors ( $W$  columns) generated by NMF and SSNMF for the Yale face database (using  $r_i = 3$  and  $r = 45$  ( $3 \times 15$ )).

$W$  and  $H_{\text{train}}$ . The  $H_{\text{test}}$  matrix is calculated in the same manner (unsupervised) as described in Section 5.1.2 [Lopes and Ribeiro, 2011b].

For many real-world problems, obtaining labeled data is a relatively expensive process [Chapelle et al., 2006]. 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 in order to create a classifier model



**Figure 5.5:** Interpretation of the same data, using either row-major or column-major orders.

(see Figure 5.2), we can still use all the data (labeled and unlabeled) for the data reduction phase, 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 samples) to it. To prevent the original (SSNMF) extracted characteristics from being completely overwritten, matrix  $\mathbf{W}$  can remain fixed during a predefined number of iterations or until there is no significant gain in updating  $\mathbf{H}$  alone.

#### 5.1.4 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 [Lopes and Ribeiro, 2012c, Lopes and Ribeiro, 2010a].

##### 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 29). Notice however that we avoid the computation of the transpose matrices (see (5.4), (5.5), (5.8) and (5.9)), 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 5.5.

---

**Listing 5.1:** 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);
    }
}

```

---

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 than calculating  $(\mathbf{W}\mathbf{H})\mathbf{H}^\top$ ). 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 5.1, updates each element of a given matrix  $\mathbf{X}$  according to (5.12):

$$X_{ij} \leftarrow X_{ij} \frac{A_{ij}}{B_{ij}}, \quad (5.12)$$

where  $\mathbf{A}$  and  $\mathbf{B}$  are matrices of the same size as  $\mathbf{X}$  (see (5.4) and (5.5)).

Listing 5.2 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.

Similarly to the multiplicative rules, the additive update rules also require an additional kernel (`UpdateMatrix_AE`), reproduced in Listing 5.3, which updates all the elements of a given matrix  $\mathbf{X}$  according to (5.13):

$$X_{ij} \leftarrow \max(0, X_{ij} + \frac{X_{ij}}{B_{ij}}(A_{ij} - B_{ij})), \quad (5.13)$$

where once again  $\mathbf{A}$  and  $\mathbf{B}$  are matrices of the same size as  $\mathbf{X}$  (see (5.8) and (5.9)).

### 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 (5.6). Similarly, `SumH` calculates  $\sum_v H_{av}$  for each

---

**Listing 5.2:** 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  $W^T W H$ 
    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  $V H^T$ 
    DeviceMatrix<cuDafloat>::Multiply(V, Ht, VHT);

    // Calculate  $H H^T$ 
    DeviceMatrix<cuDafloat> & HHT = WtW;
    Ht.ReplaceByTranspose();
    H.MultiplyBySelfTranspose(HHT);

    // Calculate  $W H H^T$ 
    DeviceMatrix<cuDafloat>::Multiply(W, HHT, WHHT);

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

---

---

**Listing 5.3:** 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;
    }
}

```

---

row  $a$  of  $\mathbf{H}$ , placing the result in a vector also with dimension  $r$ , see (5.7). Listing 5.4 presents the code of the `SumW` kernel, which uses the reduction process, described earlier in Section 2.5 (page 31). Note that for simplification, the code presented here, assumes that  $\mathbf{W}$  is stored in column-major order. However, the GPULib 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.

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  $\mu$  of both  $\mathbf{V}$  and  $\mathbf{WH}$ , as shown in Figure 5.6. 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 3.2, page 38), we found that we were able to store at least  $32 \times 32$  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 3.2 (page 38) and 2.1 (page 21)). 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

**Listing 5.4:** 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();
    }

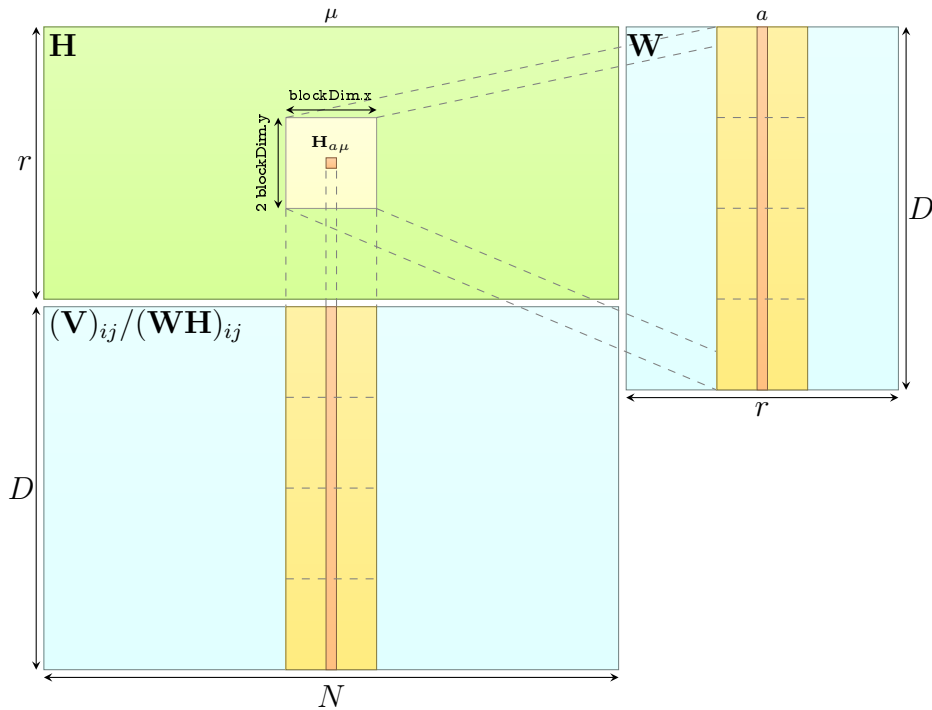
    if (blockSize >= 128) {
        if (threadIdx.x < 64) w[threadIdx.x] += w[threadIdx.x + 64];
        __syncthreads();
    }

    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;
        }
    }
}
```

---



**Figure 5.6:** Processing carried out, for each element  $\mathbf{H}_{a\mu}$ , by the `UpdateH_MD` kernel.

of one, and updates two elements of  $\mathbf{H}$  (observe Figure 5.6). 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`.

### 5.1.5 Results and Discussion

#### 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 [Wang et al., 2010]. 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 [Wang et al., 2010, Zhao et al., 2003]. 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 [Zhao et al., 2003]. A detailed survey on existing techniques and methods for face recognition can be found in [Zhao et al., 2003]. Typically, solving this problem involves several phases: (i) segmentation of the faces, (ii) extraction of relevant features from the face regions, (iii) recognition and (iv) verification [Zhao et al., 2003]. 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 5.1.2.

Altogether, in our testbed experiments we have used three different databases: the CBCL, Yale and AT&T databases. These were described in Section 3.4 (see pages 43, 46 and 49).

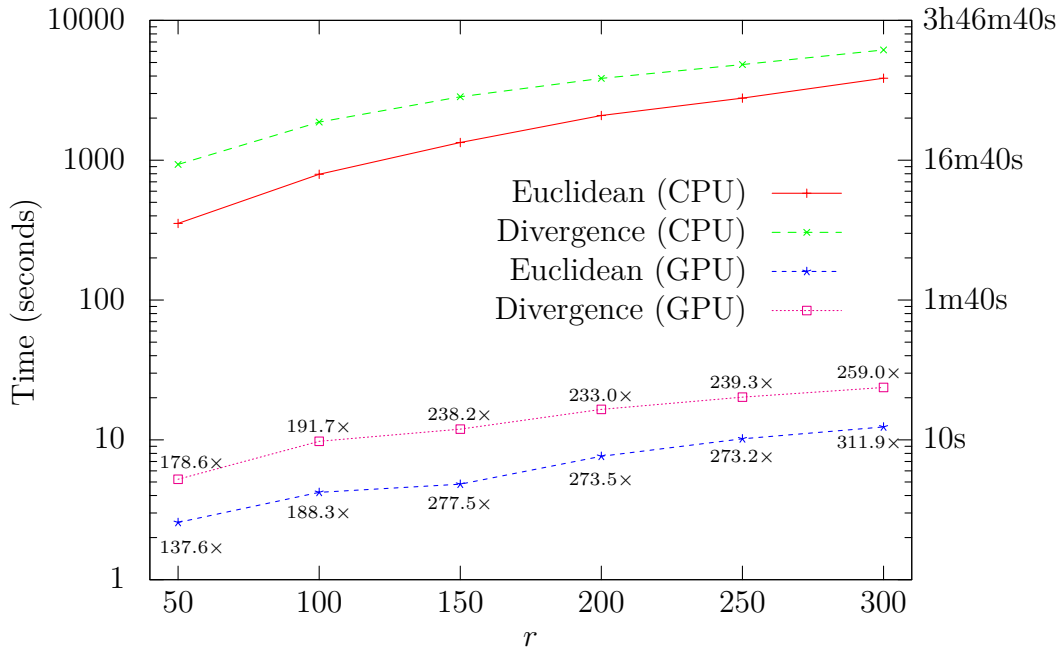
The CBCL face database #1 was used specifically to test and validate the GPU parallel implementations of the NMF algorithm. The tests were conducted using the 2,429 face images of the training dataset. The matrix containing the face images was created by placing one image per column. Thus, in this case, matrix  $\mathbf{V}$  is composed by 361 rows ( $19 \times 19$  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 5.1.2 as well as the performance of the SSNMF method described earlier (see Section 5.1.3). 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,  $\mathbf{V}_{\text{train}}$ , is composed of 4,096 rows ( $64 \times 64$  pixels) and 150 columns (face images), while the test matrix,  $\mathbf{V}_{\text{test}}$ , is composed of 4,096 rows and 15 columns. Similarly, in the case of the AT&T (ORL) database,  $\mathbf{V}_{\text{train}}$  is composed of 10,304 ( $112 \times 92$ ) rows and 360 columns and  $\mathbf{V}_{\text{test}}$  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 4.1.4). Accordingly, we have also used the process, described in Section 5.1.2, 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,  $\mathbf{V}_{\text{train}}$ , is composed of 4,096 rows and 120 columns, while the test matrix,  $\mathbf{V}_{\text{test}}$ , 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.





**Figure 5.7:** Time required to run the NMF algorithms on the CBCL face database, during 1,000 iterations, using the multiplicative update rules. The speedups ( $\times$ ) provided by the GPU are shown in the respective lines.

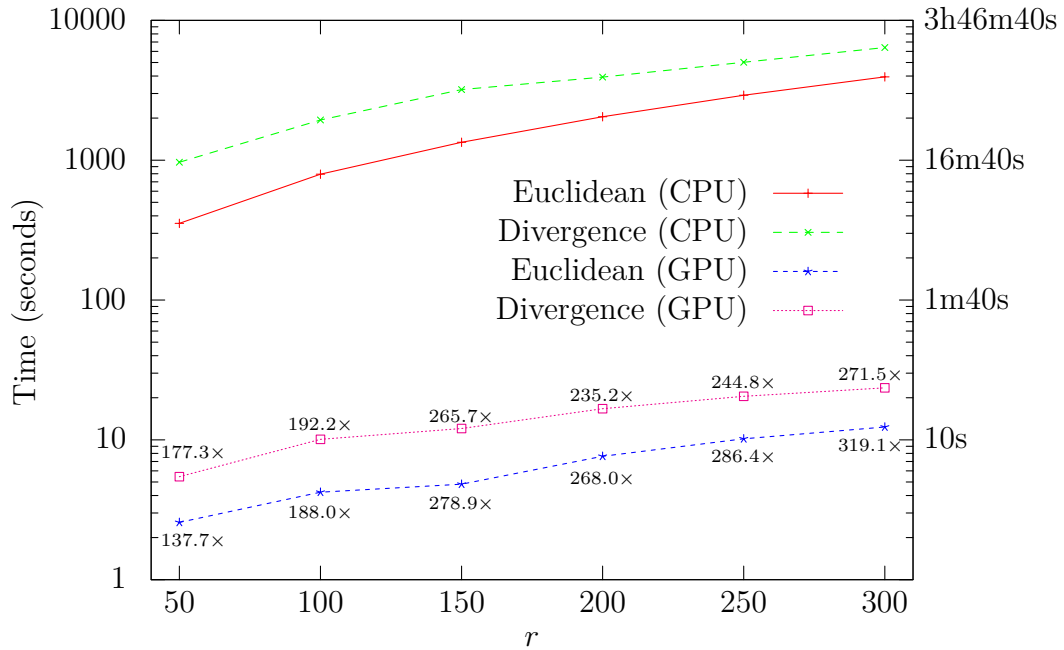
Before running the experiments, a histogram equalization was applied to the datasets' images, in order to reduce the influence of the surrounding illumination (see Section 3.6). Moreover, all the tests were performed using the computer system 2 (see Table 3.1, page 38).

### 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 5.7 and 5.8 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



**Figure 5.8:** Time required to run the NMF algorithms on the CBCL face database, during 1,000 iterations, using the additive update rules. The speedups ( $\times$ ) provided by the GPU are shown in the respective lines.

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 5.9 which exhibits the speedups provided by the GPU over the CPU.

Figure 5.10 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 5.1.2) 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 4.1.4). 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  $\mathbf{H}_{\text{train}}$  that will later be used to create (train) the

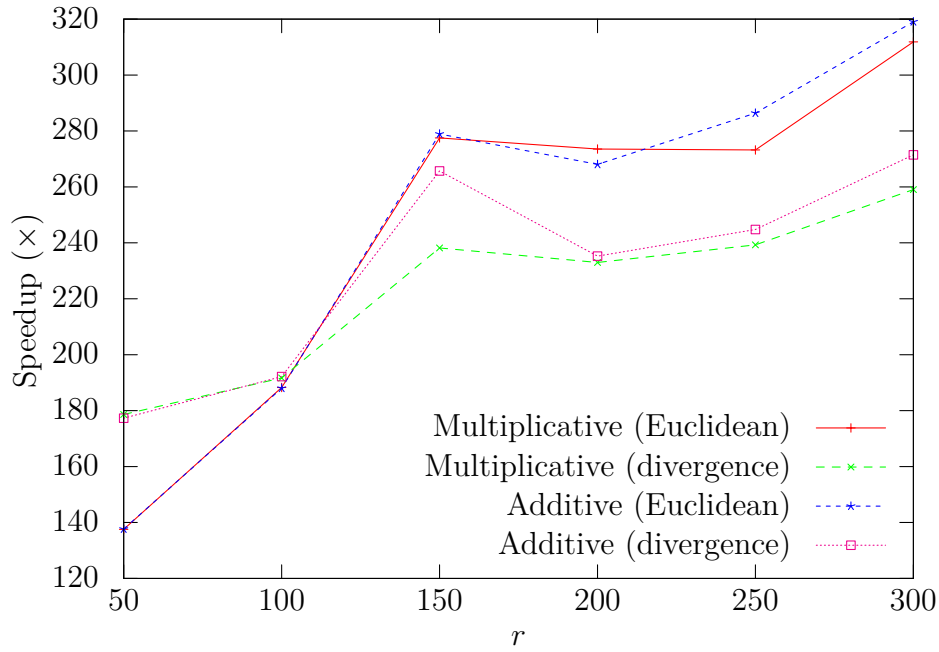


Figure 5.9: NMF GPU Speedups for the CBCL face database.

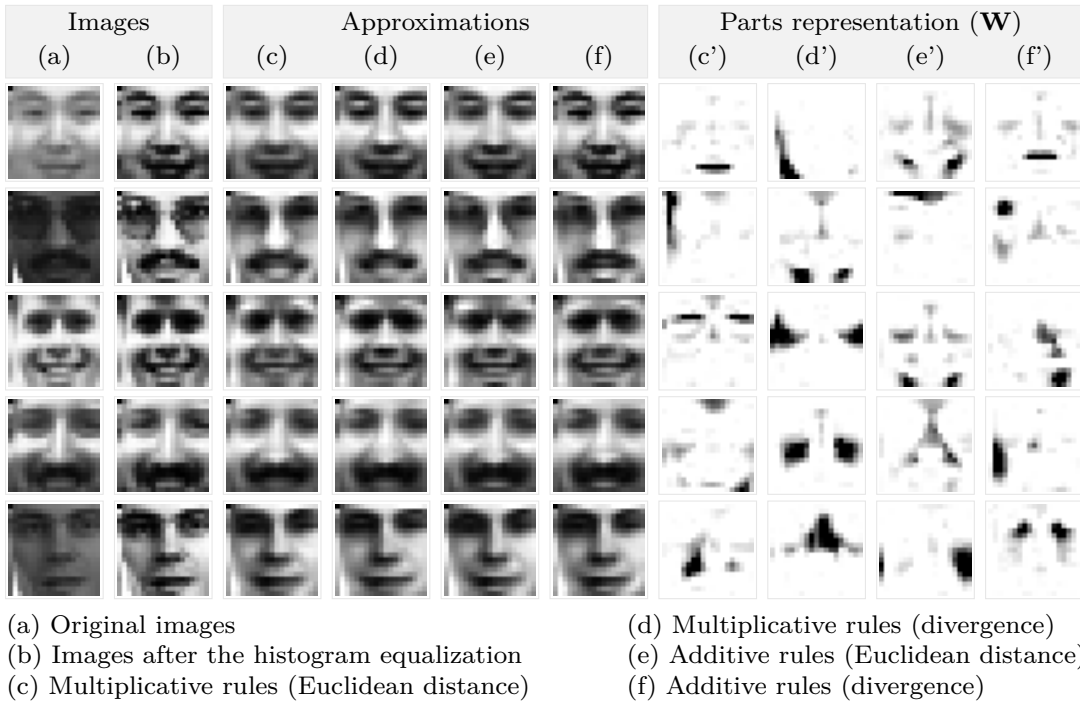


Figure 5.10: Approximations and parts representation generated by the NMF algorithms.

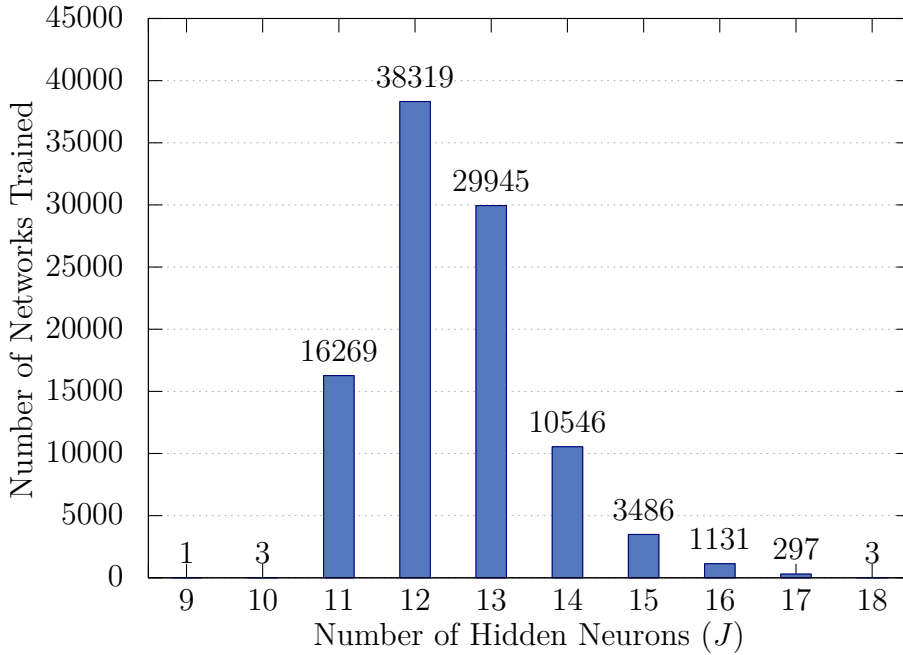


**Figure 5.11:** Parts-based faces representations,  $\mathbf{W}$ , generated by NMF for the Yale dataset.

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 shared by several individuals. Figure 5.11 shows the first 40 images of  $\mathbf{W}$ , obtained with the specified configuration.

Following the computation of  $\mathbf{W}$  and  $\mathbf{H}_{\text{train}}$ , we use the NMF algorithm again, this time on the test dataset (containing 45 samples) in order to obtain the  $\mathbf{H}_{\text{test}}$  matrix, necessary to assess the quality of the resulting NN classifiers (see Figure 5.2).

The  $\mathbf{H}_{\text{train}}$  and  $\mathbf{H}_{\text{test}}$  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 4.1.4). For the problem at hand, the ATS took less than 16 hours to train a total of 100,000 networks. Figure 5.12 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).



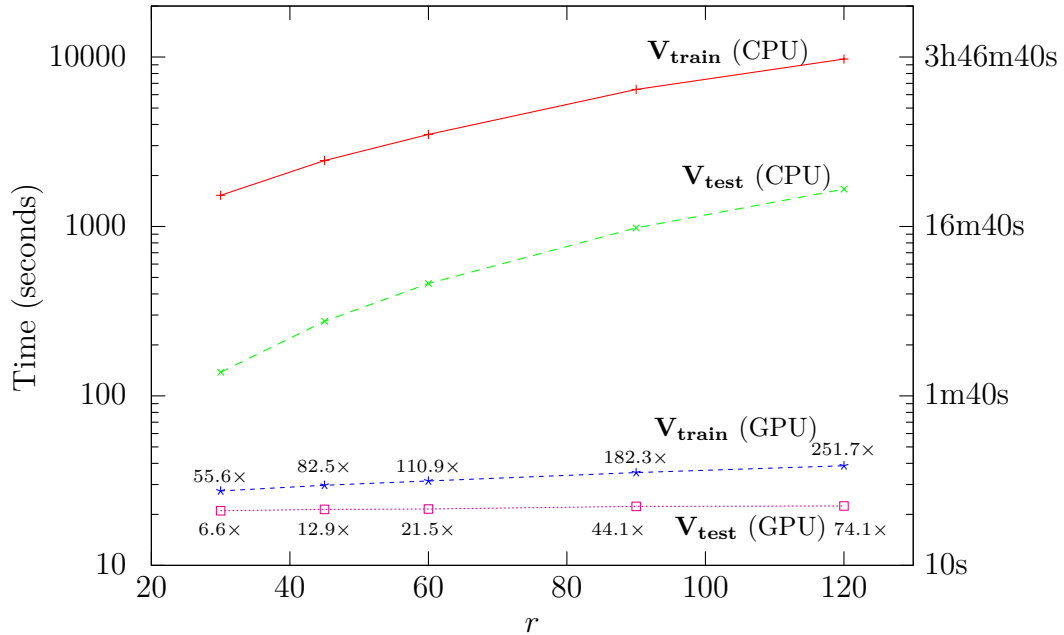
**Figure 5.12:** Number of networks trained by the ATS, according to the number of neurons.

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 ( $3 \times 15$ ), while in the AT&T dataset,  $r$  was set to 120 ( $3 \times 40$ ).

Figure 5.13 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  $\mathbf{H}_{\text{test}}$  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).

Computing the  $\mathbf{W}$  and  $\mathbf{H}_{\text{train}}$  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  $\mathbf{H}_{\text{test}}$  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 5.14 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



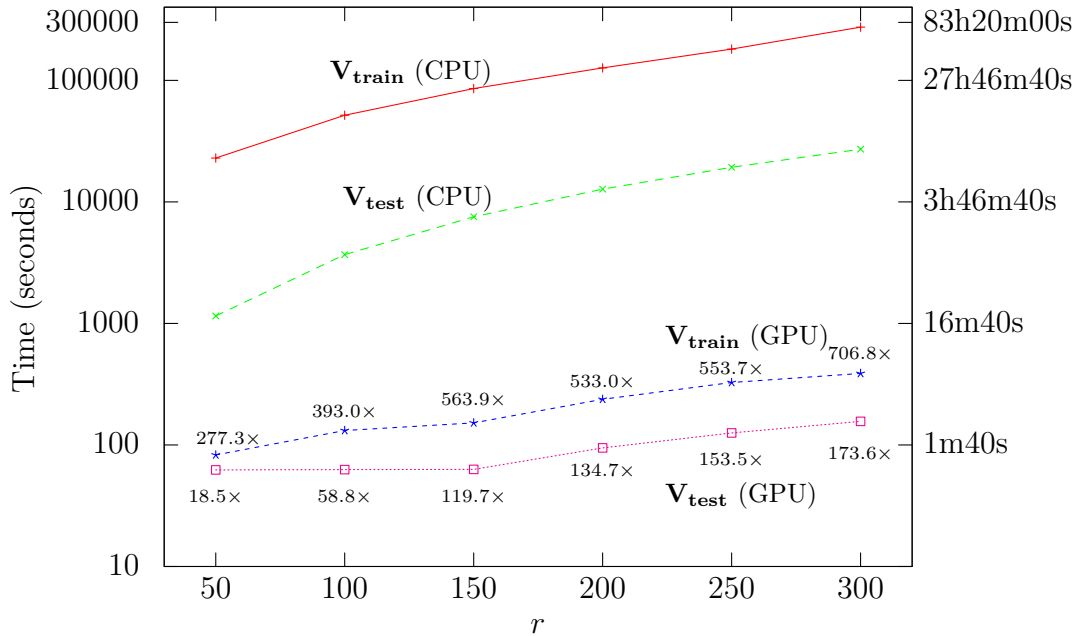
**Figure 5.13:** Time to perform 10,000 NMF iterations on the Yale database. The speedups ( $\times$ ) provided by the GPU are shown in the respective lines.

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  $\mathbf{H}_{\text{train}}$  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  $\mathbf{H}_{\text{test}}$  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  $\mathbf{H}_{\text{test}}$ , making the resulting model useless in many application scenarios.

After computing  $\mathbf{H}_{\text{train}}$  and  $\mathbf{H}_{\text{test}}$  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



**Figure 5.14:** Time to perform 10,000 NMF iterations on the AT&T (ORL) database. The speedups ( $\times$ ) provided by the GPU are shown in the respective lines.

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 5.1 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. [Tang et al., 2003]. 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 5.1) it still yields excellent results being only surpassed by the FRCM.

The results for the AT&T database are presented in Table 5.2. In this case, three of the methods (Fisherface, SVM and FRCM) perform better than the NMF-MBP

**Table 5.1:** Accuracy (%) results for the Yale dataset.

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

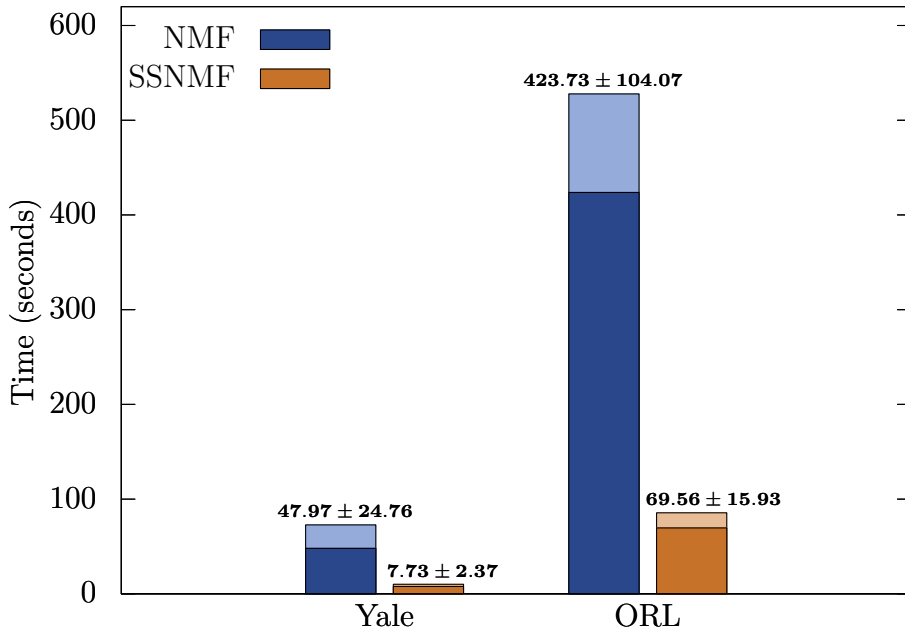
**Table 5.2:** Accuracy (%) results for the AT&T (ORL) dataset.

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

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 [Tang et al., 2003].

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





**Figure 5.15:** Time required to compute the  $\mathbf{W}$  and  $\mathbf{H}_{\text{train}}$  matrices.

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  $\mathbf{H}_{\text{train}}$ ). Figure 5.15 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 method should be greater in the CPU because the GPU scales better than the CPU when dealing with larger volumes of data (bigger matrices) [Lopes and Ribeiro, 2010b].

In terms of sparsity, it is obvious that the  $\mathbf{H}_{\text{train}}$  matrices generated by the SSNMF method will be sparser (by design) than the corresponding matrices produced by the unsupervised method. As for the  $\mathbf{H}_{\text{test}}$  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 5.3). 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 [Gillis and Glineur, 2010].

For the SVM algorithm the RBF (Gaussian) kernel was used and the  $C$  and  $\gamma$  parameters were chosen by cross-validation (using grid search). Figures 5.16 and 5.17 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  $\gamma$ ) are greater for the SSNMF method. This is better quantified in Table 5.4, which presents

**Table 5.3:** Percentage of zero values present in the  $\mathbf{H}_{\text{test}}$  matrix.

| Benchmark | NMF              | SSNMF            |
|-----------|------------------|------------------|
| Yale      | $36.70 \pm 8.39$ | $63.70 \pm 4.35$ |
| AT&T      | $45.20 \pm 2.14$ | $72.25 \pm 7.48$ |

**Table 5.4:** Grid search average accuracy on the test folds.

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

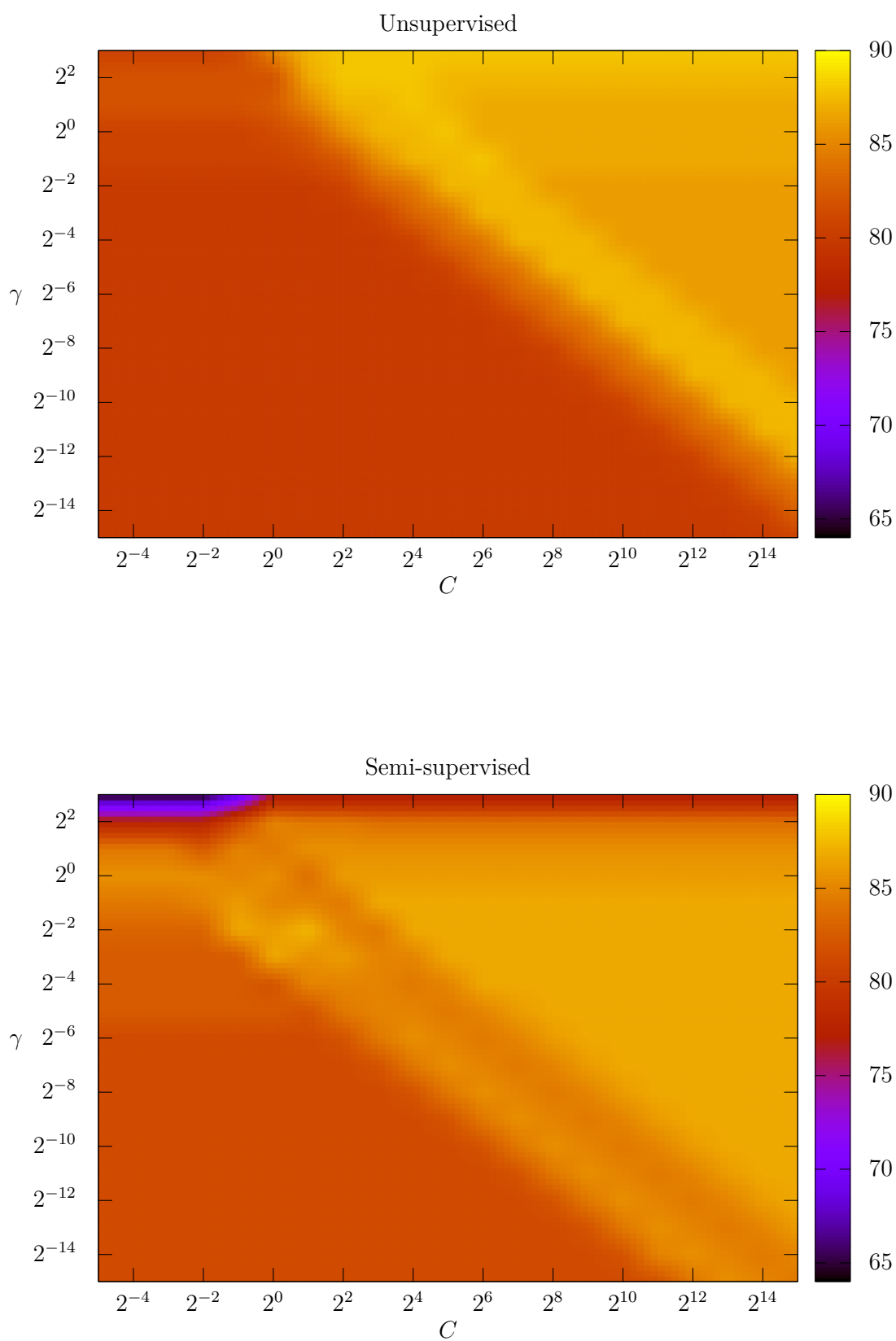
the average accuracy of the grid search test folds. While model selection by cross-validation is a good practice, the best performance may not always be obtained on unseen data. Therefore, the SSNMF method reduces the risk of creating unfitted/inadequate models.

Tables 5.5 and 5.6 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. [Tang et al., 2003], respectively for the Yale and for the AT&T databases.

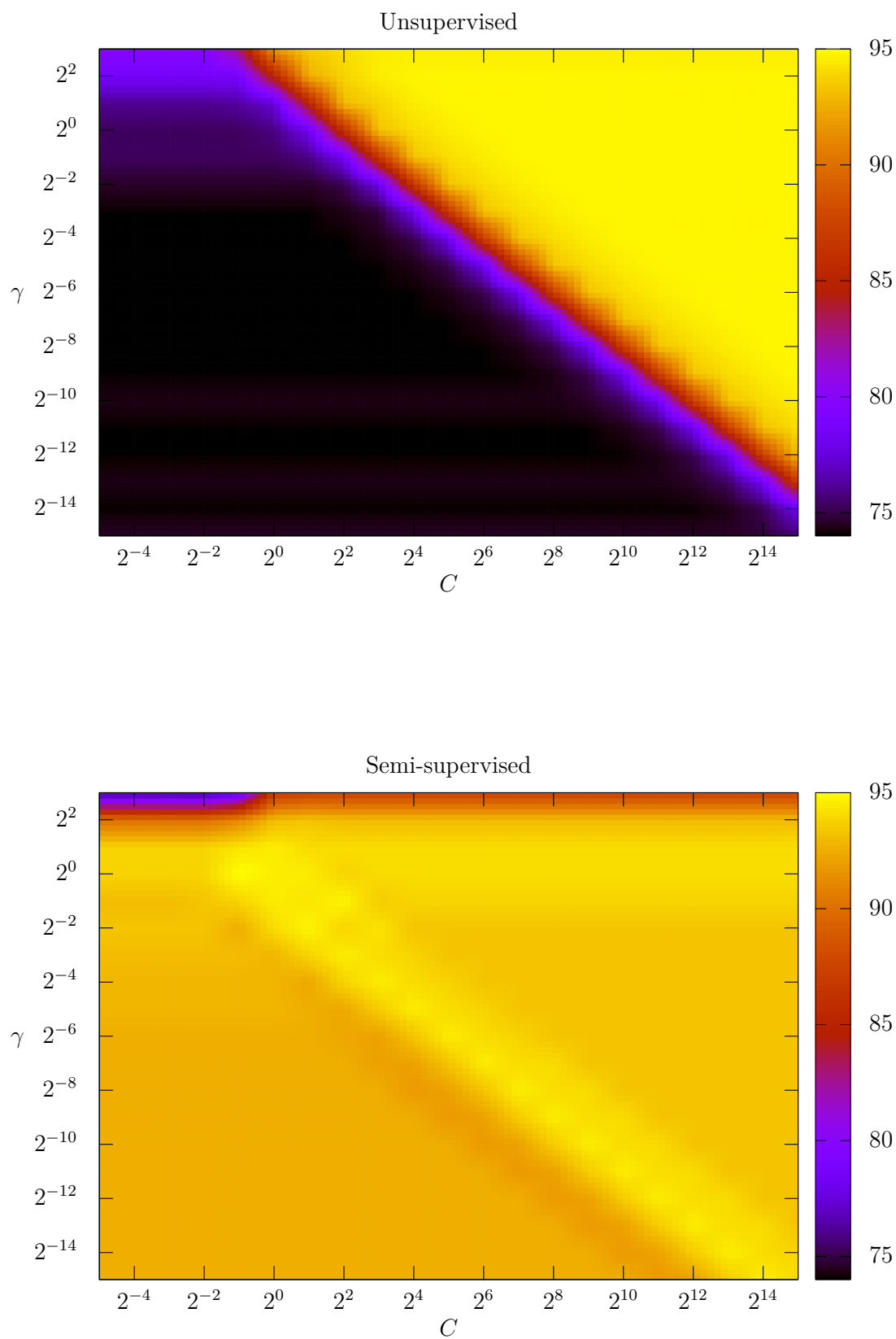
Considering the Yale database, the NMF based approaches (NMF-SVM, SSNMF-SVM and MBP-SVM) excel all the other methods in terms of accuracy. Overall, the best results were obtained by the NMF-MBP approach. As in the case of the NMF-

**Table 5.5:** Accuracy (%) results for the Yale dataset.

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



**Figure 5.16:** Average accuracy yielded by the SVM algorithm for the Yale dataset.



**Figure 5.17:** Average accuracy yielded by the SVM algorithm for the AT&T dataset.

**Table 5.6:** Accuracy results for the AT&T (ORL) dataset.

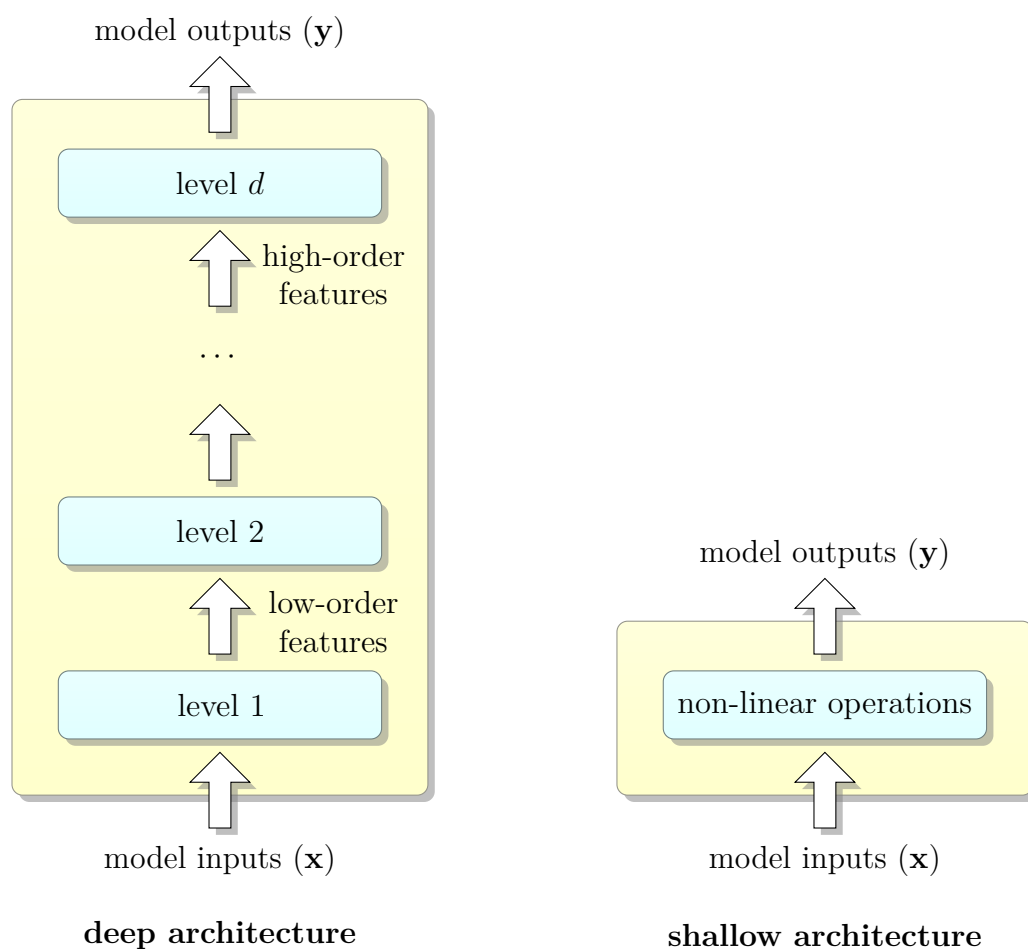
| Image Set      | NMF-SVM     | SSNMF-SVM    | NMF-MBP | Eigenface | Fisherface   | EGM  | EGM-SVM      | NN   | FRCM         |
|----------------|-------------|--------------|---------|-----------|--------------|------|--------------|------|--------------|
| 1              | 97.5        | 97.5     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

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 to perform better for facial image processing [Zhi et al., 2011]. 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 MBP-SVM). 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.

## 5.2 Deep Belief Networks (DBNs)

Recent empirical and theoretical advances in deep learning methods have led to a widespread enthusiasm in the pattern recognition and ML areas [Markoff, 2012, Larochelle et al., 2007]. 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 [Markoff, 2012]. In recent months deep learning applications have been growing both in number and accuracy [Markoff, 2012]. 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 [Markoff, 2012]. 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 was



**Figure 5.18:** Deep architectures versus shallow ones.

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 [Markoff, 2012].

Deep architecture models reflect the results of many levels of composition of non-linear operations in their outputs [Larochelle et al., 2007, Roux and Bengio, 2008, Bengio, 2009]. 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 [Lee et al., 2009]. 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 [Larochelle et al., 2007, Yu and Deng, 2011]. Figure 5.18 illustrates the main architectural differences between deep and shallow models.

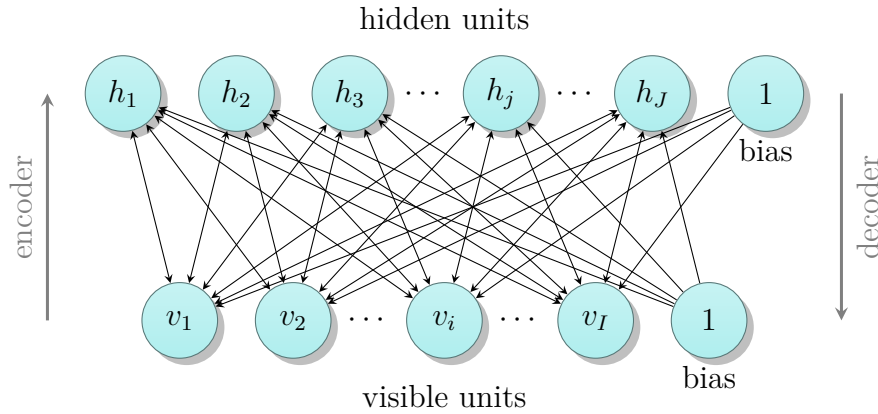
Deep architectures can be exponentially more efficient than shallow ones [Roux and Bengio, 2010]. For example, some functions can be compactly represented with an NN of depth  $l$ , while requiring an exponential number of computational elements and parameters for a network with depth  $l - 1$  [Bengio, 2009]. Shallow architectures may require a huge number of elements and, consequently, of training samples to represent highly varying functions [Roux and Bengio, 2008, Larochelle et al., 2007, Bengio, 2009]. A function is highly varying when a large number of pieces are required in order to create its piecewise approximation [Bengio, 2009]. On the other hand deep architectures can represent these functions efficiently, in particular when their Kolmogorov complexity is small [Larochelle et al., 2007].

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 [Bengio, 2009]. 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 [Bengio, 2009]. As a result, deep models tend to outperform shallow models such as SVMs [Larochelle et al., 2007]. 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) [Bengio, 2009], characterized by many factors of variation that interact in non-linear ways, making the learning process difficult [Larochelle et al., 2007].

However, the challenge of training deep multi-layer NNs remained elusive for a long time [Bengio, 2009], since traditional gradient-based optimization strategies are ineffective when propagated across multiple levels of non-linearities [Larochelle et al., 2007]. This changed with the development of DBNs [Hinton et al., 2006], 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 [Bengio, 2009, Roux and Bengio, 2008, Larochelle et al., 2007, Swersky et al., 2010, Yu and Deng, 2011], thus demonstrating its ability to outperform state-of-the-art algorithms [Bengio, 2009].

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 [Ranzato et al., 2007].

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.



**Figure 5.19:** Schematic representation of a Restricted Boltzmann Machine (RBM).

### 5.2.1 Restricted Boltzmann Machines (RBMs)




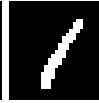

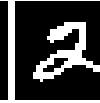

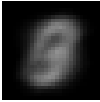

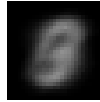
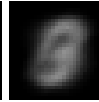
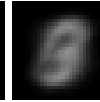
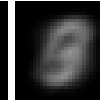
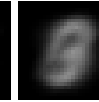

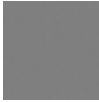





An RBM is an energy-based generative model that consists of a layer of  $I$  binary visible units (observed variables),  $\mathbf{v} = [v_1, v_2, \dots, v_I]$  where  $v_i \in \{0, 1\}$ , and a layer of  $J$  binary hidden units (explanatory factors),  $\mathbf{h} = [h_1, h_2, \dots, h_J]$  where  $h_j \in \{0, 1\}$ , with bidirectional weighted connections [Hinton, 2010], as depicted in Figure 5.19. 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 [Ranzato et al., 2007]. 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 [Ranzato et al., 2007].

Given an observed state, the energy of the joint configuration of the visible and hidden units ( $\mathbf{v}, \mathbf{h}$ ) is given by (5.14):

$$\begin{aligned} E(\mathbf{v}, \mathbf{h}) &= -\mathbf{c}\mathbf{v}^\top - \mathbf{b}\mathbf{h}^\top - \mathbf{h}\mathbf{W}\mathbf{v}^\top \\ &= -\sum_{i=1}^I c_i v_i - \sum_{j=1}^J b_j h_j - \sum_{j=1}^J \sum_{i=1}^I W_{ji} v_i h_j, \end{aligned} \quad (5.14)$$

where  $\mathbf{W} \in \mathbb{R}^{J \times I}$  is a matrix containing the RBM connection weights,  $\mathbf{c} = [c_1, c_2, \dots, c_I] \in \mathbb{R}^I$  is the bias of the visible units and  $\mathbf{b} = [b_1, b_2, \dots, b_J] \in \mathbb{R}^J$  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  $0.01$ ) [Hinton, 2010]. 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  $v_i = 1$  [Hinton, 2010]. Failure to do



|  |   |   |   |   |  |   |   |
|--|---|---|---|---|--|---|---|
| Original training images   |  |  |  |  |  |  |  |
| Initialization of $c_i = \log\left(\frac{\hat{p}_i}{1-\hat{p}_i}\right)$ |  |  |  |  |  |  |  |
| Initialization of $c_i$ at random  |  |  |  |  |  |  |  |

**Figure 5.20:** 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).

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  $\hat{p}_i$  [Hinton, 2010]. Figure 5.20 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.

The RBM assigns a probability for each configuration  $(\mathbf{v}, \mathbf{h})$ , using (5.15):

$$p(\mathbf{v}, \mathbf{h}) = \frac{e^{-E(\mathbf{v}, \mathbf{h})}}{Z}, \quad (5.15)$$

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 [Bengio, 2009, Hinton, 2010, Carreira-Perpiñán and Hinton, 2005]:

$$Z = \sum_{\mathbf{v}, \mathbf{h}} e^{-E(\mathbf{v}, \mathbf{h})}. \quad (5.16)$$

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:

$$p(\mathbf{h} | \mathbf{v}) = \prod_j p(h_j = 1 | \mathbf{v}), \quad (5.17)$$

where

$$p(h_j = 1 | \mathbf{v}) = \sigma\left(b_j + \sum_{i=1}^I v_i W_{ji}\right). \quad (5.18)$$

For implementation purposes,  $h_j$  is set to 1 when  $p(h_j = 1 | \mathbf{v})$  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 (5.19):

$$p(\mathbf{v} | \mathbf{h}) = \prod_i p(v_i = 1 | \mathbf{h}) , \quad (5.19)$$

where:

$$p(v_i = 1 | \mathbf{h}) = \sigma(c_i + \sum_{j=1}^J h_j W_{ji}) . \quad (5.20)$$

When using (5.20) 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 [Hinton, 2010].

The marginal probability assigned to a visible vector,  $\mathbf{v}$ , is given by (5.21):

$$p(\mathbf{v}) = \sum_{\mathbf{h}} p(\mathbf{v}, \mathbf{h}) = \frac{1}{Z} \sum_{\mathbf{h}} e^{-E(\mathbf{v}, \mathbf{h})} . \quad (5.21)$$

Hence, given a specific training vector  $\mathbf{v}$  its probability can be raised by adjusting (optimizing) 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<sup>1</sup>, 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 (5.22):

$$\begin{aligned} \frac{\partial \log p(\mathbf{v})}{\partial \theta} &= \frac{\partial \log \left( \frac{1}{Z} \sum_{\mathbf{h}} e^{-E(\mathbf{v}, \mathbf{h})} \right)}{\partial \theta} = \overbrace{\frac{\partial \log \left( \sum_{\mathbf{h}} e^{-E(\mathbf{v}, \mathbf{h})} \right)}{\partial \theta}}^{\text{positive phase}} \quad \overbrace{- \frac{\partial \log Z}{\partial \theta}}^{\text{negative phase}} \\ &= \frac{\partial \log \left( \sum_{\mathbf{h}} e^{-E(\mathbf{v}, \mathbf{h})} \right)}{\partial \theta} - \frac{\partial \log \left( \sum_{\mathbf{v}, \mathbf{h}} e^{-E(\mathbf{v}, \mathbf{h})} \right)}{\partial \theta} \\ &= - \frac{\sum_{\mathbf{h}} e^{-E(\mathbf{v}, \mathbf{h})} \frac{\partial E(\mathbf{v}, \mathbf{h})}{\partial \theta}}{\sum_{\mathbf{h}} e^{-E(\mathbf{v}, \mathbf{h})}} + \frac{\sum_{\mathbf{v}, \mathbf{h}} e^{-E(\mathbf{v}, \mathbf{h})} \frac{\partial E(\mathbf{v}, \mathbf{h})}{\partial \theta}}{\sum_{\mathbf{v}, \mathbf{h}} e^{-E(\mathbf{v}, \mathbf{h})}} \end{aligned} \quad (5.22)$$

---

<sup>1</sup> 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 [Bishop, 2006].

Using (5.15) we can write  $p(\mathbf{h} \mid \mathbf{v})$  as (5.23):

$$\begin{aligned}
 p(\mathbf{h} \mid \mathbf{v}) &= \frac{p(\mathbf{h}, \mathbf{v})}{p(\mathbf{v})} \\
 &= \frac{p(\mathbf{h}, \mathbf{v})}{\sum_{\mathbf{h}} p(\mathbf{h}, \mathbf{v})} \\
 &= \frac{1}{\sum_{\mathbf{h}} \frac{e^{-E(\mathbf{v}, \mathbf{h})}}{Z}} \frac{e^{-E(\mathbf{v}, \mathbf{h})}}{Z} \\
 &= \frac{e^{-E(\mathbf{v}, \mathbf{h})}}{\sum_{\mathbf{h}} e^{-E(\mathbf{v}, \mathbf{h})}}
 \end{aligned} \tag{5.23}$$

Hence, we can rewrite (5.22) as (5.24):

$$\frac{\partial \log p(\mathbf{v})}{\partial \theta} = \underbrace{-\sum_{\mathbf{h}} p(\mathbf{h} \mid \mathbf{v}) \frac{\partial E(\mathbf{v}, \mathbf{h})}{\partial \theta}}_{\text{positive phase}} + \underbrace{\sum_{\mathbf{v}, \mathbf{h}} p(\mathbf{v}, \mathbf{h}) \frac{\partial E(\mathbf{v}, \mathbf{h})}{\partial \theta}}_{\text{negative phase}} \tag{5.24}$$

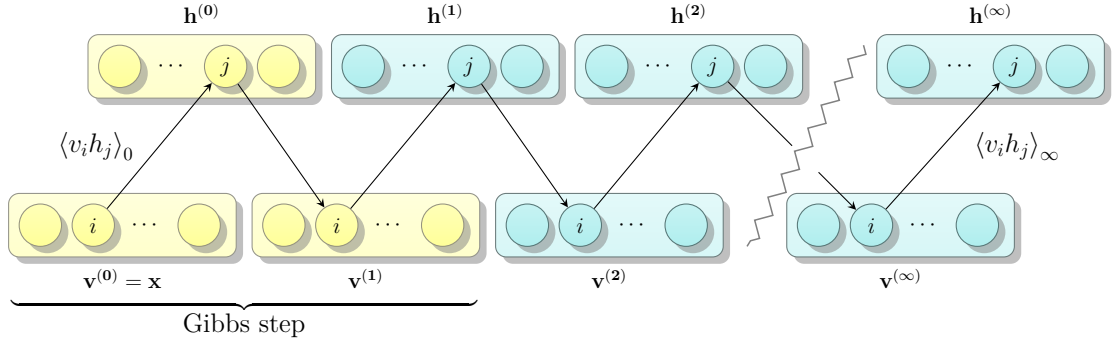
As in the maximum likelihood learning procedure, we aim at finding the set of network parameters for which the probability of the (observed) training dataset is maximized. Computing  $\frac{\partial E(\mathbf{v}, \mathbf{h})}{\partial \theta}$  is straightforward. Thus, in order to obtain an unbiased 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  $p(\mathbf{v}, \mathbf{h})$  [Bengio, 2009]. 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 [Bengio, 2009].

Sampling can be accomplished by setting up a Markov Chain Monte Carlo (MCMC) using alternating Gibbs sampling [Hinton, 2010, Bengio, 2009]. Each iteration of Gibbs sampling consists of updating all of the hidden units in parallel using (5.18) followed by updating all of the visible units in parallel using (5.20) [Hinton, 2010]. This process is represented in Figure 5.21.

Using this procedure we can rewrite (5.24) as (5.25):

$$\frac{\partial \log p(\mathbf{v})}{\partial \theta} = \underbrace{-\left\langle \frac{\partial E(\mathbf{v}, \mathbf{h})}{\partial \theta} \right\rangle_0}_{\text{positive phase}} + \underbrace{\left\langle \frac{\partial E(\mathbf{v}, \mathbf{h})}{\partial \theta} \right\rangle_{\infty}}_{\text{negative phase}} \tag{5.25}$$

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 ( $p_{\infty}(\mathbf{v}, \mathbf{h}) = p(\mathbf{v}, \mathbf{h})$ ) [Roux and Bengio, 2008, Carreira-Perpiñán and Hinton, 2005]. It makes sense to start the chain with a training sample,  $\mathbf{x}$ , because eventually the model distribution,  $p_{\infty}$ , and the data distribution,  $p_0$ , will become similar as the model gradually captures the statistical structure embedded in the training data [Bengio, 2009].



**Figure 5.21:** 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.

Unfortunately, computing  $\langle v_i h_j \rangle_\infty$  is intractable as it requires performing alternating Gibbs sampling for a very long time [Hinton, 2010, Bengio, 2009] in order to draw unbiased samples from the model distribution to generate a good gradient approximation [Swersky et al., 2010]. 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 [Bengio, 2009].

To solve this problem, Hinton proposed a much faster learning procedure: the Contrastive Divergence (CD- $k$ ) algorithm [Hinton, 2002, Hinton, 2010], whereby  $\langle \cdot \rangle_\infty$  is replaced by  $\langle \cdot \rangle_k$  for small values of  $k$  [Roux and Bengio, 2008]. 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 [Hinton, 2002, Carreira-Perpiñán and Hinton, 2005]. Changing (5.25) accordingly, we obtain the following update rule for the weights of the network:

$$\Delta W_{ji} = \eta \left( \overbrace{\langle v_i h_j \rangle_0}^{\text{positive phase}} - \underbrace{\langle v_i h_j \rangle_k}_{\text{negative phase}} \right) \quad (5.26)$$

where  $\eta$  represents the learning rate. Similarly, the following rules allow us to update the bias of the network:

$$\Delta b_j = \eta (\langle h_j \rangle_0 - \langle h_j \rangle_k) \quad (5.27)$$

$$\Delta c_i = \eta (\langle v_i \rangle_0 - \langle v_i \rangle_k) \quad (5.28)$$

Algorithm 3 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 (5.19)) for computing the state of the hidden units, instead of using the corresponding stochastic binary states which would cause unnecessary sampling noise [Hinton, 2010].

---

**Algorithm 3** CD- $k$  algorithm.

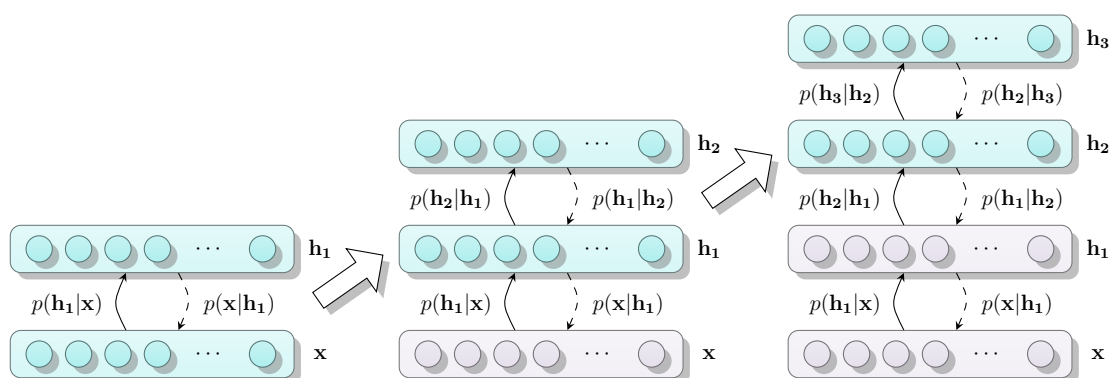
---

- 1:  $\mathbf{v}^{(0)} \leftarrow \mathbf{x}$   $\triangleright \mathbf{x}$  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. 5.18
  - 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. 5.20
  - 5:   Compute the binary features (states) for the hidden units,  $\mathbf{h}^{(n)}$ , using  $\mathbf{v}^{(n)}$  and eq. 5.18
  - 6: **end for**
  - 7: Update the weights and biases, using eq. 5.26, 5.27 and 5.28
- 

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 [Hinton, 2010]. In fact, its learning rule is actually (approximately) following an objective function which is called Contrastive Divergence and is given by the difference of two Kullback-Liebler divergences [Hinton, 2010, Hinton, 2002]. Nevertheless, although 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 [Tieleman, 2008, Bengio, 2009]. In particular, CD-1 provides a low variance, fast and reasonable approximation of the log-likelihood gradient [Tieleman, 2008], which has been empirically demonstrated to yield good results [Bengio, 2009].

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 [Hinton, 2010]. 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 [Hinton, 2010]. 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 [Swersky et al., 2010].

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 [Lee et al., 2009].



**Figure 5.22:** 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, purple color represents layers already trained, while cyan represents the Restricted Boltzmann Machine (RBM) being trained.

### 5.2.2 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 [Hinton et al., 2006]. 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 [Lee et al., 2009].

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 of learning (an otherwise complicated model) by combining multiple and simpler (RBM) models, learned sequentially [Hinton et al., 2006]. Figure 5.22 represents this process.

The number of layers of a DBN can be increased in a greedy manner [Larochelle et al., 2007]. Each new layer that is stacked on top of the DBN will model the output of the previous layer [Larochelle et al., 2007] 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 [Ranzato et al., 2007, Swersky et al., 2010]. 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 [Roux and Bengio, 2008, Swersky et al., 2010, Roux and Bengio, 2010].

The training process, also called pre-training [Larochelle et al., 2007], is unsupervised by nature, allowing the system to learn non-linear complex mapping functions directly from data, without depending on human-crafted features [Bengio,

2009]. However, the output of the top layer can easily be fed to a conventional supervised classifier [Hinton, 2010, Ranzato et al., 2007]. 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 [Swersky et al., 2010]. 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 [Swersky et al., 2010].

### 5.2.3 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 [Lopes and Ribeiro, 2012b]. In particular the learning rate is highly correlated with the training speed and convergence [Schulz et al., 2010]. 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 4.1.1 (see page 68), to the RBM networks. Hence, at each CD- $k$  iteration, the step sizes are adjusted according to the sign changes:

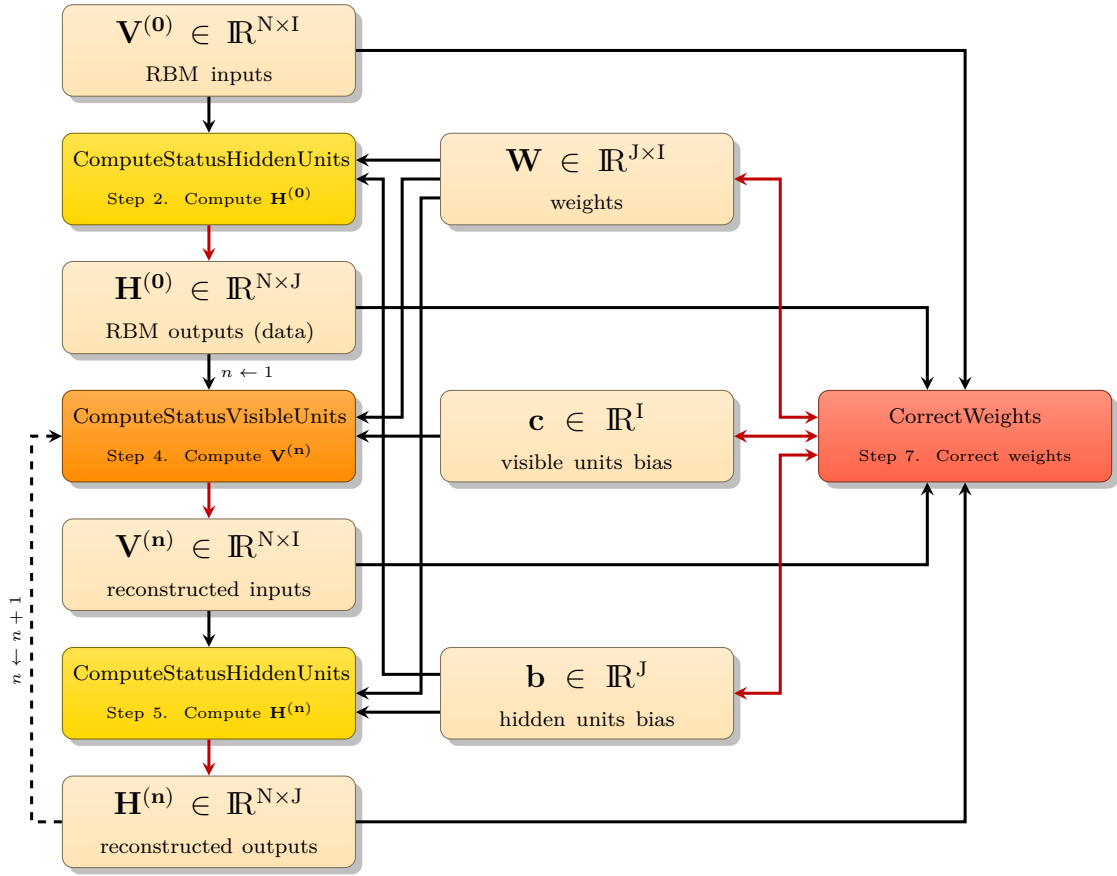
$$\eta_{ji} = \begin{cases} u\eta_{ji}^{(\text{old})} & \text{if } (\langle v_i h_j \rangle_0 - \langle v_i h_j \rangle_k)(\langle v_i h_j \rangle_0^{(\text{old})} - \langle v_i h_j \rangle_k^{(\text{old})}) > 0 \\ d\eta_{ji}^{(\text{old})} & \text{if } (\langle v_i h_j \rangle_0 - \langle v_i h_j \rangle_k)(\langle v_i h_j \rangle_0^{(\text{old})} - \langle v_i h_j \rangle_k^{(\text{old})}) < 0 \end{cases} \quad (5.29)$$

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 [Lopes and Ribeiro, 2012b]. 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 [Almeida, 1997].

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,  $\alpha$ , 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  $\alpha_{ji}$ , such that  $0.1 \leq \alpha_{ji} \leq 0.9$ .

### 5.2.4 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

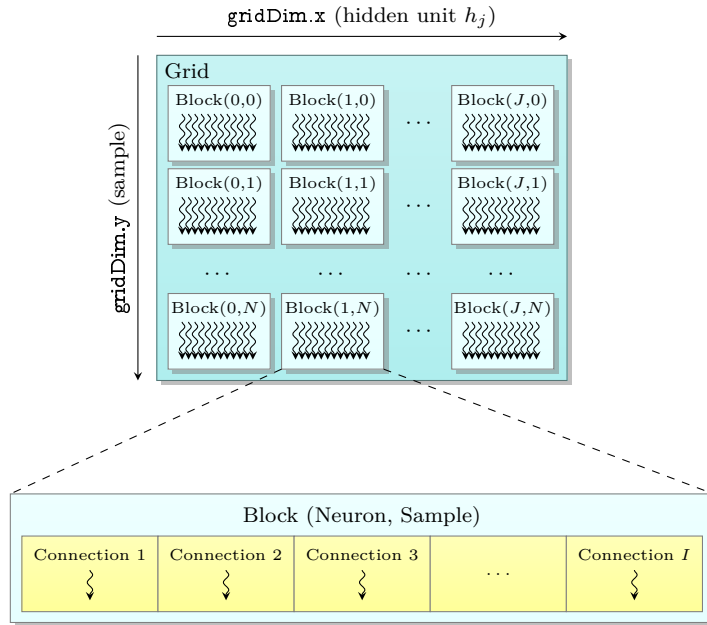


**Figure 5.23:** Sequence of GPU kernel calls, per epoch, that implement the CD- $k$  algorithm.

vectors,  $\{\mathbf{v}_1, \mathbf{v}_2, \dots, \mathbf{v}_N\}$ , 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,  $\{\mathbf{h}_1, \mathbf{h}_2, \dots, \mathbf{h}_N\}$ , 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 3 (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 5.23 shows the sequence of kernel calls (per epoch) needed to implement the CD- $k$  algorithm.

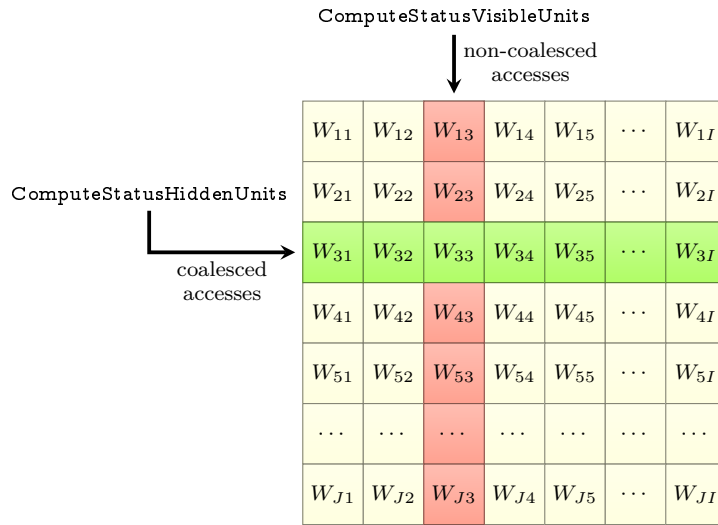




**Figure 5.24:** ComputeStatusHiddenUnits kernel grid and block structure.

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 4.1.3, page 76). 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 5.24. 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.

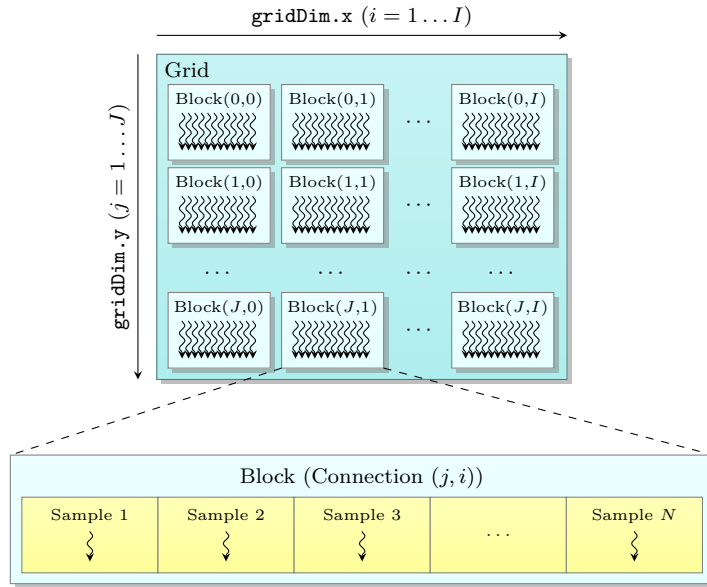
In practice, the number of connections of each neuron (defined either by the number of inputs or by the number of hidden units of a RBM, respectively for the `ComputeStatusHiddenUnits` and `ComputeStatusVisibleUnits` kernels) can easily surpass the limits of the maximum number of threads per block imposed by CUDA. As a consequence each thread may actually need to compute the output of several connections [Lopes et al., 2012b]. Thus, the actual number of



**Figure 5.25:** Implications of storing the connection weights using row-major order.

threads being executed in each call of `ComputeStatusHiddenUnits` will be equal to  $\max(N, 65535) \times J \times \max(I, 1024)$ , 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  $\max(N, 65535) \times I \times \max(J, 1024)$ . 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  $28 \times 28$  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  $\mathbf{W}$  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 5.23 and/or Algorithm 3), we decided to store  $\mathbf{W}$  in a row-major order, thus improving its performance in detriment of the `ComputeStatusVisibleUnits` kernel [Lopes et al., 2012b]. Figure 5.25 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

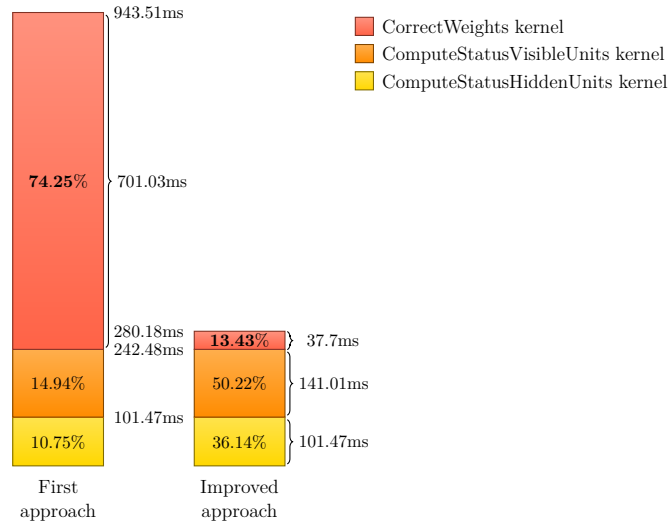


**Figure 5.26:** Grid and block structure used by the first approach of the kernel `CorrectWeights`.

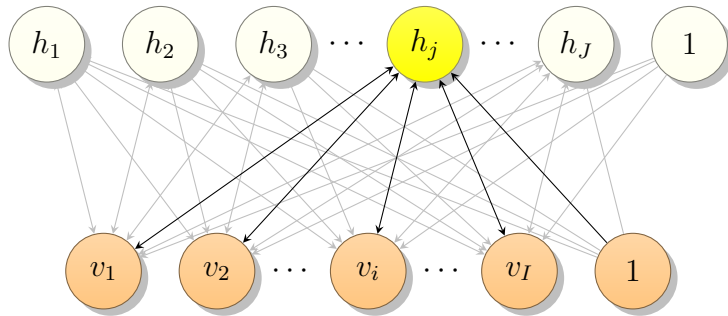
a coalesced manner, however the `ComputeStatusHiddenUnits` must access the weights in a non-coalesced manner.

The bulk work to be carried out by the `CorrectWeights` kernel consists of aggregating the values for  $\Delta W_{ji}$ ,  $\Delta b_j$  and  $\Delta c_i$  (see respectively (5.26), (5.27) and (5.28)), 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 5.26 illustrates the resulting grid and block structure.

In order to evaluate this first GPU implementation, we conducted preliminary tests using the MNIST dataset (described earlier in Section 3.4, page 49). The left column of Figure 5.27 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 5.2.5). Note that despite `ComputeStatusHiddenUnits` being called twice (see Figure 5.23 and/or Algorithm 3) 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.



**Figure 5.27:** Proportion of time spent, per epoch, in each kernel, as measured in the computer System 2 (with a GTX 280).



**Figure 5.28:** Connections to the hidden unit  $j$ .

We identify two main problems in the first approach of the kernel `CorrectWeights`, both related to memory accesses to the  $\mathbf{V}^{(0)}$ ,  $\mathbf{H}^{(0)}$ ,  $\mathbf{V}^{(n)}$  and  $\mathbf{H}^{(n)}$  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 [Lopes et al., 2012b]. Figure 5.28 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  $\mathbf{H}^{(0)}$  and  $\mathbf{H}^{(n)}$ . 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 5.29). Thus, they all need to access the same elements of  $\mathbf{V}^{(0)}$  and  $\mathbf{V}^{(n)}$ .

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

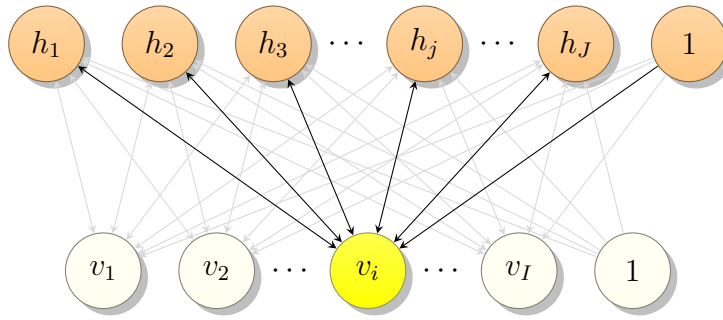
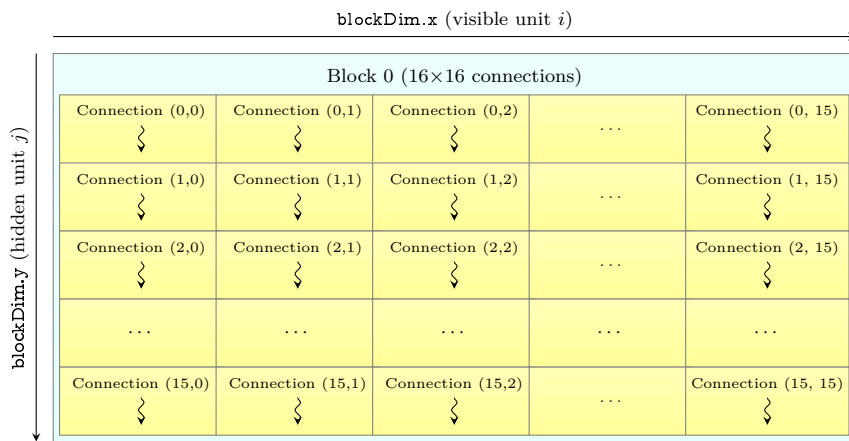
Figure 5.29: Connections to the visible unit  $i$ .

Figure 5.30: Block structure of the improved approach of the CorrectWeights kernel.

block processes several adjacent connections that require, to some degree, accessing the same elements of  $\mathbf{V}^{(0)}$ ,  $\mathbf{H}^{(0)}$ ,  $\mathbf{V}^{(n)}$  and  $\mathbf{H}^{(n)}$ . Figure 5.30 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, the block starts by copying the portions of  $\mathbf{V}^{(0)}$ ,  $\mathbf{H}^{(0)}$ ,  $\mathbf{V}^{(n)}$  and  $\mathbf{H}^{(n)}$ , 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  $\mathbf{V}^{(0)}$  and  $\mathbf{V}^{(n)}$ . 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  $\mathbf{H}^{(0)}$  and  $\mathbf{H}^{(n)}$ . Moreover, since the required portions of the matrices  $\mathbf{V}^{(0)}$ ,  $\mathbf{H}^{(0)}$ ,  $\mathbf{V}^{(n)}$  and  $\mathbf{H}^{(n)}$  are gathered for the same sample, the global memory accesses are now coalesced.

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 5.27). 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.

## 5.2.5 Results and Discussion

### Experimental Setup

In our testbed experiments we have used two datasets: the MNIST database of hand-written digits (see Section 3.4, page 49) and the HHreco multi-stroke symbol database (see Section 3.4, page 46). Altogether, three different experiments were conducted.

First, an experiment for evaluating the performance of the multi-core GPU parallel implementation was carried out. Considering that both the resulting datasets have an equal number of inputs, the tests for evaluating the multi-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 3.1, page 38).

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 3.1, page 38). 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 3.1, page 38). Moreover, the final training step of the DBNs was made using the GPU implementation of the BP and MBP algorithms, described earlier in Section 4.1.3, page 75).

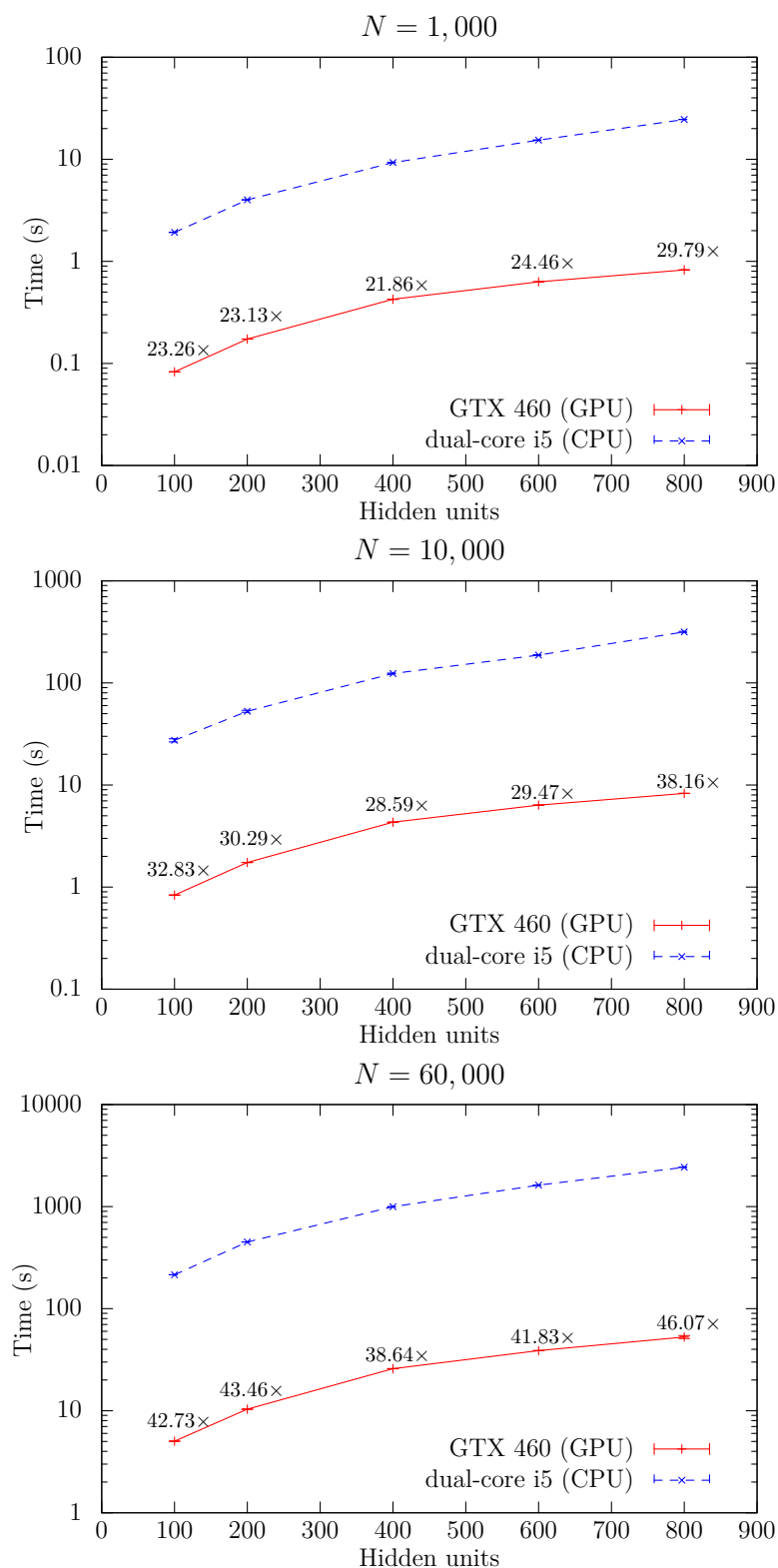
## Benchmarks Results

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 5.31 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 [Lopes et al., 2012b]. 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.

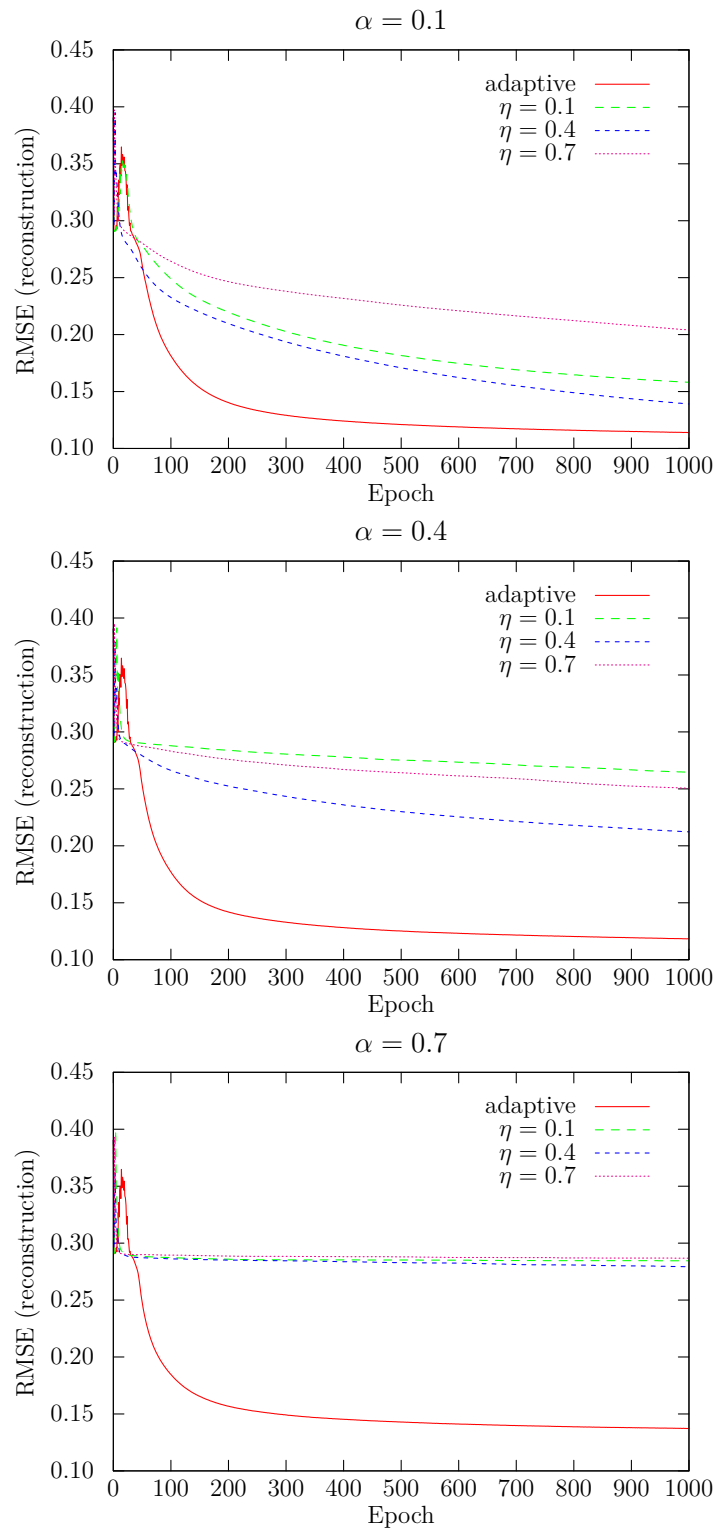
In order to evaluate the impact of the adaptive step size technique, we have compared it with three different fixed learning rate settings ( $\eta = 0.1$ ,  $\eta = 0.4$  and  $\eta = 0.7$ ), while using three distinct momentum terms ( $\alpha = 0.1$ ,  $\alpha = 0.4$  and  $\alpha = 0.7$ ). For the 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 5.32 shows the evolution of the RMSE of the reconstruction, according to the learning rate,  $\eta$ , and momentum,  $\alpha$ , 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 [Lopes and Ribeiro, 2012b]. 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  $\alpha = 0.7$  [Lopes and Ribeiro, 2012b]. Figure 5.33 shows the quality of the reconstruction of the original images in the database, for both the

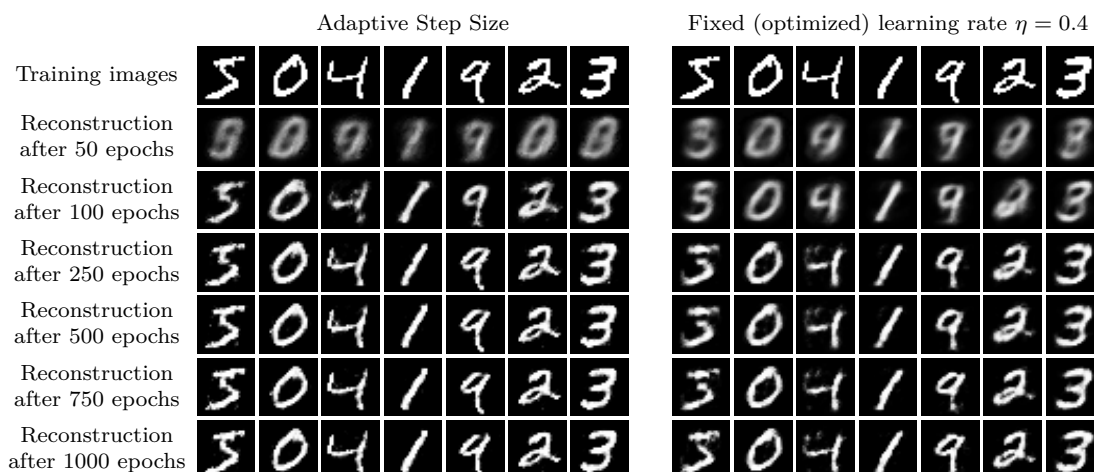


**Figure 5.31:** MNIST average training time per epoch (GPU speedups are indicated).





**Figure 5.32:** Average reconstruction error (RMSE) according to the learning parameters,  $\eta$  and  $\alpha$ .



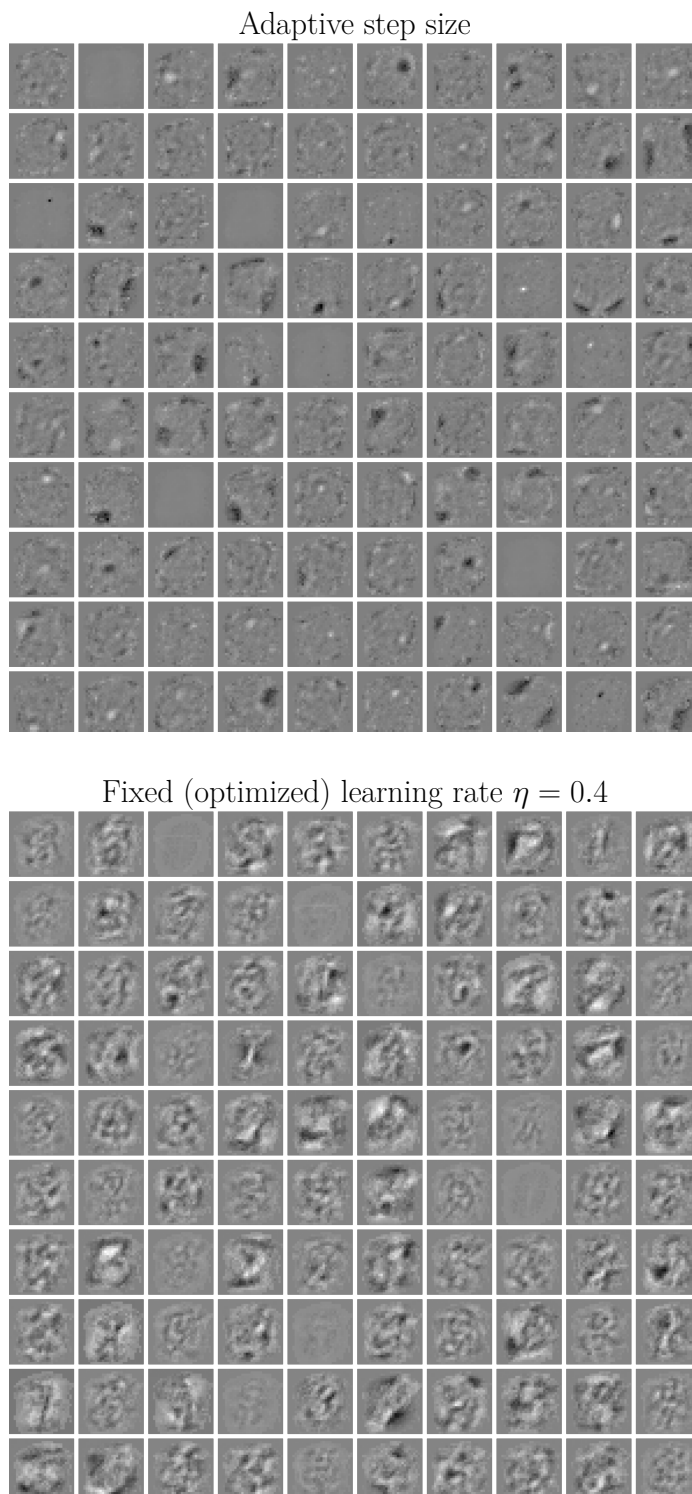
**Figure 5.33:** Impact of the step size technique on the convergence of a RBM ( $\alpha = 0.1$ ).

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, Figure 5.34 shows the receptive fields of the aforementioned networks and Figure 5.35 their excitatory and inhibitory response zones.

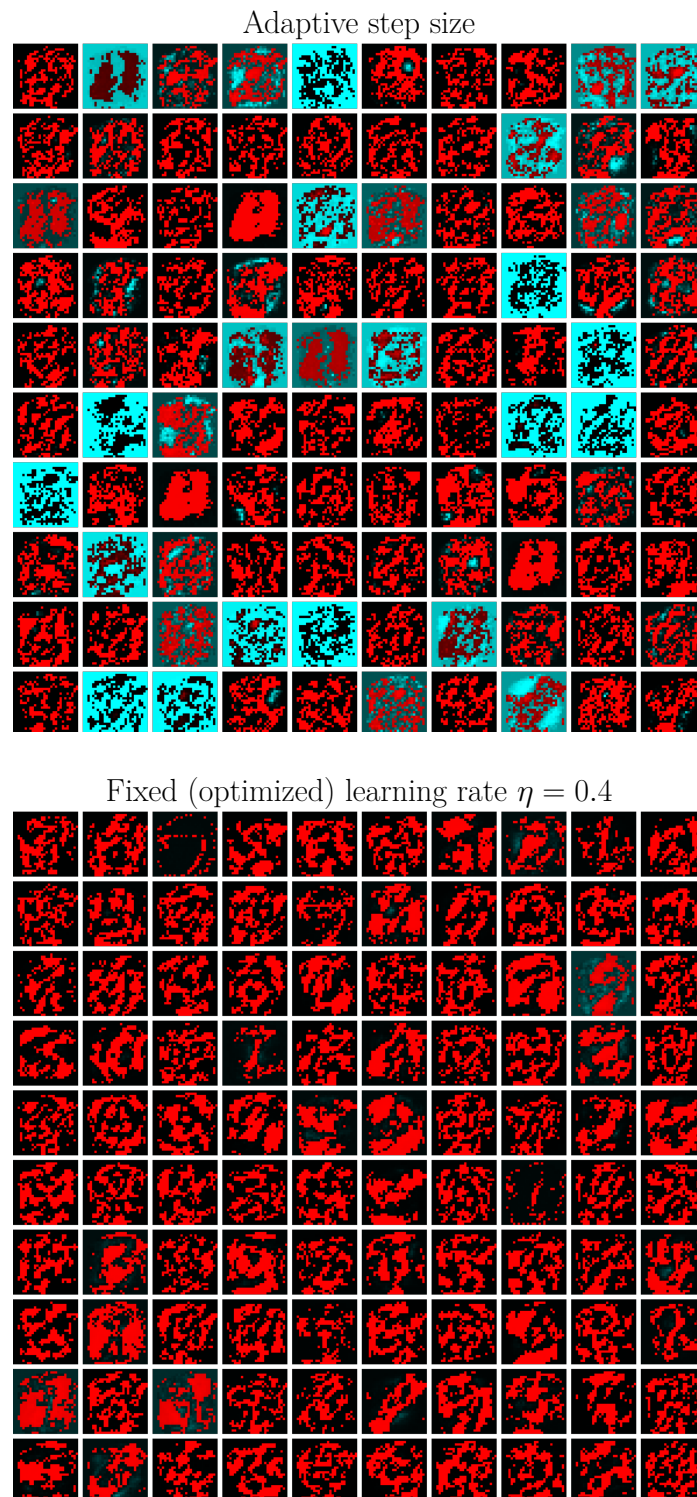
Training a network for 1,000 epochs using the adaptive step size took on average  $76.63 \pm 0.09$  seconds, while training the same network with a fixed learning rate took on average  $76.12 \pm 0.05$  seconds. Thus, the overhead of this method is not significant, while the convergence of the network is considerably enhanced. Additionally, the adaptive step size technique solves the difficulty of searching and choosing an adequate learning rate,  $\eta$ , and momentum,  $\alpha$  terms. Moreover, the step size method can easily recover from a bad choice of the initial learning rate [Almeida, 1997] and the parameters  $u$  and  $d$  are easily tuned (the values chosen for this problem will most likely yield good results for other problems) [Lopes and Ribeiro, 2012b].

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).



**Figure 5.34:** Receptive fields of the best networks trained either with the adaptive step size or with a fixed learning rate.



**Figure 5.35:** 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.

**Table 5.7:** Top 10 DBNs with the best classification performance for the MNIST dataset. The topology column refers to the topology of the added classification layers.

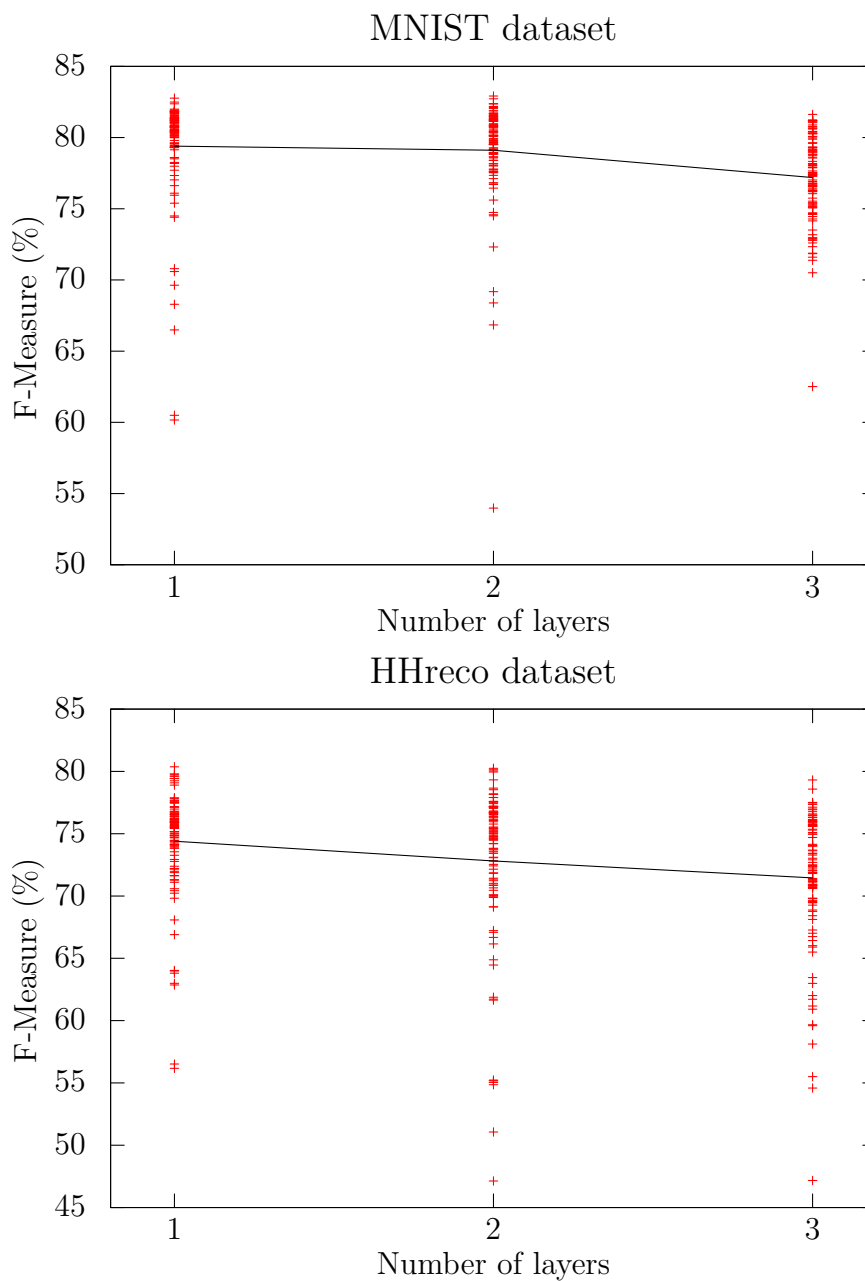
| Topology | Pre-trained DBN layers | DBN Layers          | F-Measure |
|----------|------------------------|---------------------|-----------|
| MBP      | 784-1000-1000          | 784-1000-1000-30-10 | 82.92     |
| MBP      | 784-500                | 784-500-30-10       | 82.77     |
| MBP      | 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     |

Moreover, we have also tested the effects of adding MBP layers (see Section 4.1), 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 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 5.36 shows the classification performance of the resulting models, according to the number of layers of the pre-trained DBNs. Moreover, Tables 5.7 and 5.8 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 5.36). Nevertheless, in the case of the MNIST dataset, most of the best DBNs contain four layers, not including the input layer (see Table 5.7). 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 5.8).



**Figure 5.36:** DBNs classification performance, according to the number of pre-training layers.

**Table 5.8:** 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     |

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 [Lopes and Ribeiro, 2013]. 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 a modular system, it is possible to add new layers, increasing the network depth and test whether the new features improve the overall system [Lopes and Ribeiro, 2013]. 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 5.7 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

**Table 5.9:** Confusion matrix of the best MNIST DBN (trained with 60,000 samples).

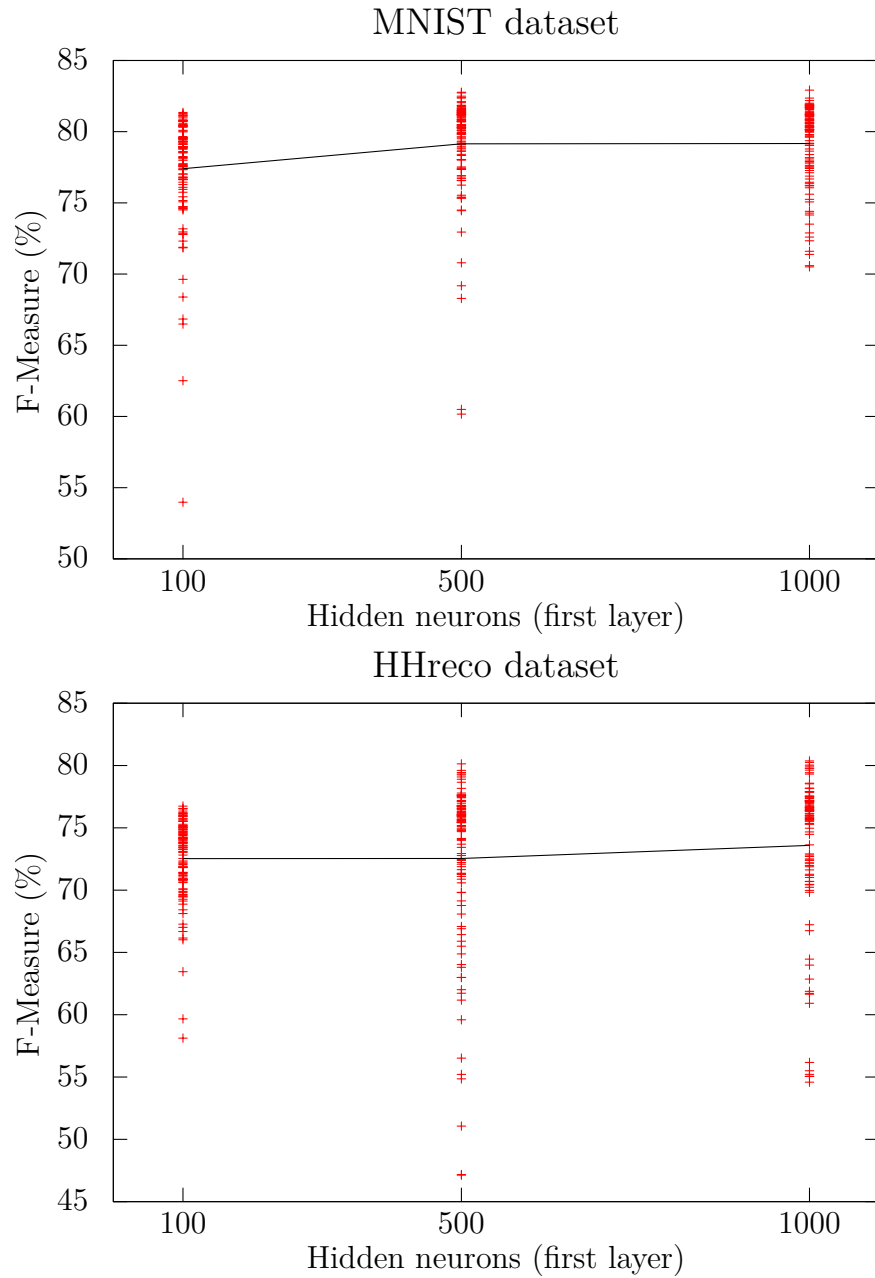
| actual | Predicted class |      |     |     |     |     |     |     |     |     |
|--------|-----------------|------|-----|-----|-----|-----|-----|-----|-----|-----|
|        | 0               | 1    | 2   | 3   | 4   | 5   | 6   | 7   | 8   | 9   |
| 0      | 959             | 0    | 1   | 2   | 1   | 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 |

epochs for approximately 3:41 hours. Note that the pre-trained weights were frozen. Table 5.9 presents the confusion matrix of this network.

It is important to point out that the classification performance of the networks presented in Table 5.7 (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 5.37 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 5.7 and 5.8, 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) and the remaining networks presented in Tables 5.7 and 5.8. 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 [Lopes and Ribeiro, 2013].





**Figure 5.37:** DBNs classification performance, according to the number of neurons in the first hidden layer.

Figure 5.38 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 5.8 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 [Lopes and Ribeiro, 2013].

Figure 5.39 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 [Lopes and Ribeiro, 2013].

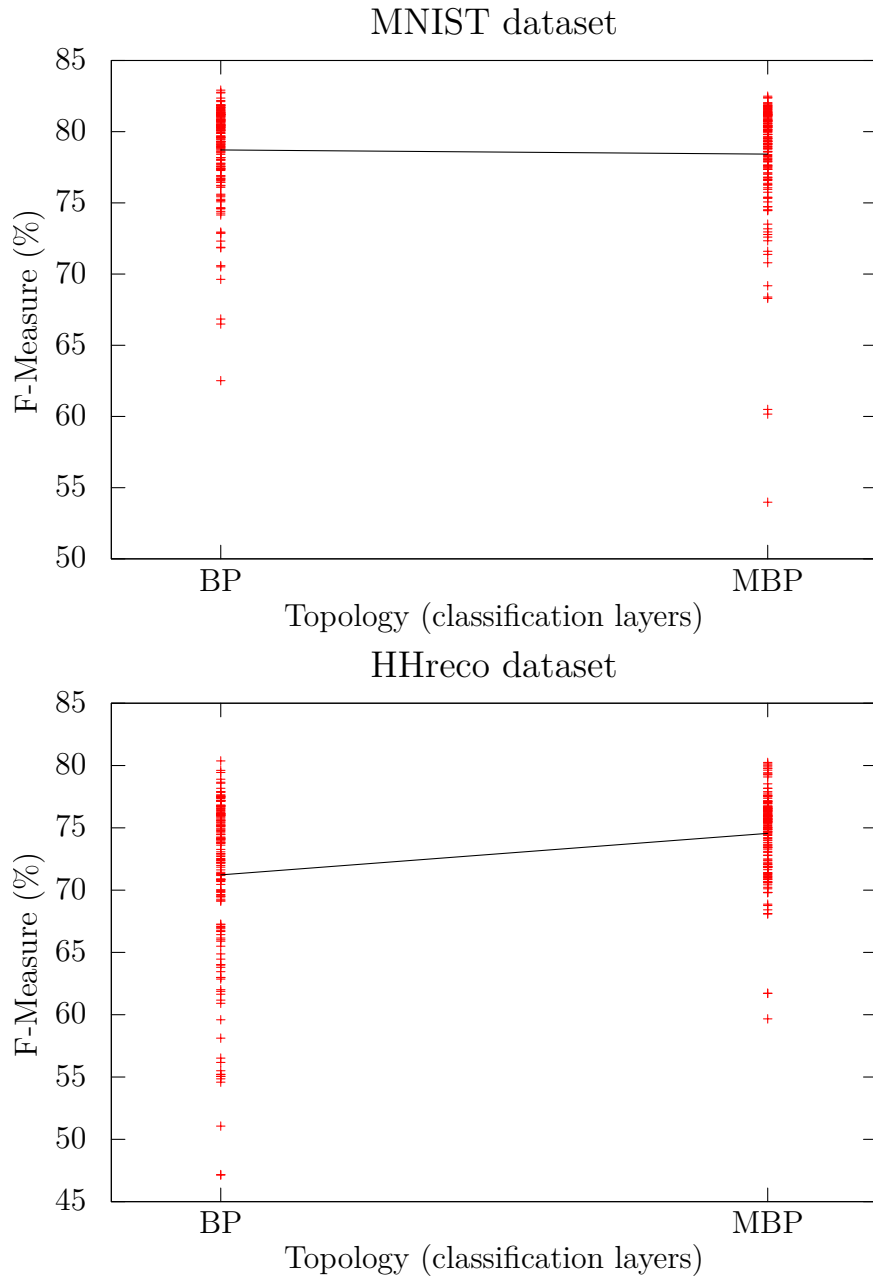
### 5.3 Summary

In this Chapter, we have addressed two state-of-the-art of the art unsupervised learning architectures. These (NMF and DBN) represent two different and powerful approaches for extracting discriminative features directly from raw-data, thus creating useful representations that can be fed into a supervised algorithm in order to create improved models (by comparison with those derived from the original data representations).

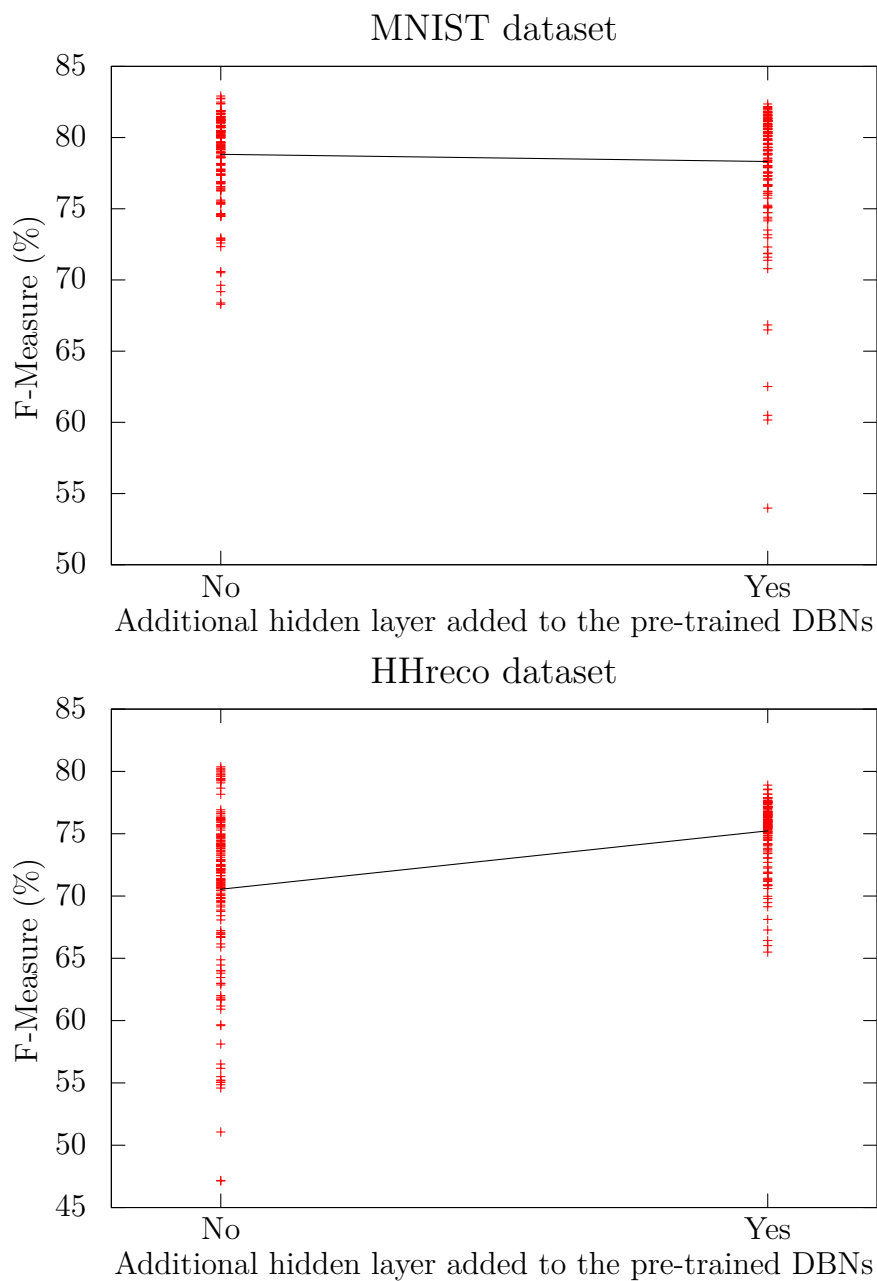
Typically, NMF is used to reduce the data dimensionality, while preserving the information of the most relevant features in order to rebuild accurate approximations of the original data, through additive combinations of a factorized parts-based matrix. The rationale is to take advantage of the data redundancies in order to extract the essential knowledge embedded within the data while discarding the random noise that it may contain.

DBNs take a different approach by using a generative deep architecture model in which more complex and sophisticated features are progressively extracted from the original raw input signals (as we move deeper through the model layers). DBNs are built by stacking several RBMs on top of each other and therefore allow for overcomplete representations.

Despite their attractiveness, building either one of the aforementioned models (NMF and DBN) is a computationally expensive and time consuming task, especially when large volumes of data are considered. Accordingly, we have explored several strategies to speedup the process of creating these models. In this context, we



**Figure 5.38:** DBNs classification performance, according to the topology of the additional layers (added to the pre-trained networks).



**Figure 5.39:** DBNs classification performance, depending on whether or not an additional hidden layer was added to the pre-trained networks.

have successfully mapped both the NMF and the DBNs (or more precisely of the CD- $k$ ) algorithms to the GPU. The resulting parallel implementations provide remarkable performance gains relatively to the corresponding standalone CPU baseline implementations. Hence, they are of considerable value as they make possible the development of useful ML applications that could be disregarded otherwise, due to temporal constraints.

In addition, we have also contemplated approaches to improve the classification performance of systems integrating NMF and DBNs models. This has resulted in the development of a new semi-supervised algorithm (SSNMF) which attempts to extract unique class features (instead of characteristics that are shared by several classes) and in an in-depth analysis of the effects of varying the number of layers and neurons in a DBN as well as the effects of adding MBP layers to this type of network.

Concerning the NMF algorithm, we have performed several experiments using three different face recognition benchmarks (CBCL, Yale and AT&T). In terms of classification performance the experiments demonstrate that the NMF-based systems present competitive results as compared with other meritorious face recognition algorithms, while evidencing superior robustness when dealing with different lighting conditions (see Tables 5.1, 5.2, 5.5 and 5.6). In particular, the SSNMF is highly desirable for unbalanced datasets due to its inherent aptitude to extract class specific features. The experimental results concerning the SSNMF proposed algorithm show that it reduces considerably the risk of creating inadequate models as compared to the original NMF method (see Figures 5.16, 5.17 and Table 5.4). This is also patent in Tables 5.5 and 5.6, which confirm that in many cases the individual class features extracted by SSNMF provide a better foundation for building a classification model than the typical NMF characteristics (compare the NMF-SVM and SSNMF-SVM columns).

In addition SSNMF presents other advantages relative to the baseline NMF algorithm. First, it reduces significantly the computational cost inherent to the factorization of the matrices (see Figure 5.15). Second, it creates sparser matrices (see Table 5.3 and Figure 5.3).

Concerning the GPU parallel implementations of the NMF algorithm, we have obtained speedups of two-orders of magnitude for several face recognition benchmarks (see Figures 5.7, 5.8, 5.9, 5.13 and 5.14). Naturally, the associated time savings will exert a profound impact in the development of NMF-based systems. In particular, in real-world scenarios these high-performance GPU implementations may very well be the key factor for the success of many applications, since in order to present new data to the classifier, first it is necessary to calculate the matrix containing the codification of the parts-based images,  $\mathbf{H}$ , that approximates the new data (see Figure 5.2).

Regarding the DBNs, we have presented two complementary approaches that result in a significant reduction of the amount of time spent in pre-training phase. First we have designed an adaptive step size technique to enhance the convergence

of the CD- $k$  algorithm, thereby reducing the number of epochs necessary to train the RBMs that support the DBN infrastructure. Second, we have implemented a highly-scalable GPU parallel implementation of the CD- $k$  algorithm, which boosts notably the algorithms' training speed. The experiments performed using the MNIST dataset, demonstrate the efficiency of both approaches as shown in Figures 5.31, 5.32 and 5.33. The careful design of the CUDA kernels supporting the GPU parallel implementation was vital to obtain speedups up to 46 times. In addition, the proposed adaptive step size technique further reduces the training time by decreasing the number of epochs needed for the networks to converge.

The resulting tool was used to analyze the effects of varying the number of layers and neurons as well as the effects of adding new layers with randomly initialized weights to the pre-trained networks. The influence of MBP layers with selective actuation neurons was also studied. To this end, hundreds of DBNs were trained in both the MNIST and the HHreco datasets. Due to the large number of networks to be trained as part of these particular experiments, we have decided to use only a small number of training samples for each dataset. This seems to be a disadvantage, however, the large number of networks trained backup and endorse the conclusions found. Moreover, according to preliminary tests our conclusions appear to be valid even for larger training datasets.

One of the findings of this study is that the number and diversity of training samples is highly correlated to the quality of the resulting DBN models. Nevertheless, it is possible to build quality models even with few training samples (see Tables 5.7 and 5.8). By increasing the number of training samples we can build better models that are able to capture the underlying regularities of the data, thereby improving the overall system discriminative capacity. Moreover, based on the results, we believe that there is a relation between the number and diversity of training samples and the maximum useful depth (in the sense of improving the classification performance) of a DBN. The rationale is that a DBN can only find the regularities that are actually present on the observed data and increasing the depth of a DBN serves no purpose when the data itself does not exhibit the type of complex regularities that would require additional layers. Therefore increasing the number of samples increases the probability of the data samples to present the same regularities of its true distribution.

Another finding is that the lower-level layer plays a fundamental role within a DBN structure. It is vital to extract a significant number of characteristics from the original data right away in this layer. Failure to do so may compromise the ability of the network to extract more complex and useful features in the next layers. In fact, the results obtained (see Figure 5.37) suggest that the more features (neurons) the first hidden layer comprises the better, although additional experiments are required to confirm this trend.

We also find 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 hidden layer of a DBN can actually improve its classification

performance by allowing the resulting network to further refine its discriminative capacity (see Figure 5.39).

Finally, we have shown that by adding MBP layers with selective actuation neurons to a DBN we could also improve its classification performance, since these neurons provide the means for better generalization by seamless partition of the feature input space (see Figure 5.38).

### **Future Work**

Future work will cover the design and execution of additional experiments concerning both unsupervised approaches (NMF and DBNs). With respect to the NMF algorithm, a line of work consists of determining the impact of the number of parts-based images,  $r$ , in the quality of the resulting solutions. Another line of work consists of assessing the performance of the SSNMF algorithm on unbalanced datasets.

Concerning the DBNs, we aim to study further the relation between the networks architecture (e.g. number of layers, neurons per layer) and the corresponding performance in order to gain a deeper understanding regarding the adequate topology selection. In particular, experiments involving a larger number of samples are desirable to further support and validate our findings.





---

Conclusions and Perspectives

---

---

|            |  |            |
|------------|--|------------|
| <b>6.1</b> | <b>Main Research Accomplishments and Conclusions . .</b> | <b>189</b> |
| <b>6.2</b> | <b>Future Work . . . . .</b>                             | <b>194</b> |

---

This Thesis aims at the development of novel ML algorithms and high-performance implementations of existing ones with data scalability in mind. The rationale consists of increasing the practical applicability of ML solutions. Incidentally, large-scale data problems (“Big Data”) require new approaches for the concomitant problem of missing data and therefore this contingency was also considered.

In this Chapter, we provide an overview of the outcomes of this research work as well as suggestions for future work. Accordingly, Section 6.1 presents the main outcomes of this work and Section 6.2 addresses future lines of research.

## 6.1 Main Research Accomplishments and Conclusions

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 5.2.3. 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.

A novel semi-supervised algorithm, based on the Non-Negative Matrix Factorization (NMF), was presented in Section 5.1.3. The algorithm, designated by 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 inadequate 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.

A novel incremental (instance-based) supervised learning algorithm with built-in multi-class support was presented in Section 4.3. This algorithm, designated by Incremental Hypersphere Classifier (IHC), 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 algorithms 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 (believed to be) lying on the decision frontier 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.

A learning framework (IHC-SVM), encompassing the IHC and SVM algorithms, was devised for application in a protein membership prediction real-world case study. The resulting system uses the IHC for immediate prediction and selection of a subset of the training samples, which is subsequently used to build a model using the SVM algorithm. Moreover, the incremental nature of IHC permits ready detection of significant changes, which require updating the SVM model. Using the IHC we were able to obtain a test F-Measure of 93.73%, while storing less than 40% of the original samples. Naturally, this value is smaller than the one yielded by the baseline SVM model (95.91%), which was created using all the training samples. However, the IHC-SVM approach is able to excel the baseline SVM using only a subset of the data. Using roughly 50% of the original data, it is possible to create improved models (with an F-measure up to 96.39%) and we can compact the data even further and still obtain models that match closely the performance of the baseline model. Thus, there is strong evidence that the process used by the IHC to decide which samples to keep and which to discard is efficient. Overall

the proposed IHC-SVM approach demonstrated that it is able to deal with the everyday dynamic changes of real-world biological databases.

A novel solution, designated by Neural Selective Input Model (NSIM), which empowers Neural Networks (NNs) with the ability to handle Missing Values (MVs), was proposed in Section 4.2. 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.

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 over 2,000 downloads) has attracted the interest of numerous people, benefiting researchers worldwide. Moreover, due to its quality and stringent documentation, GPUMLib

has received a 5 star award from the soft82.com editors, which is given to products that are considered to be above average or excellent in their category.

The GPU parallel implementations of the Back-Propagation (BP) and Multiple Back-Propagation (MBP) algorithms were presented in Section 4.1.3. 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  $5\times$  to  $180\times$  (on a GTX 280 device), are directly correlated to the complexity of the problem.

Using the aforementioned 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, 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 4.1.4. 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 privileges topologies with improved classification performance, saving researchers time and effort.

A total of four distinct GPU parallel implementations of the NMF algorithm were presented in Section 5.1.4. These yielded speedups ranging from  $55\times$  to  $706\times$  (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 5.2.4. The implementation includes the aforementioned 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  $23\times$  and  $46\times$  (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. In this context, 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 hidden 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 Thesis we have presented several methods and strategies that (from the perspective of ML algorithms) 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 resides on the well-designed kernels, which allows the devices to adaptively balance the workload across its many cores, thereby maximizing the occupancy and throughput. Moreover, we have proposed novel methods (the SSNMF and the CD- $k$  adaptive step size technique) that further enhance the scalability (relatively of the original algorithms), making the combined approach (with the GPU implementation) even more attractive. Furthermore, using a different but also successful approach, we presented an algorithm (IHC) that scales linearly with an amount of pre-defined memory, independently of the volume of data to process, which makes it suitable not only for handling large-scale datasets but also data streams.

Finally, in this Thesis we have also successfully addressed the complementary problem of handling missing data in large datasets. In this context, the NSIM proposed approach proved to be a viable and efficient solution for this problem and highlights the path for the application of similar strategies in other ML methods.

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.

## 6.2 Future Work

The work developed in this Thesis is valuable for the scientific community presenting several relevant aspects in machine learning for adaptive multi-core machines. As pointed out in the previous Chapters, and reiterated here, it provides many possible directions for future work.

Strikingly by stacking RBMs in deep architectures – as the DBNs – one can learn features from features in the hope of arriving at 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, experiments involving a larger number of (more complex) samples are needed to gain a deeper insight of its topology to match higher levels of representation. 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 [Fischer and Igel, 2013, Desjardins et al., 2010].

Given the demonstrated potential of NMF decomposition in image processing, a natural direction of future work will be their extension to other conditions which might involve the impact of the number of parts-based images,  $r$ , in the quality of the resulting solutions. On the other hand, conducting experiments on unbalanced datasets to compare the performance of the SSNMF with the baseline NMF algorithm could also be of interest to explore further.

Another possible direction for future research is the design of more efficient ways to automatically fine-tune the gravity,  $g$ , parameter for each class, thereby attributing a distinct importance to each feature in IHC. Additionally, investigating the impact of different distance metrics [Somorjai et al., 2011, Bonet et al., 2008, Skala, 2008, Bao et al., 2004] is worthwhile for fine-grained capturing of the dynamic drift concepts with the developed algorithm. In this line, the integration of Incremental Hypersphere Classifier [Bao et al., 2004] to handle dynamic number and length of high-dimensional data streams would possibly help to better handle concept drift at an instance level.

Following a recent trend towards GPU parallel implementations of ML algorithms, other algorithms could be considered such as complex-valued neural networks, which have recently gained momentum due to their aptitude to deal with specific types of data (e.g. wave phenomena), would also be interesting to accomplish [Hirose, 2013, Nitta, 2013, Alexandre, 2011]. The application field of these networks is expected to be wider, since they can represent more information (e.g. phase and amplitude) [Nitta, 2013].

A final note on the extension of GPUMLib is given. It is important to continue the development of this framework – by filling the dots in Figure 2.11 – in order to augment its attractiveness to researchers worldwide.

---

## Bibliography

---

- [Abramov et al., 2010] Abramov, A., Kulvicius, T., Wörgötter, F., and Dellen, B. (2010). Real-time image segmentation on a GPU. In *Facing the multicore-challenge, LNCS 6310*, pages 131–142. [cited at page 16]
- [Aha et al., 1991] Aha, D. W., Kibler, D., and Albert, M. K. (1991). Instance-based learning algorithms. *Machine Learning*, 6(1):37–66. [cited at page 108]
- [Alavala, 2008] Alavala, C. R. (2008). *Fuzzy Logic and Neural Networks: Basic Concepts & Applications*. New Age International Publishers. [cited at page 96]
- [Alexandre, 2011] Alexandre, L. A. (2011). Single layer complex valued neural network with entropic cost function. In *Proceedings of the 21<sup>st</sup> International Conference on Artificial Neural Networks (ICANN 2011)*, LNCS 6791, pages 331–338. Springer-Verlag. [cited at page 194]
- [Almeida, 1997] Almeida, L. B. (1997). *Handbook of Neural Computation*, chapter C1.2 Multilayer perceptrons, pages C1.2:1–C1.2:30. IOP Publishing Ltd and Oxford University Press. [cited at page 68, 69, 163, 174]
- [Alpaydin, 2010] Alpaydin, E. (2010). *Introduction to Machine Learning (Adaptive Computation and Machine Learning)*. The MIT Press, 2<sup>nd</sup> edition. [cited at page 3, 41, 125]
- [Ayuyev et al., 2009] Ayuyev, V. V., Joseph Jupin, P. W. H., and Obradovic, Z. (2009). Dynamic clustering-based estimation of missing values in mixed type data. In *Proceedings of the 11<sup>th</sup> International Conference on Data Warehousing and Knowledge Discovery (DaWaK 2009)*, LNCS 5691, pages 366–377. Springer-Verlag. [cited at page 94, 95, 101, 102]
- [Bache and Lichman, 2013] Bache, K. and Lichman, M. (2013). UCI machine learning repository. <http://archive.ics.uci.edu/ml>. [cited at page 43]

- [Bao et al., 2004] Bao, Y., Ishii, N., and Du, X. (2004). Combining multiple k-nearest neighbor classifiers using different distance functions. In *Proceedings of the 5<sup>th</sup> International Conference on Intelligent Data Engineering and Automated Learning (IDEAL 2004)*, LNCS 3177, pages 634–641. Springer. [cited at page 194]
- [Bengio, 2009] Bengio, Y. (2009). Learning deep architectures for AI. *Foundations and Trends in Machine Learning*, 2(1):1–127. [cited at page 154, 155, 157, 159, 160, 161, 162]
- [Beringer and Hüllermeier, 2007] Beringer, J. and Hüllermeier, E. (2007). Efficient instance-based learning on data streams. *Intelligent Data Analysis*, 11(6):627–650. [cited at page 115]
- [Bernhard and Keriven, 2006] Bernhard, F. and Keriven, R. (2006). Spiking neurons on GPUs. In *Proceedings of the 2006 International Conference on Computational Science (ICCS 2006)*, LNCS 3994, pages 236–243. Springer. [cited at page 17]
- [Bibi and Stamelos, 2006] Bibi, S. and Stamelos, I. (2006). Selecting the appropriate machine learning techniques for the prediction of software development costs. In *Proceedings of Artificial Intelligence Applications and Innovations*, pages 533–540. [cited at page 106]
- [Bishop, 2006] Bishop, C. M. (2006). *Pattern Recognition and Machine Learning*. Springer. [cited at page 5, 6, 41, 56, 64, 66, 67, 90, 111, 158]
- [Blackard and Dean, 1999] Blackard, J. A. and Dean, D. J. (1999). 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. [cited at page 84]
- [Bohn, 1998] Bohn, C.-A. (1998). Kohonen feature mapping through graphics hardware. In *Proceedings of the 1998 International Conference on Computational Intelligence and Neurosciences (ICCN 1998)*, pages 64–67. [cited at page 17]
- [Bonet et al., 2008] Bonet, I., Rodríguez, A., Grau, R., García, M. M., Saeys, Y., and Nowé, A. (2008). Comparing distance measures with visual methods. In *Proceedings of the 7<sup>th</sup> Mexican International Conference on Artificial Intelligence: Advances in Artificial Intelligence (MICA I 2008)*, LNCS 5317, pages 90–99. Springer. [cited at page 194]
- [Bottou and Bousquet, 2008] Bottou, L. and Bousquet, O. (2008). Learning using large datasets. In *Mining Massive DataSets for Security*, NATO ASI Workshop Series. IOS Press. [cited at page 4]



- [Bramer, 2007] Bramer, M. A. (2007). *Principles of data mining*. Springer-Verlag. [cited at page 92, 95, 96]
- [Brandstetter and Artusi, 2008] Brandstetter, A. and Artusi, A. (2008). Radial basis function networks GPU-based implementation. *IEEE Transactions on Neural Networks*, 19(12):2150–2154. [cited at page 16, 17]
- [Brunton et al., 2006] Brunton, A., Shu, C., and Roth, G. (2006). Belief propagation on the GPU for stereo vision. In *Proceedings of the 3<sup>rd</sup> Canadian Conference on Computer and Robot Vision (CRV 2006)*, pages 76–81. IEEE Computer Society. [cited at page 17]
- [Bucur and Florea, 2011] Bucur, L. and Florea, A. (2011). Techniques for prediction in chaos: A comparative study on financial data. *U.P.B. Scientific Bulletin, Series C*, 73(3):17–32. [cited at page 74]
- [Campbell et al., 2005] Campbell, A., Berglund, E., and Streit, A. (2005). Graphics hardware implementation of the parameter-less self-organising map. In *Proceedings of the 2005 Intelligent Data Engineering and Automated Learning (IDEAL)*, LNCS 3578, pages 343–350. Springer. [cited at page 17]
- [Cano et al., 2012] Cano, A., Zafra, A., and Ventura, S. (2012). Speeding up the evaluation phase of GP classification algorithms on GPUs. *Soft Computing*, 16(2):187–202. [cited at page 19]
- [Carreira-Perpiñán and Hinton, 2005] Carreira-Perpiñán, M. A. and Hinton, G. E. (2005). On contrastive divergence learning. In *Proceedings of the 10<sup>th</sup> International Workshop on Artificial Intelligence and Statistics (AISTATS 2005)*, pages 33–40. [cited at page 157, 159, 160]
- [Catanzaro et al., 2008] Catanzaro, B., Sundaram, N., and Keutzer, K. (2008). Fast support vector machine training and classification on graphics processors. In *Proceedings of the 25<sup>th</sup> International Conference on Machine Learning (ICML 2008)*, volume 307, pages 104–111. ACM. [cited at page 16, 17]
- [Cavuoti et al., 2014] Cavuoti, S., Garofalo, M., Brescia, M., Paolillo, M., Pescape', A., Longo, G., and Ventre, G. (2014). Astrophysical data mining with GPU. a case study: Genetic classification of globular clusters. *New Astronomy*, 26:12–22. [cited at page 19]
- [Cavuoti et al., 2013] Cavuoti, S., Garofalo, M., Brescia, M., Pescape', A., Longo, G., and Ventre, G. (2013). Genetic algorithm modeling with GPU parallel computing technology. In Apolloni, B., Bassis, S., Esposito, A., and Morabito, F. C., editors, *Neural Nets and Surroundings*, volume 19 of *Smart Innovation, Systems and Technologies*, pages 29–39. Springer Berlin Heidelberg. [cited at page 19]

- [Cecilia et al., 2013] Cecilia, J. M., Nisbet, A., Amos, M., García, J. M., and Ujaldón, M. (2013). Enhancing GPU parallelism in nature-inspired algorithms. *The Journal of Supercomputing*, 63(3):773–789. [cited at page 19]
- [Chacko et al., 2010] Chacko, B. P., Krishnan, V. R. V., and Anto, P. B. (2010). Character recognition using multiple back propagation algorithm. In *Proceedings of the National Conference on Image Processing*. [cited at page 74]
- [Chang and Lin, 2011] Chang, C.-C. and Lin, C.-J. (2011). LIBSVM: A library for support vector machines. *ACM Transactions on Intelligent Systems and Technology*, 2(3):1–27. [cited at page 111]
- [Chapelle et al., 2006] Chapelle, O., Schölkopf, B., and Zien, A. (2006). Introduction to semi-supervised learning. In *Semi-Supervised Learning*, chapter 1, pages 1–14. The MIT Press. [cited at page 5, 133]
- [Che et al., 2008a] Che, S., Boyer, M., Meng, J., Tarjan, D., Sheaffer, J. W., and Skadron, K. (2008a). A performance study of general-purpose applications on graphics processors using CUDA. *Journal of Parallel and Distributed Computing*, 68(10):1370–1380. [cited at page 20, 22]
- [Che et al., 2008b] Che, S., Li, J., Sheaffer, J. W., Skadron, K., and Lach, J. (2008b). Accelerating compute-intensive applications with GPUs and FPGAs. In *Symposium on Application Specific Processors (SASP 2008)*, pages 101–107. [cited at page 16]
- [Chellapilla et al., 2006] Chellapilla, K., Puri, S., and Simard, P. (2006). High performance convolutional neural networks for document processing. In *Proceedings of the 10<sup>th</sup> International Workshop on Frontiers in Handwriting Recognition*. [cited at page 17]
- [Cheng et al., 2005] Cheng, B. Y. M., Carbonell, J. G., and Klein-Seetharaman, J. (2005). Protein classification based on text document classification techniques. *Proteins: Structure, Function, and Bioinformatics*, 58(4):955–970. [cited at page 54]
- [Cherkassky and Mulier, 2007] Cherkassky, V. and Mulier, F. (2007). *Learning From Data: Concepts, Theory, and Methods*. John Wiley & Sons, 2<sup>nd</sup> edition. [cited at page 3, 56, 62, 123]
- [Chitty, 2012] Chitty, D. M. (2012). Fast parallel genetic programming: multi-core CPU versus many-core GPU. *Soft Computing*, 16(10):1795–1814. [cited at page 19]
- [Clarke et al., 2009] Clarke, B., Fokoué, E., and Zhang, H. H. (2009). *Principles and Theory for Data Mining and Machine Learning*. Springer. [cited at page 106]

- [Correia et al., 2011] Correia, D., Pereira, C., Verissimo, P., and Dourado, A. (2011). A platform for peptidase detection based on text mining techniques. In *International Symposium on Computational Intelligence for Engineering Systems*. [cited at page 54]
- [Cybenko, 1989] Cybenko, G. (1989). Approximation by superpositions of a sigmoidal function. *Mathematics of Control, Signals, and Systems (MCSS)*, 2(4):303–314. [cited at page 66]
- [Desjardins et al., 2010] Desjardins, G., Courville, A., Bengio, Y., Vincent, P., and Delalleau, O. (2010). Tempered Markov Chain Monte Carlo for training of restricted Boltzmann machines. In *Proceedings of the 13<sup>th</sup> International Conference on Artificial Intelligence and Statistics*, pages 145–152. [cited at page 194]
- [Do et al., 2008] Do, T.-N., Nguyen, V.-H., and Poulet, F. (2008). Speed up SVM algorithm for massive classification tasks. In *Proceedings of the 4<sup>th</sup> International Conference on Advanced Data Mining and Applications (ADMA 2008)*, LNCS 5139, pages 147–157. Springer. [cited at page 17]
- [Drugowitsch, 2008] Drugowitsch, J. (2008). A learning classifier systems model. In *Design and Analysis of Learning Classifier Systems*, volume 139 of *Studies in Computational Intelligence*, pages 29–44. Springer. [cited at page 62]
- [Duch and Jankowski, 1999] Duch, W. and Jankowski, N. (1999). Survey of neural transfer functions. *Neural Computing Surveys*, 2:163–213. [cited at page 62, 65, 66]
- [Džeroski et al., 2009] Džeroski, S., Panov, P., and Ženko, B. (2009). Machine learning, ensemble methods in. In *Encyclopedia of Complexity and Systems Science*, pages 5317–5325. Springer. [cited at page 69, 70]
- [Essen et al., 2012] Essen, B. V., Macaraeg, C., Gokhale, M., and Prenger, R. (2012). Accelerating a random forest classifier: Multi-core, GP-GPU, or FPGA? In *IEEE 20<sup>th</sup> Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM 2012)*, pages 232–239. [cited at page 19]
- [Fahlman and Lebiere, 1990] Fahlman, S. E. and Lebiere, C. (1990). The cascade-correlation learning architecture. In *Advances in Neural Information Processing Systems*, volume 2, pages 524–532. [cited at page 49]
- [Fernando and Kilgard, 2003] Fernando, R. and Kilgard, M. J. (2003). *The Cg Tutorial: The Definitive Guide to Programmable Real-Time Graphics*. Addison-Wesley Professional. [cited at page 19]

- [Fischer and Igel, 2013] Fischer, A. and Igel, C. (2013). Training restricted Boltzmann machines: An introduction. *Pattern Recognition*. [cited at page 194]
- [Funahashi, 1989] Funahashi, K. (1989). On the approximate realization of continuous mappings by neural networks. *Neural Networks*, 2(3):183–192. [cited at page 66]
- [Gama et al., 2004] Gama, J., Medas, P., and Rodrigues, P. (2004). Concept drift in decision trees learning from data streams. In *European Symposium on Intelligent Technologies Hybrid Systems and their implementation on Smart Adaptive Systems Eunate 2004*, pages 218–225. [cited at page 46]
- [Gama et al., 2009] Gama, J., Sebastião, R., and Rodrigues, P. (2009). Issues in evaluation of stream learning algorithms. In *Proceedings of the 15<sup>th</sup> ACM SIGKDD International Conference on Knowledge Discovery and Data Mining (KDD 2009)*, pages 329–338. [cited at page 104]
- [Garcia et al., 2008] Garcia, V., Debreuve, E., and Barlaud, M. (2008). 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)*, pages 1–6. [cited at page 17]
- [García-Laencina et al., 2010] García-Laencina, P. J., Sancho-Gómez, J.-L., and Figueiras-Vidal, A. R. (2010). Pattern classification with missing data: a review. *Neural Computing and Applications*, 19(2):263–282. [cited at page 92, 93, 94, 95, 96]
- [García-Pedrajas et al., 2010] García-Pedrajas, N., Castillo, J. A. R. D., and Ortiz-Boyer, D. (2010). A cooperative coevolutionary algorithm for instance selection for instance-based learning. *Machine Learning*, 78(3):381–420. [cited at page 4, 105, 112]
- [Garg and Murty, 2009] Garg, V. K. and Murty, M. (2009). Feature subspace SVMs (FS-SVMs) for high dimensional handwritten digit recognition. *International Journal of Data Mining, Modelling and Management (IJDM)*, 1(4):411–436. [cited at page 128]
- [Garland and Kirk, 2010] Garland, M. and Kirk, D. B. (2010). Understanding throughput-oriented architectures. *Communications of the ACM*, 53(11):58–66. [cited at page 4, 16, 18]
- [Giannesini and Saux, 2012] Giannesini, F. and Saux, L. B. (2012). GPU-accelerated one-class SVM for exploration of remote sensing data. In *IEEE International Geoscience and Remote Sensing Symposium (IGARSS 2012)*, pages 7349–7352. [cited at page 19]

- [Gillis and Glineur, 2010] Gillis, N. and Glineur, F. (2010). Using under-approximations for sparse nonnegative matrix factorization. *Pattern Recognition*, 43(4):1676–1687. [cited at page 128, 129, 149]
- [Gonçalves, 2012] Gonçalves, J. (2012). Development of support vector machines (SVMs) in graphics processing units for object recognition. Master’s thesis, University of Coimbra. [cited at page 33]
- [Granmo, 2012] Granmo, O.-C. (2012). Short-term forecasting of electricity consumption using gaussian processes. Master’s thesis, University of Agder. [cited at page 74]
- [Grauer-Gray et al., 2008] Grauer-Gray, S., Kambhamettu, C., and Palaniappan, K. (2008). GPU implementation of belief propagation using CUDA for cloud tracking and reconstruction. In *Proceedings of the 5<sup>th</sup> IAPR Workshop on Pattern Recognition in Remote Sensing (PRRS 2008)*, pages 1–4. [cited at page 17]
- [Guzhva et al., 2009] Guzhva, A., Dolenko, S., and Persiantsev, I. (2009). Multifold acceleration of neural network computations using GPU. In *Proceedings of the 19<sup>th</sup> International Conference on Artificial Neural Networks (ICANN 2009)*, LNCS 5768, pages 373–380. Springer. [cited at page 17]
- [Halfhill, 2009] Halfhill, T. R. (2009). Looking beyond graphics. Technical report, In-Stat. [cited at page 24, 25, 27]
- [Hall et al., 2009] Hall, M., Frank, E., Holmes, G., Pfahringer, B., Reutemann, P., and Witten, I. H. (2009). The WEKA data mining software: an update. *SIGKDD Explorations Newsletter*, 11(1):10–18. [cited at page 101]
- [Harding and Banzhaf, 2007] Harding, S. and Banzhaf, W. (2007). Fast genetic programming on GPUs. In *Proceedings of the 10<sup>th</sup> European Conference on Genetic Programming (EuroGP 2007)*, LNCS 4445, pages 90–101. Springer. [cited at page 17]
- [Haykin, 1998] Haykin, S. (1998). *Neural Networks: A Comprehensive Foundation*. Prentice Hall, 2<sup>nd</sup> edition. [cited at page 68]
- [Herrero-Lopez, 2011] Herrero-Lopez, S. (2011). Accelerating SVMs by integrating GPUs into mapreduce clusters. In *IEEE International Conference on Systems, Man, and Cybernetics (SMC)*, pages 1298–1305. [cited at page 19]
- [Hey et al., 2009] Hey, T., Tansley, S., and Tolle, K., editors (2009). *The Fourth Paradigm: Data-Intensive Scientific Discovery*. Microsoft Research. [cited at page 2, 3, 4, 14, 16]
- [Hinton, 2002] Hinton, G. E. (2002). Training products of experts by minimizing contrastive divergence. *Neural Computation*, 14(8):1771–1800. [cited at page 160, 161]

- [Hinton, 2010] Hinton, G. E. (2010). A practical guide to training restricted Boltzmann machines. Technical report, Department of Computer Science, University of Toronto. [cited at page 156, 157, 158, 159, 160, 161, 163]
- [Hinton et al., 2006] Hinton, G. E., Osindero, S., and Teh, Y.-W. (2006). A fast learning algorithm for deep belief nets. *Neural Computation*, 18(7):1527–1554. [cited at page 155, 162]
- [Hirose, 2013] Hirose, A., editor (2013). *Complex-Valued Neural Networks: Advances and Applications*. John Wiley & Sons. [cited at page 194]
- [Hornik et al., 1989] Hornik, K., Stinchcombe, M., and White, H. (1989). Multilayer feedforward networks are universal approximators. *Neural Networks*, 2(5):359–366. [cited at page 66]
- [Hse and Newton, 2004] Hse, H. and Newton, A. R. (2004). Sketched symbol recognition using Zernike moments. In *Proceedings of the 17<sup>th</sup> International Conference on Pattern Recognition*, volume 1, pages 367–370. [cited at page 46]
- [Hua and Sun, 2001] Hua, S. and Sun, Z. (2001). Support vector machine approach for protein subcellular localization prediction. *Bioinformatics*, 17(8):721–728. [cited at page 111]
- [Hui, 2011] Hui, C.-L., editor (2011). *Artificial Neural Networks - Application*. InTech. [cited at page 62]
- [Hung and Wang, 2012] Hung, Y. and Wang, W. (2012). Accelerating parallel particle swarm optimization via GPU. *Optimization Methods and Software*, 27(1):33–51. [cited at page 19]
- [Jain et al., 2006] Jain, S., Lange, S., and Zilles, S. (2006). Towards a better understanding of incremental learning. In *Proc. of the 17<sup>th</sup> International Conference on Algorithmic Learning Theory, LNAI 4264*, pages 169–183. [cited at page 4, 105]
- [Jang et al., 2008] Jang, H., Park, A., and Jung, K. (2008). Neural network implementation using CUDA and OpenMP. In *Proceedings of the 2008 Digital Image Computing: Techniques and Applications (DICTA 2008)*, pages 155–161. [cited at page 14, 17]
- [Jian et al., 2013] Jian, L., Wang, C., Liu, Y., Liang, S., Yi, W., and Shi, Y. (2013). Parallel data mining techniques on graphics processing unit with compute unified device architecture (CUDA). *The Journal of Supercomputing*, 64(3):942–967. [cited at page 19]
- [Jowell and the Central Coordinating Team, 2007] Jowell, R. and the Central Coordinating Team (2003, 2005, 2007). European social survey 2002/2003; 2004/2005; 2006/2007. [cited at page 49]

- [Kanwisher, 2010] Kanwisher, N. (2010). 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. [cited at page 70]
- [Karhunen, 2011] Karhunen, J. (2011). Robust PCA methods for complete and missing data. *Neural Network World*, 21(5):357–392. [cited at page 5, 91]
- [King, 2009] King, D. E. (2009). Dlib-ml: A machine learning toolkit. *Journal of Machine Learning Research*, 10:1755–1758. [cited at page 14]
- [Kotsiantis et al., 2006a] Kotsiantis, S., Kanellopoulos, D., and Pintelas, P. (2006a). Data preprocessing for supervised learning. *International Journal of Computer Science*, 1(2):111–117. [cited at page 55, 57, 105]
- [Kotsiantis et al., 2006b] Kotsiantis, S. B., Zaharakis, I. D., and Pintelas, P. E. (2006b). Machine learning: a review of classification and combining techniques. *Artificial Intelligence Review*, 26(3):159–190. [cited at page 5, 91, 94]
- [Kumar et al., 2013] Kumar, S., Kumawat, T., kumar Marwal, N., and Singh, B. K. (2013). Artificial neural network and its applications. *International Journal of Computer Science and Management Research*, 2(2):1621–1626. [cited at page 62]
- [Lahabar et al., 2008] Lahabar, S., Agrawal, P., and Narayanan, P. J. (2008). High performance pattern recognition on GPU. In *Proceedings of the 2008 National Conference on Computer Vision Pattern Recognition Image Processing and Graphics*, pages 154–159. [cited at page 17]
- [Langdon, 2011] Langdon, W. B. (2011). Graphics processing units and genetic programming: an overview. *Soft Computing*, 15(8):1657–1669. [cited at page 19]
- [Langdon and Banzhaf, 2008] Langdon, W. B. and Banzhaf, W. (2008). A SIMD interpreter for genetic programming on GPU graphics cards. In *Proceedings of the 11<sup>th</sup> European Conference on Genetic Programming (EuroGP 2008)*, LNCS 4971, pages 73–85. Springer. [cited at page 17]
- [Larochelle et al., 2007] Larochelle, H., Erhan, D., Courville, A., Bergstra, J., and Bengio, Y. (2007). An empirical evaluation of deep architectures on problems with many factors of variation. In *Proceedings of the 24<sup>th</sup> international conference on Machine learning (ICML 2007)*, pages 473–480. ACM. [cited at page 153, 154, 155, 162]
- [Lee and Seung, 1999] Lee, D. D. and Seung, H. S. (1999). Learning the parts of objects by non-negative matrix factorization. *Nature*, 401:788–791. [cited at page 127, 129]

- [Lee and Seung, 2000] Lee, D. D. and Seung, H. S. (2000). Algorithms for non-negative matrix factorization. In *Advances in Neural Information Processing Systems (NIPS 2000)*, pages 556–562. MIT Press. [cited at page 130]
- [Lee et al., 2009] Lee, H., Grosse, R., Ranganath, R., and Ng, A. Y. (2009). Convolutional deep belief networks for scalable unsupervised learning of hierarchical representations. In *Proceedings of the 26<sup>th</sup> Annual International Conference on Machine Learning (ICML 2009)*, pages 609–616. ACM. [cited at page 154, 161, 162]
- [Li et al., 2013] Li, Q., Salman, R., Test, E., Strack, R., and Kecman, V. (2013). Parallel multitask cross validation for support vector machine using GPU. *Journal of Parallel and Distributed Computing*, 73(3):293–302. [cited at page 19]
- [Li et al., 2010] Li, Z., Wu, X., and Peng, H. (2010). Nonnegative matrix factorization on orthogonal subspace. *Pattern Recognition Letters*, 31(9):905–911. [cited at page 128, 129]
- [Lim and Zainuddin, 2008] Lim, E. A. and Zainuddin, Z. (2008). A comparative study of missing value estimation methods: Which method performs better? In *Proceedings of the International Conference on Electronic Design (ICED 2008)*, pages 1–5. [cited at page 94, 96]
- [Little and Rubin, 2002] Little, R. J. A. and Rubin, D. B. (2002). *Statistical analysis with missing data*. Wiley, 2<sup>nd</sup> edition. [cited at page 93, 96]
- [Lopes et al., 2012a] Lopes, N., Correia, D., Pereira, C., Ribeiro, B., and Dourado, A. (2012a). An incremental hypersphere learning framework for protein membership prediction. In *7<sup>th</sup> International Conference on Hybrid Artificial Intelligent Systems (HAIS 2012)*, LNCS 7208, pages 429–439. [cited at page 8, 10, 53]
- [Lopes and Ribeiro, 1999] Lopes, N. and Ribeiro, B. (1999). A data pre-processing tool for neural networks (DTPNN) use in a moulding injection machine. In *Second World Manufacturing Congress (WMC 1999)*. [cited at page 55]
- [Lopes and Ribeiro, 2001] Lopes, N. and Ribeiro, B. (2001). Hybrid learning in a multi-neural network architecture. In *INNS-IEEE International Joint Conference on Neural Networks (IJCNN 2001)*, volume 4, pages 2788–2793. [cited at page 70, 71, 73, 74, 89]
- [Lopes and Ribeiro, 2003] Lopes, N. and Ribeiro, B. (2003). An efficient gradient-based learning algorithm applied to neural networks with selective actuation neurons. *Neural, Parallel and Scientific Computations*, 11:253–272. [cited at page 70, 71, 73, 74, 81]



- [Lopes and Ribeiro, 2009a] Lopes, N. and Ribeiro, B. (2009a). Fast pattern classification of ventricular arrhythmias using graphics processing units. In *Proceedings of the 14<sup>th</sup> Iberoamerican Congress on Pattern Recognition (CIARP 2009)*, LNCS 5856, pages 603–610. Springer. [cited at page 9, 10, 15, 63, 89]
- [Lopes and Ribeiro, 2009b] Lopes, N. and Ribeiro, B. (2009b). GPU implementation of the multiple back-propagation algorithm. In *Proceedings of the 2009 Intelligent Data Engineering and Automated Learning (IDEAL 2009)*, LNCS 5788, pages 449–456. Springer. [cited at page 9, 17]
- [Lopes and Ribeiro, 2009c] Lopes, N. and Ribeiro, B. (2009c). MBPGPU: A supervised pattern classifier for graphical processing units. In *15<sup>th</sup> edition of the Portuguese Conference on Pattern Recognition (RECPAD 2009)*. [cited at page 9, 79]
- [Lopes and Ribeiro, 2010a] Lopes, N. and Ribeiro, B. (2010a). A hybrid face recognition approach using GPULib. In *15th Iberoamerican Congress on Pattern Recognition (CIARP 2010)*, LNCS 6419, pages 96–103. [cited at page 10, 11, 134]
- [Lopes and Ribeiro, 2010b] Lopes, N. and Ribeiro, B. (2010b). Non-negative matrix factorization implementation using graphic processing units. In *Proceedings of the 11<sup>th</sup> International Conference on Intelligent Data Engineering and Automated Learning (IDEAL 2010)*, LNCS 6283, pages 275–283. Springer Berlin Heidelberg. [cited at page 10, 17, 149]
- [Lopes and Ribeiro, 2010c] Lopes, N. and Ribeiro, B. (2010c). Stochastic GPU-based multithread implementation of multiple back-propagation. In *Second International Conference on Agents and Artificial Intelligence (ICAART 2010)*, pages 271–276. [cited at page 9]
- [Lopes and Ribeiro, 2010d] Lopes, N. and Ribeiro, B. (2010d). 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)*. [cited at page 9, 98]
- [Lopes and Ribeiro, 2011a] Lopes, N. and Ribeiro, B. (2011a). An evaluation of multiple feed-forward networks on GPUs. *International Journal of Neural Systems (IJNS)*, 21(1):31–47. [cited at page 9, 20, 63]
- [Lopes and Ribeiro, 2011b] Lopes, N. and Ribeiro, B. (2011b). A fast optimized semi-supervised non-negative matrix factorization algorithm. In *IEEE International Joint Conference on Neural Networks (IJCNN 2011)*, pages 2495–2500. [cited at page 8, 132, 133]

- [Lopes and Ribeiro, 2011c] Lopes, N. and Ribeiro, B. (2011c). GPULib: An efficient open-source GPU machine learning library. *International Journal of Computer Information Systems and Industrial Management Applications*, 3:355–362. [cited at page 9, 14, 35]
- [Lopes and Ribeiro, 2011d] Lopes, N. and Ribeiro, B. (2011d). An incremental class boundary preserving hypersphere classifier. In *International Conference on Neural Information Processing (ICONIP 2011), Part II, LNCS 7063*, pages 690–699. [cited at page 8, 105, 118]
- [Lopes and Ribeiro, 2011e] Lopes, N. and Ribeiro, B. (2011e). Incremental learning for non-stationary patterns. In *17<sup>th</sup> edition of the Portuguese Conference on Pattern Recognition (RECPAD 2011)*. [cited at page 8, 105]
- [Lopes and Ribeiro, 2011f] Lopes, N. and Ribeiro, B. (2011f). A robust learning model for dealing with missing values in many-core architectures. In *10<sup>th</sup> International Conference on Adaptive and Natural Computing Algorithms (ICANNGA 2011), Part II, LNCS 6594*, pages 108–117. Springer Berlin Heidelberg. [cited at page 9, 10, 53, 92, 96, 97]
- [Lopes and Ribeiro, 2012a] Lopes, N. and Ribeiro, B. (2012a). Handling missing values via a neural selective input model. *Neural Network World*, 22(4):357–370. [cited at page 9, 96, 97, 98, 99]
- [Lopes and Ribeiro, 2012b] Lopes, N. and Ribeiro, B. (2012b). Improving convergence of restricted Boltzmann machines via a learning adaptive step size. In *Progress in Pattern Recognition, Image Analysis, Computer Vision, and Applications, LNCS 7441*, pages 511–518. Springer Berlin / Heidelberg. [cited at page 8, 10, 163, 171, 174]
- [Lopes and Ribeiro, 2012c] Lopes, N. and Ribeiro, B. (2012c). 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. [cited at page 10, 11, 134]
- [Lopes and Ribeiro, 2013] Lopes, N. and Ribeiro, B. (2013). Towards adaptive learning with improved convergence of deep belief networks on graphics processing units. *Pattern Recognition*. [cited at page 8, 10, 11, 179, 180, 182]
- [Lopes et al., 2012b] Lopes, N., Ribeiro, B., and Gonçalves, J. (2012b). Restricted Boltzmann machines and deep belief networks on multi-core processors. In *IEEE International Joint Conference on Neural Networks (IJCNN 2012)*. [cited at page 8, 10, 165, 166, 168, 171]
- [Lopes et al., 2010] Lopes, N., Ribeiro, B., and Quintas, R. (2010). GPULib: A new library to combine machine learning algorithms with graphics processing

- units. In *10<sup>th</sup> International Conference on Hybrid Intelligent Systems (HIS 2010)*, pages 229–232. [cited at page 9, 14, 35]
- [López-Molina et al., 2008] López-Molina, T., Pérez-Méndez, A., and Rivas-Echeverría, F. (2008). Missing values imputation techniques for neural networks patterns. In *Proceedings of the 12<sup>th</sup> WSEAS international conference on Systems (ICS 2008)*, pages 290–295. [cited at page 93, 94, 95, 96]
- [Luo et al., 2005] Luo, Z., Liu, H., and Wu, X. (2005). Artificial neural network computation on graphic process unit. In *Proceedings of the 2005 IEEE International Joint Conference on Neural Networks (IJCNN 2005)*, volume 1, pages 622–626. [cited at page 17]
- [Lyman et al., 2003] Lyman, P., Varian, H. R., Swearingen, K., Charles, P., Good, N., Jordan, L. L., and Pal, J. (2003). How much information? <http://www.sims.berkeley.edu/how-much-info-2003>. [cited at page 2]
- [Markey et al., 2006] Markey, M. K., Tourassi, G. D., Margolis, M., and DeLong, D. M. (2006). Impact of missing data in evaluating artificial neural networks trained on complete data. *Computers in Biology and Medicine*, 36(5):516–525. [cited at page 91]
- [Markoff, 2012] Markoff, J. (2012). Giant steps in teaching computers to think like us: ‘neural nets’ mimic the ways human minds listen, see and execute. *International Herald Tribune*,, November, 24-25:1,8. [cited at page 153, 154]
- [Marques, 2007] Marques, A. (2007). Feature extraction and PVC detection using neural networks and support vector machines. Master’s thesis, University of Coimbra. [cited at page 54, 55, 89]
- [Marsland, 2009] Marsland, S. (2009). *Machine Learning: An Algorithmic Perspective*. Chapman & Hall / CRC. [cited at page 125, 126]
- [Martínez-Zarzuola et al., 2007] Martínez-Zarzuola, M., Pernas, F. J. D., Higuera, J. F. D., and Rodríguez, M. A. (2007). Fuzzy ART neural network parallel computing on the GPU. In *Proceedings of the 9<sup>th</sup> International Work-Conference on Artificial Neural Networks (IWANN 2007)*, LNCS 4507, pages 463–470. Springer. [cited at page 17]
- [Masud et al., 2010] Masud, M. M., Chen, Q., Khan, L., Aggarwal, C. C., Gao, J., Han, J., and Thuraisingham, B. M. (2010). Addressing concept-evolution in concept-drifting data streams. In *Proceedings of the 10<sup>th</sup> IEEE International Conference on Data Mining (ICDM 2010)*, pages 929–934. [cited at page 117]
- [Mjolsness and DeCoste, 2001] Mjolsness, E. and DeCoste, D. (2001). Machine learning for science: State of the art and future prospects. *Science*, 293(5537):2051–2055. [cited at page 3]

- [Mockus, 2008] Mockus, A. (2008). *Guide to Advanced Empirical Software Engineering*, chapter 7 – Missing Data in Software Engineering, pages 185–200. Springer-Verlag. [cited at page 93, 94]
- [Moens, 2006] Moens, M.-F. (2006). *Information Extraction: Algorithms and Prospects in a Retrieval Context*. The Information Retrieval Series. Springer-Verlag. [cited at page 2]
- [Morgado et al., 2011] Morgado, L., Pereira, C., Veríssimo, P., and Dourado, A. (2011). A support vector machine based framework for protein membership prediction. In *Computational Intelligence for Engineering Systems*, volume 46 of *Intelligent Systems, Control and Automation: Science and Engineering*, pages 90–103. Springer. [cited at page 53]
- [Munakata, 2008] Munakata, T. (2008). *Fundamentals of the New Artificial Intelligence: Neural, Evolutionary, Fuzzy and More (Texts in Computer Science)*. Springer-Verlag, 2<sup>nd</sup> edition. [cited at page 63]
- [Murphy, 2012] Murphy, K. P. (2012). *Machine Learning: A Probabilistic Perspective*. The MIT Press. [cited at page 61]
- [Murzin et al., 1995] Murzin, A. G., Brenner, S. E., Hubbard, T., and Chothia, C. (1995). SCOP: a structural classification of proteins database for the investigation of sequences and structures. *Journal of Molecular Biology*, 247(4):536–540. [cited at page 53]
- [Nageswaran et al., 2009] Nageswaran, J. M., Dutt, N., Krichmar, J. L., Nicolau, A., and Veidenbaum, A. V. (2009). A configurable simulation environment for the efficient simulation of large-scale spiking neural networks on graphics processors. *Neural Networks*, 22(5–6):791–800. [cited at page 17]
- [Nelwamondo et al., 2007] Nelwamondo, F. V., Mohamed, S., and Marwala, T. (2007). Missing data: A comparison of neural network and expectation maximization techniques. *Current Science*, 93(11):1514–1521. [cited at page 92, 101]
- [Nicoletti et al., 2009] Nicoletti, M. C., Bertini Jr., J. R., Elizondo, D., Franco, L., and Jerez, J. M. (2009). Constructive neural network algorithms for feedforward architectures suitable for classification tasks. In *Constructive Neural Networks*, volume 258 of *Studies in Computational Intelligence*, pages 1–23. Springer. [cited at page 123]
- [Nitta, 2013] Nitta, T. (2013). Local minima in hierarchical structures of complex-valued neural networks. *Neural Networks*, 43:1–7. [cited at page 194]
- [NVIDIA, 2009] NVIDIA (2009). NVIDIA’s next generation CUDA compute architecture: Fermi. [cited at page 20, 25]

- [NVIDIA, 2012a] NVIDIA (2012a). CUDA C best practices guide: Design guide. [cited at page 27]
- [NVIDIA, 2012b] NVIDIA (2012b). NVIDIA CUDA C programming guide: Version 4.2. [cited at page 21, 22, 25, 27, 28]
- [Oh and Jung, 2004] Oh, K.-S. and Jung, K. (2004). GPU implementation of neural networks. *Pattern Recognition*, 37(6):1311–1314. [cited at page 17]
- [Olvera-López et al., 2010] Olvera-López, J. A., Carrasco-Ochoa, J. A., Martínez-Trinidad, J. F., and Kittler, J. (2010). A review of instance selection methods. *Artificial Intelligence Review*, 34(2):133–143. [cited at page 105]
- [Osuna et al., 1997] Osuna, E. E., Freund, R., and Girosi, F. (1997). Support vector machines: Training and applications. Technical report, Massachusetts Institute of Technology. [cited at page 5]
- [Owens et al., 2008] Owens, J. D., Houston, M., Luebke, D., Green, S., Stone, J. E., and Phillips, J. C. (2008). GPU computing. *Proceedings of the IEEE*, 96(5):879–899. [cited at page 4, 14, 15, 18]
- [Owens et al., 2007] Owens, J. D., Luebke, D., Govindaraju, N., Harris, M., Krüger, J., Lefohn, A., and Purcell, T. J. (2007). A survey of general-purpose computation on graphics hardware. *Computer Graphics Forum*, 26(1):80–113. [cited at page 15, 19, 20]
- [Pappa and Freitas, 2010] Pappa, G. L. and Freitas, A. (2010). *Automating the Design of Data Mining Algorithms: An Evolutionary Computation Approach*. Natural Computing Series. Springer. [cited at page 106]
- [Pereira et al., 2011] Pereira, C., Morgado, L., Correia, D., Verissimo, P., and Dourado, A. (2011). Kernel machines for proteomics data analysis: Algorithms and tools. presented at the European Network for Business and Industrial Statistics, Coimbra, Portugal. [cited at page 54]
- [Piękniewski and Rybicki, 2004] Piękniewski, F. and Rybicki, L. (2004). Visual comparison of performance for different activation functions in MLP networks. In *IEEE International Joint Conference on Neural Networks (IJCNN 2004)*, volume 4, pages 2947–2952. [cited at page 62, 64, 65]
- [Quintas, 2010] Quintas, R. (2010). GPU implementation of RBF neural networks in audio steganalysis. Master’s thesis, University of Coimbra. [cited at page 33]
- [Raina et al., 2009] Raina, R., Madhavan, A., and Ng, A. Y. (2009). Large-scale deep unsupervised learning using graphics processors. In *Proceedings of the 26<sup>th</sup> Annual International Conference on Machine Learning (ICML 2009)*, volume 382, pages 873–880. ACM. [cited at page 17]

- [Ranzato et al., 2007] Ranzato, M., Boureau, Y., and LeCun, Y. (2007). Sparse feature learning for deep belief networks. In *Advances in Neural Information Processing Systems (NIPS 2007)*, volume 20, pages 1185–1192. [cited at page 126, 155, 156, 162, 163]
- [Rätsch et al., 2006] Rätsch, G., Sonnenburg, S., and Schäfer, C. (2006). Learning interpretable SVMs for biological sequence classification. *BMC Bioinformatics*, 7(S-1). [cited at page 111]
- [Rawlings et al., 2010] Rawlings, N. D., Barrett, A. J., and Bateman, A. (2010). MEROPS: the peptidase database. *Nucleic Acids Research*, 38:227–233. [cited at page 53]
- [Refaeilzadeh et al., 2009] Refaeilzadeh, P., Tang, L., and Liu, H. (2009). Cross-validation. In *Encyclopedia of Database Systems*, pages 532–538. Springer. [cited at page 41, 42]
- [Reinartz, 2002] Reinartz, T. (2002). A unifying view on instance selection. *Data Mining and Knowledge Discovery*, 6(2):191–210. [cited at page 105]
- [Ribeiro et al., 2010] Ribeiro, B., Lopes, N., and Silva, C. (2010). High-performance bankruptcy prediction model using graphics processing units. In *IEEE World Congress on Computational Intelligence (WCCI 2010)*. [cited at page 103]
- [Ribeiro et al., 2007] Ribeiro, B., Marques, A., Henriques, J., and Antunes, M. (2007). Choosing real-time predictors for ventricular arrhythmia detection. *International Journal of Pattern Recognition and Artificial Intelligence (IJPRAI)*, 21(8):1249–1263. [cited at page 55, 89]
- [Ribeiro et al., 2009] Ribeiro, B., Silva, C., Vieira, A., and ao Neves, J. (2009). Extracting discriminative features using non-negative matrix factorization in financial distress data. In *9<sup>th</sup> International Conference on Adaptive and Natural Computing Algorithms (ICANNGA 2009)*, LNCS 5495, pages 537–547. Springer Berlin Heidelberg. [cited at page 128, 131]
- [Richtárik et al., 2012] Richtárik, P., Takác, M., and Ahipasaoglu, S. D. (2012). Alternating maximization: Unifying framework for 8 sparse PCA formulations and efficient parallel codes. Cornell Universty Library. [cited at page 19]
- [Robilliard et al., 2009] Robilliard, D., Marion-Poty, V., and Fonlupt, C. (2009). Genetic programming on graphics processing units. *Genetic Programming and Evolvable Machines*, 10(4):447–471. [cited at page 17]
- [Roux and Bengio, 2008] Roux, N. L. and Bengio, Y. (2008). Representational power of restricted Boltzmann machines and deep belief networks. *Neural Computation*, 20(6):1631–1649. [cited at page 154, 155, 159, 160, 162]

- [Roux and Bengio, 2010] Roux, N. L. and Bengio, Y. (2010). Deep belief networks are compact universal approximators. *Neural Computation*, 22(8):2192–2207. [cited at page 155, 162]
- [Ryoo et al., 2008] Ryoo, S., Rodrigues, C. I., Baghsorkhi, S. S., Stone, S. S., Kirk, D. B., and Hwu, W. W. (2008). Optimization principles and application performance evaluation of a multithreaded GPU using CUDA. In *Proceedings of the 13<sup>th</sup> ACM Symposium on Principles and practice of parallel programming (PPoPP 2008)*, pages 73–82. [cited at page 22, 28, 75]
- [Samarasinghe, 2007] Samarasinghe, S. (2007). *Neural Networks for Applied Sciences and Engineering: From Fundamentals to Complex Pattern Recognition*. Auerbach Publications. [cited at page 62]
- [Schaa and Kaeli, 2009] Schaa, D. and Kaeli, D. (2009). Exploring the multiple-GPU design space. In *Proceedings of the 2009 IEEE International Symposium on Parallel & Distributed Processing (IPDPS 2009)*, pages 1–12. IEEE Computer Society. [cited at page 4, 16]
- [Schafer, 1999] Schafer, J. L. (1999). Norm: Multiple imputation of incomplete multivariate data under a normal model, version 2. <http://www.stat.psu.edu/~jls/misoftwa.html>. [cited at page 101]
- [Schafer and Graham, 2002] Schafer, J. L. and Graham, J. W. (2002). Missing data: our view of the state of the art. *Psychological Methods*, 7(2):147–177. [cited at page 92]
- [Schulz et al., 2010] Schulz, H., Müller, A., and Behnke, S. (2010). Investigating convergence of restricted boltzmann machine learning. In *NIPS 2010 Workshop on Deep Learning and Unsupervised Feature Learning, Whistler, Canada*. [cited at page 163]
- [Serpen, 2005] Serpen, G. (2005). A heuristic and its mathematical analogue within artificial neural network adaptation context. *Neural Network World*, 15(2):129–136. [cited at page 96]
- [Shalom et al., 2008] Shalom, S. A., Dash, M., and Tue, M. (2008). Efficient k-means clustering using accelerated graphics processors. In *Proceedings of the 10<sup>th</sup> International Conference on Data Warehousing and Knowledge Discovery (DaWaK 2008)*, LNCS 5182, pages 166–175. Springer. [cited at page 17]
- [Sharp, 2008] Sharp, T. (2008). Implementing decision trees and forests on a GPU. In *Proceedings of the 10<sup>th</sup> European Conference on Computer Vision (ECCV 2008)*, LNCS 5305, pages 595–608. Springer. [cited at page 17]

- [She et al., 2003] She, R., Chen, F., Wang, K., Ester, M., Gardy, J. L., and Brinkman, F. S. L. (2003). Frequent-subsequence-based prediction of outer membrane proteins. In *Proceedings of the 9<sup>th</sup> ACM SIGKDD international conference on Knowledge Discovery and Data Mining (KDD 2003)*, pages 436–445. [cited at page 111]
- [Shenouda, 2006] Shenouda, E. A. M. A. (2006). A quantitative comparison of different MLP activation functions in classification. In *Proceedings of the 3<sup>rd</sup> international conference on Advances in Neural Networks, LNCS 3971*, pages 849–857. Springer-Verlag. [cited at page 65]
- [Silva et al., 2009] Silva, M., Moutinho, L., Coelho, A., and Marques, A. (2009). Market orientation and performance: modelling a neural network. *European Journal of Marketing*, 43(3/4):421–437. [cited at page 74]
- [Skala, 2008] Skala, M. A. (2008). *Aspects of metric spaces in computation*. PhD thesis, University of Waterloo. [cited at page 194]
- [Sokolova and Lapalme, 2009] Sokolova, M. and Lapalme, G. (2009). A systematic analysis of performance measures for classification tasks. *Information Processing & Management*, 45(4):427–437. [cited at page 40]
- [Somorjai et al., 2011] Somorjai, R. L., Dolenko, B., Nikulin, A., Roberson, W., and Thiessen, N. (2011). Class proximity measures – dissimilarity-based classification and display of high-dimensional data. *Journal of Biomedical Informatics*, 44(5):775–788. [cited at page 194]
- [Sonnenburg et al., 2007] 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., and Williamson, R. C. (2007). The need for open source software in machine learning. *Journal of Machine Learning Research*, 8:2443–2466. [cited at page 17, 18]
- [Stamatopoulos et al., 2012] Stamatopoulos, C., Chuang, T. Y., Fraser, C. S., and Lu, Y. Y. (2012). Fully automated image orientation in the absence of targets. In *International Archives of the Photogrammetry, Remote Sensing and Spatial Information Sciences (XXII ISPRS Congress)*, volume Volume XXXIX-B5, pages 303–308. [cited at page 16]
- [Steinkraus et al., 2005] Steinkraus, D., Buck, I., and Simard, P. Y. (2005). Using GPUs for machine learning algorithms. In *Proceedings of the 2005 Eight International Conference on Document Analysis and Recognition (ICDAR 2005)*, volume 2, pages 1115–1120. [cited at page 16, 17]
- [Swersky et al., 2010] Swersky, K., Chen, B., Marlin, B., and de Freitas, N. (2010). A tutorial on stochastic approximation algorithms for training restricted



- Boltzmann machines and deep belief nets. In *Information Theory and Applications Workshop*, pages 1–10. [cited at page 155, 160, 161, 162, 163]
- [Tahir and Smith, 2010] Tahir, M. A. and Smith, J. (2010). Creating diverse nearest-neighbour ensembles using simultaneous metaheuristic feature selection. *Pattern Recognition Letters*, 31(11):1470–1480. [cited at page 69, 110]
- [Tang et al., 2007] Tang, H., Tan, K. C., and Yi, Z. (2007). *Neural Networks: Computational Models and Applications (Studies in Computational Intelligence)*, volume 53. Springer-Verlag. [cited at page 62]
- [Tang et al., 2003] Tang, H.-M., Lyu, M. R., and King, I. (2003). Face recognition committee machine. In *Proceedings of the 2003 IEEE International Conference on Acoustics, Speech, and Signal Processing (ICASSP 2003)*, volume 2, pages 837–840. [cited at page 147, 148, 150]
- [Tieleman, 2008] Tieleman, T. (2008). Training restricted Boltzmann machines using approximations to the likelihood gradient. In *Proceedings of the 25<sup>th</sup> international conference on Machine learning (ICML 2008)*, pages 1064–1071. [cited at page 161]
- [Trebatický and Pospíchal, 2008] Trebatický, P. and Pospíchal, J. (2008). Neural network training with extended kalman filter using graphics processing unit. In *Proceedings of the 18<sup>th</sup> International Conference on Artificial Neural Networks (ICANN 2008)*, LNCS 5164, pages 198–207. Springer. [cited at page 17]
- [Tuikkala et al., 2008] Tuikkala, J., Elo, L. L., Nevalainen, O. S., and Aittokallio, T. (2008). Missing value imputation improves clustering and interpretation of gene expression microarray data. *BMC bioinformatics*, 9(202):1–14. [cited at page 91, 93, 94, 96]
- [Vapnik, 1995] Vapnik, V. N. (1995). *The nature of statistical learning theory*. Springer-Verlag. [cited at page 111]
- [Čerňanský, 2009] Čerňanský, M. (2009). Training recurrent neural network using multistream extended kalman filter on multicore processor and CUDA enabled graphic processor unit. In *Proceedings of the 19<sup>th</sup> International Conference on Artificial Neural Networks (ICANN 2009)*, LNCS 5768, pages 381–390. Springer. [cited at page 17]
- [Verleysen, 2003] Verleysen, M. (2003). Learning high-dimensional data. In Ablameyko, S., Gori, M., Goras, L., and Piuri, V., editors, *Limitations and Future Trends in Neural Computation*, volume 186 of *NATO Science Series: Computer and Systems Sciences*, pages 141–162. IOS Press. [cited at page 127, 128]

- [Verleysen et al., 2009] Verleysen, M., Rossi, F., and François, D. (2009). Advances in feature selection with mutual information. In *Similarity-Based Clustering: Recent Developments and Biomedical Applications*, Lecture Notes in Artificial Intelligence 5400, pages 52–69. Springer-Verlag. [cited at page 127]
- [Vieira et al., 2009] Vieira, A. S., ao Duarte, J., Ribeiro, B., and ao C. Neves, J. (2009). Accurate prediction of financial distress of companies with machine learning algorithms. In *Proceedings of the 9<sup>th</sup> international conference on Adaptive and natural computing algorithms (ICANNGA 2009)*, LNCS 5495, pages 569–576. [cited at page 103]
- [Vonk et al., 1995] Vonk, E., Jain, L. C., and Veelenturf, L. P. J. (1995). Neural network applications. In *Electronic Technology Directions*, pages 63–67. [cited at page 62]
- [Žliobaitė, 2009] Žliobaitė, I. (2009). Combining time and space similarity for small size learning under concept drift. In *of the 18<sup>th</sup> International Symposium on Foundations of Intelligent Systems*, LNCS 5722, pages 412–421. [cited at page 49]
- [Wang et al., 2010] Wang, J., Zhang, B., Wang, S., Qi, M., and Kong, J. (2010). An adaptively weighted sub-pattern locality preserving projection for face recognition. *Journal of Network and Computer Applications*, 33(3):323–332. [cited at page 139]
- [Wang, 2005] Wang, S. (2005). Classification with incomplete survey data: a hopfield neural network approach. *Computers & Operations Research*, 32(10):2583–2594. [cited at page 96]
- [Wang, 2008] Wang, W. (2008). Some fundamental issues in ensemble methods. In *International Joint Conference on Neural Networks (IJCNN 2008)*, pages 2243–2250. [cited at page 69]
- [Widrow et al., 1994] Widrow, B., Rumelhart, D. E., and Lehr, M. A. (1994). Neural networks: applications in industry, business and science. *Communications of the ACM*, 37(3):93–105. [cited at page 62]
- [Wilson and Martinez, 2000] Wilson, D. R. and Martinez, T. R. (2000). Reduction techniques for instance-based learning algorithms. *Machine Learning*, 38(3):257–286. [cited at page 108, 112]
- [WolframAlpha, 2013] WolframAlpha (2013). WolframAlpha – computational knowledge engine. <http://www.wolframalpha.com>. [cited at page 55]
- [Wong et al., 2005] Wong, M., Wong, T., and Fok, K. (2005). Parallel evolutionary algorithms on graphics processing unit. In *Proceedings of the 2005 IEEE Congress on Evolutionary Computation*, volume 3, pages 2286–2293. [cited at page 17]

- [Wurst, 2007] Wurst, M. (2007). The word vector tool user guide operator reference developer tutorial. [cited at page 54]
- [Xiang et al., 2012] Xiang, X., Zhang, M., Li, G., He, Y., and Pan, Z. (2012). Real-time stereo matching based on fast belief propagation. *Machine Vision and Applications*, 23(6):1219–1227. [cited at page 19]
- [Xu et al., 2003] Xu, B., Lu, J., and Huang, G. (2003). A constrained non-negative matrix factorization in information retrieval. In *IEEE International Conference on Information Reuse and Integration (IRI 2003)*, pages 273–277. [cited at page 128]
- [Xu et al., 2009] Xu, Y., Chen, H., Klette, R., Liu, J., and Vaudrey, T. (2009). Belief propagation implementation using CUDA on an NVIDIA GTX 280. In *Proceedings of the 22<sup>nd</sup> Australasian Joint Conference on Advances in Artificial Intelligence (AI 2009)*, LNCS 5866, pages 180–189. Springer. [cited at page 17]
- [Yang et al., 2006] Yang, Q., Wang, L., Yang, R., Wang, S., Liao, M., and Nistér, D. (2006). Real-time global stereo matching using hierarchical belief propagation. In *Proceedings of the 2006 British Machine Vision Conference (BMVC 2006)*, volume 3, pages 989–998. [cited at page 17]
- [Ye, 2003] Ye, N., editor (2003). *The handbook of data mining*. Lawrence Erlbaum Associates. [cited at page 122]
- [Yu and Deng, 2011] Yu, D. and Deng, L. (2011). Deep learning and its applications to signal and information processing. *IEEE Signal Processing Magazine*, 28(1):145–154. [cited at page 154, 155]
- [Yu et al., 2005] Yu, Q., Chen, C., and Pan, Z. (2005). Parallel genetic algorithms on programmable graphics hardware. In *Proceedings of the 1<sup>st</sup> International Conference on Advances in Natural Computation (ICNC 2005)*, LNCS 3612, pages 1051–1059. Springer. [cited at page 17]
- [Yuksel et al., 2012] Yuksel, S. E., Wilson, J. N., and Gader, P. D. (2012). Twenty years of mixture of experts. *IEEE Transactions on Neural Networks and Learning Systems*, 23(8):1177–1193. [cited at page 70]
- [Yuming and Yuanyuan, 2012] Yuming, M. and Yuanyuan, Z. (2012). Research on method of double-layers BP neural network in bicycle flow prediction. In *International Conference on Industrial Control and Electronics Engineering (ICICEE 2012)*, pages 86–88. [cited at page 63]
- [Zapranis and Refenes, 1999] Zapranis, A. D. and Refenes, A.-P. (1999). *Principles of Neural Model Identification, Selection and Adequacy: With Applications to Financial Econometrics*. Perspectives in Neural Computing. Springer Verlag. [cited at page 123]

- [Zhang et al., 2009] Zhang, Y., Shalabi, Y. H., Jain, R., Nagar, K. K., and Bakos, J. D. (2009). FPGA vs. GPU for sparse matrix vector multiply. In *International Conference on Field-Programmable Technology (FPT 2009)*, pages 255–262. [cited at page 16]
- [Zhao et al., 2003] Zhao, W., Chellappa, R., Phillips, P. J., and Rosenfeld, A. (2003). Face recognition: A literature survey. *ACM Computing Surveys (CSUR)*, 35(4):399–458. [cited at page 139, 140]
- [Zhi et al., 2011] Zhi, R., Flierl, M., Ruan, Q., and Kleijn, W. B. (2011). 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. [cited at page 153]
- [Zhongwen et al., 2005] Zhongwen, L., hongzhi, L., Zhengping, Y., and Xincan, W. (2005). Self-organizing maps computing on graphic process unit. In *Proceedings of the 13<sup>th</sup> European Symposium on Artificial Neural Networks*, pages 557–562. [cited at page 15, 17]
- [Zhou, 2003] Zhou, Z.-H. (2003). Three perspectives of data mining. *Artificial Intelligence*, 143(1):139–146. [cited at page 104]
- [Zilu and Guoyi, 2009] Zilu, Y. and Guoyi, Z. (2009). Facial expression recognition based on NMF and SVM. In *Proceedings of the 2009 International Forum on Information Technology and Applications (IFITA 2009)*, volume 3, pages 612–615. IEEE Computer Society. [cited at page 57, 127, 128, 129]

- Accuracy, 39, 40
- Activation function, 64–66
- Adaptive step size, 8, 10, 68, 126, 127, 163, 170, 171, 174, 189, 192, 193
- ATS, 9, 11, 33, 62, 79, 80, 89–91, 103, 121, 123, 140, 146, 192
- Experimental results, 89–91, 142–145
- Autonomous Training System, *see* ATS
- 
- Back-Propagation, *see* BP
- Batch learning, 62, 105, 117, 121
- Benchmarks, *see* Datasets
- Bias-variance dilemma, 90
- Big Data, 1–4, 7, 8, 14, 16, 121, 189, 193
- BP, 5, 9, 11, 29, 33, 49, 62–69, 73, 74, 91, 99, 101, 102, 121, 147, 160, 163, 170, 177, 179, 182, 191, 192
- Experimental results, 79–91
- GPU Parallel Implementation, 75–78
- 
- Case study
- Financial distress, *see* Datasets - Financial distress
- Protein membership, *see* Datasets - Protein membership
- Ventricular arrhythmias, *see* Datasets - Ventricular arrhythmias
- CD-k, 160, 161, 163, 164, 170, 185, 186, 192, 193
- Classification problem, 5
- Compute Unified Device Architecture, *see* CUDA
- Concept drifts, 4, 105, 115, 123, 190
- Confusion matrix, 39, 40
- Contrastive divergence, *see* CD-k
- CPU, 14, 15, 141, 145–147, 149, 185
- Cross-validation, *see* Validation
- CUDA, 11, 16, 18–29, 31, 137, 186
- architecture, 20, 24–28
- blocks, 20–23, 25, 27
- built-in variables, 22
- coalesced accesses, 27–29, 31
- compute capability, 21, 24, 25, 27
- grid (kernel), 20–23, 25, 27
- kernels, 20–25, 31
- programming model, 16, 18, 20–24
- warp, 21, 22, 26–29
- Curse of dimensionality, 127
- 
- Datasets
- Annealing, 44, 100–102

AT&T, *see* Datasets - ORL face database  
Audiology, 44, 100, 101  
Bankruptcy, *see* Datasets - Financial distress  
Breast cancer, 44, 100, 101, 112, 113  
CBCL face database, 44, 46, 47, 57, 59, 140–143, 185  
Congressional, 44, 100–102  
Ecoli, 44, 112, 113  
elec2, *see* Datasets - electricity demand  
Electricity demand, 44, 46, 111, 115, 116, 190  
Financial distress, 10, 51, 53, 54, 96, 102–104, 122, 191  
Forest cover type, 44, 81, 84, 85  
German credit data, 44, 111–113  
Glass identification, 44, 112, 113  
Haberman’s survival, 44, 112, 113  
Heart - Statlog, 44, 112, 113  
Hepatitis, 44, 100–102  
HHreco multi-stroke symbol, 11, 12, 44, 46, 48, 127, 170, 174, 177–184, 186, 193  
Horse colic, 44, 100–102  
Ionosphere, 44, 112, 113  
Iris, 44, 112, 113  
Japanese credit, 44, 100–102  
KDD Cup 1999, 44, 46, 110, 113, 114, 190  
Luxembourg Internet usage, 44, 49, 111, 115, 116, 190  
Mammographic, 44, 100–102  
MNIST hand-written digits, 11, 12, 44, 49, 50, 125, 127, 157, 161, 167, 170–184, 186, 189, 192, 193  
Mushroom, 44, 100, 101  
ORL face database, 11, 43–45, 57, 58, 140, 145, 147–150, 152, 153, 185, 192

Peptidases, *see* Datasets - Protein membership  
Pima Indian diabetes, 44, 112, 113  
Poker hand, 44, 81, 84–86  
Protein membership, 10, 51, 53, 54, 111, 117, 119, 123, 190  
Sinus cardinalis, 44, 49, 51, 81, 82  
Sonar, 44, 81, 89, 112, 113  
Soybean, 44, 100–102  
Tic-Tac-Toe, 44, 111–113  
Two-spirals, 44, 49, 51, 81, 82  
Vehicle, 44, 112, 113  
Ventricular arrhythmias, 10, 11, 51, 54, 55, 81, 85, 87–89, 91, 121, 192  
Wine, 44, 112, 113  
Yale face database, 11, 44, 49, 51, 52, 57, 60, 132, 140, 142, 144–146, 149, 151, 153, 185, 192  
Yeast, 44, 112, 113  
DBN, 8, 10–12, 33, 46, 57, 126, 127, 155–164, 170, 174, 177, 179, 180, 182, 185–187, 189, 192–194  
Experimental results, 170–182  
GPU parallel implementation, 163–170  
Deep belief networks, *see* DBN  
Deep learning, *see also* DBN, 153–155  
Empirical risk minimization, 5, 6  
F<sub>1</sub> score, *see* F-Measure  
F-measure, 39–41  
Face recognition, 10–12, 43, 57, 126, 128, 131, 139, 140, 182  
Feature extraction, 7, 128, 182, 193  
Feed-forward network, 63, 64, 66  
Field-Programmable Gate Array, *see* FPGA  
FPGA, 15, 16  
General-Purpose computing on GPUs, *see* GPGPU

- 
- Generalization, 41, 55, 56, 69, 90, 95, 96
- Gibbs sampling, 159, 160
- GPGPU, 14, 16, 19, 20
- GPU, 4, 9–22, 24–29, 35, 126, 127, 131, 134, 140–142, 145–147, 149, 165, 167, 170, 185, 186
- Pipeline, 19
- GPU computing, *see also* GPGPU, 4, 14, 16, 20, 33
- GPULib, 9, 11, 13, 14, 18, 19, 28–35, 191, 193, 194
- Graphics Processing Units, *see* GPU
- Histogram Equalization, 57, 141
- IB3, 108, 110–113, 115
- IHC, 8, 10, 11, 42, 62, 106–110, 121–123, 190, 193, 194
- Experimental results, 110–121
- IHC-SVM, 10, 117–121, 123, 190, 191
- Imputation, 92, 94–96, 191
- Incremental Hypersphere Classifier, *see* IHC
- Incremental learning, 4, 62, 105, 117, 118, 121, 123
- Instance selection, 4, 39, 105
- Interpretability, 105, 123
- k-nearest neighbor, *see* k-nn
- k-nn, 106, 110–112
- Machine Learning, 1–5, 7, 9, 11, 13–15, 17–19, 29, 30, 33, 35, 40, 53, 61, 92–94, 96, 104, 105, 123, 126, 131, 153, 185, 189, 191, 193, 194
- Macro-average
- F-Measure, 41
- Precision, 41
- Recall, 41
- MAR, 93, 94, 96, 99, 122, 191
- Markov Chain Monte Carlo, 159, 194
- MBP, 9, 11, 29, 33, 62, 69–75, 79, 91, 99, 101–103, 121, 122, 140, 142, 146, 165, 170, 177, 179, 182, 185–187, 191–193
- Experimental results, 79–91
- GPU Parallel Implementation, 75–78
- MCAR, 93, 94, 99, 122, 191
- MCMC, *see* Markov Chain Monte Carlo, 160
- MFF, 71–74
- min-max rescaling, *see* Rescaling
- Missing data, 5, 8, 9, 62, 91–103, 121, 122, 191, 192
- mechanisms, 93–94
- methods, 94–97
- Missing values, *see* Missing data
- Missing values problem, *see* Missing data
- MLP, *see* Feed-forward network
- Multi-layer perceptron, *see* Feed-forward network
- Multiple Back-Propagation, *see* MBP
- Multiple back-propagation software, 79, 99
- Multiple feed-forward network, *see* MFF
- Neural networks, 8, 9, 39, 62–104, 106, 121–123, 128, 144, 146, 147, 153–182, 191, 192
- Neural Selective Input Model, *see* NSIM
- Neuron, 63–65
- selective actuation, 70–73, 97
- selective input, 97, 98, 122
- NMAR, 93, 94
- NMF, 8, 10, 12, 33, 43, 57, 126–132, 134, 139–142, 146, 148–150, 153, 182, 185, 187, 190, 192, 194
- Combining with other algorithms, 131
- Experimental results, 139–153
-

- GPU parallel implementation, 134–139
- Non-Negative Matrix Factorization, *see* NMF
  - semi-supervised, *see* SSNMF
- NSIM, 8–11, 33, 62, 97–99, 121–123, 191–193
  - experimental results, 99–104
  - GPU Parallel Implementation, 99
- Open source, 17, 18
- Precision, 39–41
- Preprocessing, 55–57, 94, 97
- RBF, 33, 70, 117
- RBM, 8, 10, 33, 57, 126, 155–166, 170, 171, 174, 177, 179, 182, 185, 186, 189, 192–194
  - Experimental results, *see* DBN - Experimental results
  - GPU parallel implementation, *see* DBN - GPU parallel implementation
- Recall, 39–41
- Reinforcement learning, 5, 6
- Rescaling, 56, 57
- Restricted Boltzmann machines, *see* RBM
- RMSE, 39, 76, 78, 83
- Root mean square error, *see* RMSE
- Scalar Processor, 24–27
- Semi-supervised learning, 5
- Sensitivity, 39, 40, 89
- SIMT, 27
- Single-instruction multiple-thread, *see* SIMT
- Space savings, *see* Storage reduction
- Specificity, 39, 40
- Speedup, 39
- SSNMF, 8, 12, 126, 131–134, 140, 148–150, 185, 187, 190, 192–194
  - Experimental results, 140, 148–153
- Storage reduction, 39
- Stratification (data), 42, 43
- Streaming Multiprocessor, 24–27
- Structural risk minimization, 5
- Supervised learning, 2, 5–7, 61, 63, 125, 126, 182
- SVM, 5, 10, 11, 33, 92, 111, 117, 118, 120, 123, 147–149, 154, 155, 185, 190
- Test dataset, 41, 56
- Train dataset, 5, 6, 41, 56
- True negative rate, *see* Specificity
- True positive rate, *see* Sensitivity
- Unsupervised learning, 5–7, 125, 126, 182
- Validation
  - hold-out, 41
  - k-fold cross-validation, 42
  - leave-one-out cross-validation, 42, 43
  - leave-one-out-per-class cross-validation, 43
  - repeated k-fold cross-validation, 42
  - repeated random sub-sampling validation, 43



