Modifications To The Fuzzy-ARTMAP Algorithm For Distributed Learning In Large Data Sets

2004

Jose R. Castro

University of Central Florida

Find similar works at: http://stars.library.ucf.edu/etd

University of Central Florida Libraries http://library.ucf.edu

Part of the Electrical and Computer Engineering Commons

STARS Citation

Castro, Jose R., "Modifications To The Fuzzy-ARTMAP Algorithm For Distributed Learning In Large Data Sets" (2004). Electronic Theses and Dissertations. 5.
http://stars.library.ucf.edu/etd/5

This Doctoral Dissertation (Open Access) is brought to you for free and open access by STARS. It has been accepted for inclusion in Electronic Theses and Dissertations by an authorized administrator of STARS. For more information, please contact lee.dotson@ucf.edu.
Modifications of the Fuzzy–ARTMAP Algorithm For Distributed Learning in Large Data Sets

by

JOSÉ CASTRO
M.Sc. Instituto Tecnológico de Costa Rica, 2000
Lic. Computer Science, Universidad de Costa Rica, 1992

A dissertation submitted in partial fulfillment of the requirements for the degree of Doctor of Philosophy in the School of Electrical and Computer Engineering in the College of Engineering and Computer Science at the University of Central Florida Orlando, Florida

Spring Term
2004

Major Professor:
Michael Georgiopoulos
The Fuzzy-ARTMAP (FAM) algorithm has been proven to be one of the premier neural network architectures for classification problems. FAM can learn on line and is usually faster than other neural network approaches. Nevertheless the learning time of FAM can slow down considerably when the size of the training set increases into the hundreds of thousands. In this dissertation we apply data partitioning and network partitioning to the FAM algorithm in a sequential and parallel setting to achieve better convergence time and to efficiently train with large databases (hundreds of thousands of patterns).

We implement our parallelization on a Beowulf clusters of workstations. This choice of platform requires that the process of parallelization be coarse grained. Extensive testing of all the approaches is done on three large datasets (half a million data points). One of them is the Forest Covertype database from Blackard and the other two are artificially generated Gaussian data with different percentages of overlap between classes.

Speedups in the data partitioning approach reached the order of the hundreds without having to invest in parallel computation. Speedups on the network partitioning approach are close to linear on a cluster of workstations. Both methods allowed us to reduce the computation time of training the neural network in large databases from days to minutes. We prove formally that the workload balance of our
network partitioning approaches will never be worse than an acceptable bound, and also demonstrate the correctness of these parallelization variants of FAM.
To my unborn child, because the future belongs to you

—

Reading makes the full man
Speaking makes the ready man
Writing makes the exact man

— Francis Bacon.
ACKNOWLEDGMENTS

I would like to thank the Technological Institute of Costa Rica for giving me the initial financial support to embark on this journey. Dr. Avelino Gonzalez for coming up with a feasible plan to attract Latin American instructors to UCF, The University of Central Florida for providing the resources necessary to work on this dissertation, not least of which is the vision to equip itself with a Beowulf cluster of computers. The LINK foundation for believing in the proposal enough to grant me the 2002 scholarship in advanced simulation and training; without it I could not have dedicated myself the way I did to my research. My wife Leonora and 3 children: Camilo, Federico and Ximena who have patiently accepted my recluse during the last few months. And last but certainly not least, Dr. Michael Georgiopoulos, my advisor, who’s knowledge, patience, support, optimism and guidance have sharpened and focused the direction and vision of this work. It is by his constant example that I have learned that optimism and an impeccable work ethic are the fundamental ingredients of any fruitful research endeavor.
TABLE OF CONTENTS

LIST OF TABLES ............................................................. x

LIST OF FIGURES ........................................................... xi

CHAPTER 1 INTRODUCTION .................................................. 1
  1.1 Machine Learning Generalities ........................................ 1
      1.1.1 Online and Offline learning .................................... 2
      1.1.2 Supervised and UN-supervised learning ....................... 3
      1.1.3 Function approximation and Classification .................. 3
  1.2 Pattern Recognition ..................................................... 4
  1.3 Neural Networks ........................................................ 6
  1.4 Managing Large Training Sets ...................................... 8
  1.5 Parallelism and Distributed Processing ............................ 9
  1.6 Organization ............................................................ 12

CHAPTER 2 FUZZY-ART AND FUZZY-ARTMAP ............................ 13
  2.1 Fuzzy-ART ............................................................. 13
      2.1.1 Fuzzy-ART Learning and Classification ..................... 17
2.1.2 Fuzzy-ART pseudo-code ........................................ 18

2.2 Fuzzy-ARTMAP ....................................................... 20
  2.2.1 The Fuzzy-ARTMAP Architecture ............................ 21
  2.2.2 The Fuzzy-ARTMAP Learning Algorithm .................. 25
  2.2.3 Fuzzy-ARTMAP pseudo-code ................................. 27

2.3 Fuzzy-ARTMAP properties ......................................... 28

2.4 Geometric interpretation of Fuzzy-ARTMAP ..................... 35

2.5 Fuzzy-ARTMAP time complexity analysis ...................... 40
  2.5.1 Online Fuzzy-ARTMAP complexity .......................... 41
  2.5.2 Off line Fuzzy-ARTMAP complexity ......................... 42

CHAPTER 3 PARALLEL PROCESSING .................................... 46
  3.1 Types of Parallel Machines ..................................... 46
    3.1.1 Flynn’s Classification ....................................... 47
  3.2 Message-Passing Architectures and Techniques ............... 52
    3.2.1 Network Topologies .......................................... 52
  3.3 Clusters of Workstations and the Scerola cluster ........... 58
  3.4 Speedup, Efficiency and Cost .................................. 58

CHAPTER 4 PROPOSAL .................................................... 62
  4.1 Literature Review ................................................ 62
  4.2 Justification ...................................................... 65
<table>
<thead>
<tr>
<th>Section</th>
<th>Title</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>6.2</td>
<td>The FAM pseudo-code</td>
<td>121</td>
</tr>
<tr>
<td>6.3</td>
<td>Anagnostopoulos “No Matchtracking” FAM</td>
<td>125</td>
</tr>
<tr>
<td>6.4</td>
<td>The Beowulf parallel platform</td>
<td>126</td>
</tr>
<tr>
<td>6.5</td>
<td>Parallel, no matchtracking, FAM implementation</td>
<td>129</td>
</tr>
<tr>
<td>6.6</td>
<td>Properties of the Parallel, no matchtracking, FAM Algorithm</td>
<td>136</td>
</tr>
<tr>
<td>6.7</td>
<td>Matchtracking Parallel FAM implementation</td>
<td>152</td>
</tr>
<tr>
<td>6.8</td>
<td>Experiments</td>
<td>153</td>
</tr>
<tr>
<td>6.8.1</td>
<td>Forest Covertype Database</td>
<td>157</td>
</tr>
<tr>
<td>6.9</td>
<td>conclusions</td>
<td>161</td>
</tr>
</tbody>
</table>

**CHAPTER 7 CONCLUSIONS** | 162 |

**APPENDIX A PARAMETER ESTIMATION** | 166 |

**APPENDIX B SEQUENTIAL MODIFICATIONS** | 170 |

**REFERENCES** | 182 |
# LIST OF TABLES

<table>
<thead>
<tr>
<th>Table</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>6.1</td>
<td>No-matchtracking FAM CORRECTNESS theorems</td>
<td>137</td>
</tr>
<tr>
<td>6.2</td>
<td>No-matchtracking FAM PERFORMANCE theorems</td>
<td>137</td>
</tr>
<tr>
<td>6.3</td>
<td>Training time in seconds of Forest Covertype database</td>
<td>158</td>
</tr>
<tr>
<td>6.4</td>
<td>Classification performance of no-matchtracking <em>Fuzzy</em>-ARTMAP</td>
<td>158</td>
</tr>
</tbody>
</table>
LIST OF FIGURES

2.1 Block Diagram of the Fuzzy-ART Architecture . . . . . . . . . . . . . 14
2.2 Fuzzy–ART off–line training phase pseudocode . . . . . . . . . . . . . 19
2.3 Fuzzy–ART online training phase pseudocode . . . . . . . . . . . . . 20
2.4 Fuzzy–ART performance phase pseudocode . . . . . . . . . . . . . . 21
2.5 Block Diagram of the Fuzzy–ARTMAP Architecture . . . . . . . . . . . 23
2.6 Fuzzy–ARTMAP off–line training phase pseudocode . . . . . . . . . . 29
2.7 Fuzzy–ARTMAP online training phase pseudocode . . . . . . . . . . . 30
2.8 Fuzzy–ARTMAP performance phase pseudocode . . . . . . . . . . . . 31
2.9 Fuzzy–ARTMAP Geometric interpretation in 2 dimensions . . . . . . 37

3.1 Shared memory multiprocessor model . . . . . . . . . . . . . . . . . . 49
3.2 Message passing multiprocessor model . . . . . . . . . . . . . . . . . . 50
3.3 Distributed shared memory multiprocessor model . . . . . . . . . . . . 51
3.4 Fully connected Network Architecture . . . . . . . . . . . . . . . . . . 53
3.5 Line Network Architecture . . . . . . . . . . . . . . . . . . . . . . . . . 53
3.6 Star Network Architecture . . . . . . . . . . . . . . . . . . . . . . . . . 54
3.7 Tree Network Architecture . . . . . . . . . . . . . . . . . . . . . . . . . 55
List of Symbols

<table>
<thead>
<tr>
<th>Symbol</th>
<th>definition</th>
</tr>
</thead>
<tbody>
<tr>
<td>$\rho$</td>
<td>vigilance parameter of Fuzzy−ART, $\rho \in [0, 1]$.</td>
</tr>
<tr>
<td>$\rho_a$</td>
<td>baseline vigilance parameter of Fuzzy−ARTMAP, $\rho_a \in [0, 1]$.</td>
</tr>
<tr>
<td>$\beta$</td>
<td>choice parameter of Fuzzy−ART, $\beta &gt; 0$.</td>
</tr>
<tr>
<td>$\beta_a$</td>
<td>choice parameter of Fuzzy−ARTMAP, $\beta_a &gt; 0$.</td>
</tr>
<tr>
<td>$\varepsilon$</td>
<td>small positive constant.</td>
</tr>
<tr>
<td>$w, w_j$</td>
<td>templates in the Fuzzy−ART neural network.</td>
</tr>
<tr>
<td>$w^a_j$</td>
<td>template in the Fuzzy−ARTMAP neural network.</td>
</tr>
<tr>
<td>$w_0$</td>
<td>uncommitted node.</td>
</tr>
<tr>
<td>$T(I', w, \beta)$</td>
<td>activation of the Fuzzy−ART nose with template $w$, due to input $I'$.</td>
</tr>
<tr>
<td>$\rho(I, w)$</td>
<td>vigilance ratio of Fuzzy−ART node with template $w$, due to input $I'$.</td>
</tr>
<tr>
<td>$\Gamma$</td>
<td>repeat factor of matchtracking.</td>
</tr>
<tr>
<td>$\kappa$</td>
<td>compression ratio.</td>
</tr>
<tr>
<td>$T_1$</td>
<td>sequential time for Fuzzy−ARTMAP training.</td>
</tr>
<tr>
<td>$T^p_1$</td>
<td>sequential time for Fuzzy−ARTMAP training with $p$ partitions.</td>
</tr>
<tr>
<td>$T^p_n$</td>
<td>parallel time for Fuzzy−ARTMAP training with $p$ partitions in $n$ processors.</td>
</tr>
<tr>
<td>$P$</td>
<td>number of training patterns in Fuzzy−ART or training pairs in Fuzzy−ARTMAP.</td>
</tr>
<tr>
<td>$M$</td>
<td>dimensionality of input patterns in Fuzzy−ART.</td>
</tr>
<tr>
<td>$M_a$</td>
<td>dimensionality of the input patterns in Fuzzy−ARTMAP.</td>
</tr>
<tr>
<td>$m$</td>
<td>order of approximation of a space−filling curve.</td>
</tr>
<tr>
<td>$N$</td>
<td>number of bits in a derived key of a Hilbert index.</td>
</tr>
<tr>
<td>$r$</td>
<td>an $N$−Bit binary Hilbert derived−key.</td>
</tr>
<tr>
<td>$\gamma^i_j$</td>
<td>a binary digit in $r$.</td>
</tr>
<tr>
<td>$\gamma^i$</td>
<td>$i^{th}$ binary byte in $r$.</td>
</tr>
<tr>
<td>$a_j$</td>
<td>a coordinate in dimension $j$ of the point $(a_1, a_2, \ldots, a_j, \ldots, a_{M_a})$.</td>
</tr>
<tr>
<td>$\alpha^i_j$</td>
<td>a binary digit in a coordinate $a_j$.</td>
</tr>
<tr>
<td>$\alpha^i$</td>
<td>a concatenation of all the $i^{th}$ entries of the $a_j$’s.</td>
</tr>
</tbody>
</table>
CHAPTER 1
INTRODUCTION

Fuzzy–ARTMAP Neural Networks have desirable properties like on line learning and have been proven to be good classifiers when the amount of classification patterns and features are in the hundreds. Their performance is known to lag once this measure grows. Comparable complexity algorithms like C4.5 have been adapted to handle very large data sets by applying algorithm modification, data structure and parallelization techniques. Fuzzy–ARTMAP is amenable to these approaches. In this dissertation we outline 2 data partitioning and 2 network partitioning methods to achieve this. Experimental results have shown that both data partitioning and network partitioning variants of the Fuzzy–ARTMAP algorithm have been able to bring down the training time in large databases from days to minutes.

1.1 Machine Learning Generalities

The ability for a machine to learn from experience is an undoubtedly useful feature that we would like many computer systems to possess. Ever since the appearance and coining of the term Artificial Intelligence in 1956 by John McCarthy [Dar56] many systems have tried to emulate the properties of an intelligent system. One of the properties that is commonly associated with intelligent behavior is the capacity to
learn from experience or to extract information from data. This topic, called *machine learning* is commonly recognized as an important attribute of Intelligent Systems and has not only become an important field of research within the Artificial Intelligence community, but also an independent field of research in its own right.

In machine learning, a set of examples called the *training set* are taken from an input *feature space* and presented to a *learning algorithm*. The learning algorithm develops an internal representation that abstracts information from the set. This abstracting of information usually, but not necessarily, implies compression of the information, and happens as examples are coded in a format that can be anything from neural network weights to rules in a rule base. This process of learning associations present in the training set is called *training*, and from now on we will use the terms learning and training interchangeably.

### 1.1.1 Online and Offline learning

When the training is accomplished from a continuous potentially infinite input stream of examples the algorithm is said to proceed *online*. If, on the other hand, the training is accomplished in batch mode on a given database of examples the training is said to proceed in an *offline* fashion. Some machine learning algorithms have the capacity to learn online, while others require the training set to be specified before learning is commenced.

If the algorithm is offline, then it’s functioning is divided into a *learning phase* and a *performance phase*. When the algorithm is in the learning phase it is presented with the training set, and the learning algorithm then uses this training set to learn the
required task. If the algorithm is in the performance phase it will be presented with an example (not necessarily from the training set) and what is expected of it is to respond appropriately to the input by producing a pertinent output. Given these definitions the previously mentioned online algorithms can alternatively be characterized as algorithms in which the training phase and performance phase may overlap.

1.1.2 Supervised and UN-supervised learning

Learning can also be supervised or UN-supervised. In supervised learning each element of the training set is called a training pair and is composed of an element of the input feature space and it’s corresponding desired output response. When performing supervised learning the object of the learning algorithm can be stated broadly as that of finding an internal representation that can map the input patterns to the specified output values. On the other hand, when performing UN-supervised learning, given that no explicit response is specified for the algorithm, the algorithm usually performs the task of agglomerating the input data into clusters. These clusters contain input patterns that are close to each other by some predefined measure of similarity or distance.

1.1.3 Function approximation and Classification

The nature of the problem to be learned also affects the learning algorithm used. When doing supervised learning the type of problems can further be divided into
Function approximation problems and classification problems. Function approximation problems are trained with input patterns that are paired with output values. These are assumed to be examples of a continuous function that has to be learned by the algorithm. Since the training set is finite and the function assumes values over a continuum the training set cannot cover all the possible outputs of the function. Under this situation the algorithm must be able to interpolate values in between the example points of the training set.

Classification problems on the other hand map inputs onto a discrete set of labels, so no interpolation of outputs is necessary. On both classification and function approximation problems the process of responding to previously unseen patterns is called generalization and is an important measure of the quality of a machine learning algorithm.

The representation of the training patterns varies depending on the nature of the algorithm used. Symbolic algorithms like C4.5 and FOIL [Die97] maintain the representation of the original problem for the input patterns, usually as typed n-tuples of data taken directly for a database table. Statistical and connectionist or neural network algorithms map the original representation onto a vector in \( D^n \) where \( D \) is usually the real interval \([0, 1]\), or the set \( \{0, 1\} \) if the domain is discrete.

1.2 Pattern Recognition

A special case of supervised learning that we will be interested in is recognition or classification. In this learning scheme inputs are paired with labels that represent the input class of the patterns and the object of the algorithm is to correctly classify
the inputs. An example of pattern recognition is the research done on handwritten character recognition for automatic ZIP code identification by Le Cun et al [CBD89].

Some classification algorithms are statistical in nature. These methods assume that there’s an underlying probability distribution function for the input domain that characterizes the distribution of the input patterns. Statistical classifier methods are called parametric if a specific distribution function (i.e., normal distribution) is assumed for the input domain. It will be a non-parametric method if the distribution for the input is assumed unknown or unspecified. An example of a parametric statistical learning algorithm is the Bayesian family of classifiers (Duda & Hart, 1973).

Most statistical methods work by estimating parameters from the assumed to exist probability distribution function. These methods have the strength that by assuming a distribution that the input data obey end up with an analytical formula for classification, and some of them can be proven optimal if the conditions which they require are satisfied (i.e., the simple Bayes classifier). These methods also have the advantage of functioning very well in the presence of random noise, since random elements in the input will probably be canceled out in the estimation of parameters.

Other methods do not rely on the assumption of a distribution function for the input domain. An example is the $k$-Nearest Neighbor ($k$NN) algorithm which classifies using a majority voting scheme from the $k$ nearest input patterns (this method does not provide compression). This method has been successfully used in complex document classification tasks by Han et al [HKK01]. Other examples are the decision tree algorithms ID3 and its successor C4.5 developed by Quinlan, both of which rely on information theoretic principles to generate a decision tree for the input domain [Qui93].
1.3 Neural Networks

One of the most popular machine learning approaches today for pattern recognition classification and function approximation problems is the neural network approach. When using neural networks for classification, a network of simple processors called neurons are assembled to perform the classification task. Each neuron receives as input a list of numeric values and produces a numeric output. The output of each neuron can feed as input to other neurons weighted appropriately through interconnection weights. The neural network paradigm is massively parallel in nature although it is not a requirement that it be implemented as such. In neural networks the knowledge about the problem is stored in the interconnection weights. Neural Network architectures usually come hand in hand with their specific learning algorithms, so much so that in the following we will refer interchangeably to the neural network architecture and it’s corresponding learning algorithm.

There are two basic types of neural network architectures, feed forward neural network architectures and recurrent neural network architectures.

An example of a well known neural network model is the Multi-Layer Perceptron (MLP), which uses the Backpropagation learning algorithm. This model was popularized by the hallmark 1986 paper Learning Internal Representations by Error Propagation \[\text{[RHW86]}\] by Rumelhart et al. In spite of it’s overwhelming popularity back-propagation has some drawbacks, such as:

- it cannot learn online.
- it’s a gradient descent algorithm and as such it is at times notoriously slow in converging to a solution.
it suffers from the problem of local minima which can impede the algorithm from finding an acceptable solution to the learning problem.

- the network architecture and learning parameters have to be guessed by the designer.

Other examples of neural network approaches that have had a good measure of success are the Radial Basis function neural networks, Probabilistic neural networks, Hopfield neural networks, Kohonen Self Organizing Map, and Adaptive Resonance Theory (ART) based neural networks.

ART based neural networks comprise a large family of related neural network paradigms. Specifically, Fuzzy–ARTMAP is the main focus of our research. A description of them with their variants will be the subject of the next chapter. For now, it will suffice to state that ART architectures exhibit incremental on-line learning capabilities, their learning process is guaranteed to converge to a solution, they possess a novelty detector feature that recognizes novel inputs (inputs that are significantly different from inputs that the architecture has seen before). They can, under analysis, provide explanations for the answers that they produce, and as a result they can address the black box problem that most neural networks have been criticized for [CT95]. Finally, these architectures have the property that they can dynamically increase their size as the learning process progresses and only when the learning task at hand requires it, a characteristic that eliminates the need to specify an arbitrary neural network architecture prior to the learning process.

However, ART neural networks also have their drawbacks. They can sometimes train very slowly when the size of the training set is large, and they occasionally produce large amount of network nodes compared with other neural network archi-
tectures. Addressing the first of these issues by relying on parallel processing will be the main topic of this dissertation.

1.4 Managing Large Training Sets

Most of the machine learning algorithms presented in the previous section have been designed for use when the number of training patterns is in the hundreds to a few thousands and the number of input features in the input domain falls in the tens to hundreds. Also all of these algorithms assume that the training set and internal representation produced by the algorithm will fit into the machine's main memory. These assumptions make the data structure representation of the problem a rather straightforward task. However, most of the aforementioned algorithms suffer severe performance deterioration when the number of training pairs or/and the number of input features are increased into the many thousands or hundreds of thousands. This situation will happen in certain problem domains, like information retrieval: were the task at hand is the pertinent retrieval from a database of documents the set of documents that are relevant to a particular database query. In these circumstances the number of documents to classify can be in the hundreds of thousands and the number of features can also be in the thousands since in some information retrieval tasks each word in a document is potentially a feature in the input domain. ART neural networks are particularly susceptible to this problem since they occasionally suffer from the aforementioned node proliferation problem. On the other hand, as mentioned in the previous section, ART neural networks have desirable characteristics like online learning that the other algorithms do not possess. Hence, it’s valid to raise
the question of whether there exists a way that we can modify the ART algorithms so that they will be able to handle efficiently large and huge data sets.

In order to handle large databases we can identify the following approaches:

- New algorithms can be proposed that are simpler and therefore consume less processing time.
- More efficient data structures can be used to represent the information and therefore:
  - reduce the execution time and
  - allow more information to be stored in main memory efficiently.
- The algorithm can be partitioned and run in parallel. This implies the distribution of the algorithmic task onto a set of processors.

The last point in the previous list merits a section by itself before proceeding with the statement of purpose of this dissertation.

1.5 Parallelism and Distributed Processing

When talking about parallel and distributed processing systems, a very coarse taxonomy can be was established by Flynn in 1966 [Fly66] for the different types and levels of parallel architectures available, ranging from no parallelism at all to highly parallel systems. This taxonomy is:
• SISD (Single Instruction Single Data machine): In this category we can find small microprocessors like the 8086. Today’s Pentium line of processors would not formally fall into this category since it already has multiple threads of execution with an incorporated mathematical coprocessor, but by most practical standards it is considered a SISD machine.

• SIMD (Single Instruction Multiple Data machine): In this category we can find vector machines that perform graphics and signal processing tasks.

• MISD (Multiple Instruction Single Data machine): This category is not normally implemented unless we count in this category the pipeline architecture where a stream of data is processed using a sequence of instructions.

• MIMD (Multiple Instruction Multiple Data machine): In this category we can find networks of workstations and more decoupled systems.

It is common to use the term Parallelism when the processing is being done within the same hardware chip. This obviously requires special purpose hardware and is more common to be done with the SIMD model.

The term Distributed processing is more commonly applied for the distribution of a process (in our case the Fuzzy ARTMAP algorithm) between a set of independent but networked machines. Distributed processing does not require special purpose hardware and can be achieved in a normal network of computers. Distributed processing is a type MIMD but it is common that identical processes run on all the machines, making it in that case a kind of simulation of the SIMD model.

Given a distributed approach there are many different types of topology that can be used to implement the network, some of these are ring, star, mesh, torus, tree,
complete. Within these topologies there are many variants on the network protocols used for communication, some of these are packet switched, connection switched, broadcast, datagram and others. It is not the purpose of this proposal to go into the details of network architectures and protocols but we will propose and test the Fuzzy ARTMAP algorithm in a pipelined computer network architecture that we will show maps naturally into the neural network architecture.

Also, communication between processes can be achieved using messages or shared memory. By interrupt or by hand shaking. If using the shared memory paradigm synchronization and mutual exclusion become an issue, it can be achieved by semaphores, monitors or by some other mechanism like Unix’s wait and signal.

Different considerations apply when developing applications for a parallel processing hardware than when developing for a distributed network of workstations. The most striking difference concerns communication costs between processing entities, which in networks is orders of magnitude slower than in a parallel hardware chip. For example, in a 100Mb sec network the theoretical lower bound on the transfer of 32 bit word from machine to machine will be .3 micro seconds while in a vector machine it will be in the order of nano-seconds. The practical lower bound is much larger than the result calculated here due to network bandwidth lost to communication overhead.

The main advantages of a distributed approach is that it can benefit from standard and easily available hardware, and that given a good software design a distributed approach is more easily scaled and ported to other sites, and therefore has a higher reutilization potential.
1.6 Organization

This dissertation is organized as follows. Chapter 2 deals with the Fuzzy–ART (FA) and Fuzzy–ARTMAP (FAM) family of algorithms, their explanation and how they pertain to our research. Algorithms are provided for the different variants of training and execution, this sets the stage for the modifications to the algorithm that will be presented later. Chapter 3 discusses parallel processing and different models of parallelization, focusing by the end of the chapter on the Beowulf cluster of workstations platform, which is the platform of choice for the implementation of the parallel algorithms presented here. Chapter 4 discusses the current literature relevant to the topic of interest and outlines the main ideas and proposals of this dissertation. Chapter 5 explains in detail the data partitioning approaches for FAM problem subdivision. Chapter 6 focuses on different FAM parallelization approaches that relies on network partitioning for task subdivision and workload balance. Finally, chapter 7 wraps up the dissertation by providing the conclusions and future directions of this work.
CHAPTER 2
FUZZY-ART AND FUZZY-ARTMAP

2.1 Fuzzy-ART

Fuzzy-ART (FA) is an unsupervised learning algorithm that clusters the input domain. It was introduced by Carpenter et al in 1991 [CGR91]. This algorithm extends the ART1 architecture, which was originally designed to process only binary data, by accepting analog inputs and outputs. The FA network consists of three layers and is divided into an attention subsystem and an orientation subsystem. A diagram of the Fuzzy-ART architecture can be seen in figure 2.1.

The assumption here is that the input vector \( \mathbf{a} \) is such that each one of its components lies in the interval \([0, 1]\). The first layer \( F_0 \) receives as input a pattern vector \( \mathbf{a} \) of dimensionality \( M \) and produces as output a complement coded vector \( \mathbf{I} = (\mathbf{a}, \mathbf{a}^c) \) where

\[
\mathbf{I} = (\mathbf{a}, \mathbf{a}^c) = (a_1, a_2, \ldots, a_M, a_1^c, a_2^c, \ldots, a_M^c)
\]  
(2.1)

and

\[
a_i^c = 1 - a_i; \quad \forall i \in \{1, 2, \ldots, M\}
\]  
(2.2)

The output of the \( F_0 \) layer feeds as input to the \( F_1 \) layer. There is an all to all connectivity between the \( F_1 \) layer and the \( F_2 \) layer. The connection weights from

13
Field $F_0$ - $I = (a, a^c)$
Field $F_1$ - $w_j$
Field $F_2$ - $W_j$

Attentional Subsystem

Orienting Subsystem

reset node

Figure 2.1: Block Diagram of the Fuzzy-ART Architecture
the $F_1$ to $F_2$ are called bottom up weights and the connections from the $F_2$ to $F_1$ are called top down weights.

Of the aforementioned two sets of weights only the top down weights are sufficient in explaining the FA algorithm. The top down weights are designated as $w_{ji}, (1 \leq j \leq N, 1 \leq i \leq 2M)$, these can also be expressed in vector form as the set of weights that emanates from a given node $j$ in the $F_2$ layer as

$$w_j = (w_{j,1}, w_{j,2}, \ldots, w_{j,2M})$$

and are also called templates. Their purpose is to represent the group of input patterns that chose node $j$ in the $F_2$ layer of FA as their representative node.

FA can operate in two distinct phases: the training phase and the performance phase, but being an online algorithm it can also intermix them. The training phase of FA can be succinctly described as follows: Given a set of inputs, $I^1, \ldots, I^r, \ldots, I^P$, we want to train FA to map every input pattern of the training set to a reasonable template (cluster). To achieve the aforementioned goal we present the training set to FA architecture repeatedly. That is, we present $I^1$ to $F_1$, $I^2$ to $F_1$, and finally $I^P$ to $F_1$.

The operation of FA is affected by two network parameters, the choice parameter $\beta$, and the vigilance parameter $\rho$. The choice parameter takes values in the interval $(0, \infty)$, while the vigilance parameter assumes values in the interval $[0,1]$. Both of these parameters affect the number of nodes created in the category representation layer of FA. Higher values of $\beta$ and/or $\rho$ create more nodes in the category representation layer of FA, and consequently produce less compression of the input patterns. The number of nodes $N$ in the category representation layer of FA increases, while
training the network, and corresponds to the number of committed nodes in FA plus one uncommitted node.

Prior to initiating the training phase of FA the top-down weights (the \(w_{ji}\)'s) are chosen equal to 1. At this point the network will only have one node in the \(F_2\) layer that we will refer to as the uncommitted node. There are three major operations that take place during the presentation of a training input \(I^r\) to FA: calculation of the bottom up inputs, calculation of the vigilance ratio and change of the weights. One of the specific operands involved in all of these operations is the fuzzy min operand, designated by the symbol \(\wedge\). The fuzzy min operation of two vectors \(x = (x_1, \ldots, x_n)\), and \(y = (y_1, \ldots, y_n)\) is denoted as

\[
x \wedge y
\]

and is defined by the equation

\[
(x \wedge y)_i = \min(x_i, y_i) \cdot \forall i \in \{1, 2, \ldots, n\}
\]

that is: the fuzzy min of vectors \(x\) and \(y\) is a vector whose components are equal to the component-wise minimum of the components of \(x\) and \(y\). Another specific operand involved in these equations is designated by the symbol \(|\cdot|\). In particular, \(|x|\) is the size of a vector \(x = (x_1, x_2, \ldots, x_n)\) and is defined to be the sum of its components

\[
|x| = \sum_{i=0}^{n} x_i
\]
2.1.1 Fuzzy-ART Learning and Classification

The FA learning algorithm presents all the training pairs to the network one by one. For every input pattern $I^r, r \in \{1, 2, \ldots, P\}$ in the training set we execute the following steps.

1. Calculation of bottom up inputs for every node $j$ in $F_2$, as follows:

$$T_j(I^r, w_j, \beta) = \frac{|I^r \land w_j|}{\beta + |I^r|}$$  \hspace{1cm} (2.7)

2. After calculation of the bottom up inputs the node $j_{\text{max}}$ with the maximum bottom up input is chosen

$$j_{\text{max}} = \max \{T_j(I^r, w_j, \beta)\}$$  \hspace{1cm} (2.8)

3. The node $j_{\text{max}}$ with the maximum bottom up input is examined to determine whether it passes the vigilance criterion. A node passes the vigilance criterion if for the function (referred to as vigilance ratio):

$$\rho(I^r, w_j) = \frac{|I^r \land w_j|}{|I^r|}$$  \hspace{1cm} (2.9)

the following condition is met:

$$\rho(I^r, w_j) \geq \rho$$  \hspace{1cm} (2.10)

if the vigilance criterion is satisfied then the node is said to represent the input pattern and we proceed with the next step; otherwise node $j_{\text{max}}$ is disqualified.
from the competition and we find the next node in sequence in $F_2$ layer that maximizes the bottom up input. Eventually we will end up with a node $j_{\text{max}}$ that maximizes the bottom up input and passes the vigilance criterion.

4. Once we have found a node that passes all the previous tests then the weights in FA are modified as follows:

$$w_{j_{\text{max}}} \leftarrow w_{j_{\text{max}}} \wedge I$$  \hspace{1cm} (2.11)

If the winning template is the uncommitted node then a new uncommitted node is created and added to the network. This action guarantees that the network always has one uncommitted node.

FA training is considered complete if and only if after repeated presentations of all training input patterns to the network, where the previous operations are iteratively applied for every pattern, we find ourselves in a situation where a complete cycle through all the patterns produced no weight changes.

In the performance phase of FA only Operations 1, 2 and 3 are implemented for every input pattern presented to the network.

### 2.1.2 Fuzzy-ART pseudo-code

The pseudo-code for the FA training phase can be seen in figure 2.2.

where $T(I, w_j, \beta)$ is defined by equation (2.7) and $\rho(I, w_j)$ is defined by equation (2.9).
FA-Training-Phase($\{I^1, I^2, \ldots, I^P\}$, epochs, $\rho, \beta, \varepsilon$)

1. $w_0 \leftarrow \{1, 1, \ldots, 1\}_{2^M}$
2. templates $\leftarrow \{w_0\}$
3. iterations $\leftarrow 0$
4. repeat
5. modified $\leftarrow$ FALSE
6. for each $I^r$ in $\{I^1, I^2, \ldots, I^P\}$
7. do $T_{max} \leftarrow 0$
8. for each $w_j$ in templates
9. do if $\rho(I^r, w_j) \geq \rho$ and $T(I^r, w_j, \beta) > T_{max}$
10. then $T_{max} \leftarrow T(I^r, w_j, \beta)$
11. $w_{j_{max}} \leftarrow w_j$
12. if $w_{j_{max}} \neq w_0$
13. then
14. if $w_{j_{max}} \neq w_{j_{max}} \land I^r$
15. then $w_{j_{max}} \leftarrow w_{j_{max}} \land I^r$
16. modified $\leftarrow$ TRUE
17. else
18. templates $\leftarrow$ templates $\cup \{I^r\}$
19. iterations $\leftarrow$ iterations $+ 1$
20. until modified $=$ FALSE or iterations $=$ epochs
21. return templates

Figure 2.2: Fuzzy-ART off-line training phase pseudocode
FA-ONLINE-TRAINING-PHASE(\{I^1, I^2, \ldots, I^P\}, \rho, \beta, \varepsilon)

1. \( w_0 \leftarrow \{1, 1, \ldots, 1\} \)
2. \( \text{templates} \leftarrow \{w_0\} \)
3. for each \( I^r \) in \( \{I^1, I^2, \ldots, I^P\} \)
4. do \( T_{\text{max}} \leftarrow 0 \)
5. for each \( w_j \) in templates
6. do if \( \rho(I^r, w_j) \geq \rho \) and \( T(I^r, w_j, \beta) > T_{\text{max}} \)
7. then \( T_{\text{max}} \leftarrow T(I^r, w_j, \beta) \)
8. \( w_{\text{max}} \leftarrow w_j \)
9. if \( w_{\text{max}} \neq w_0 \)
10. then \( w_{\text{max}} \leftarrow w_{\text{max}} \land I^r \)
11. else \( \text{templates} \leftarrow \text{templates} \cup \{I^r\} \)
12. return \( \text{templates} \)

Figure 2.3: Fuzzy–ART online training phase pseudocode

When dealing with an the online training phase of FA we simplify the process by deleting the outer repeat loop to get the algorithm of figure 2.3.

Finally, the FA performance phase which presented in figure 2.4.

## 2.2 Fuzzy-ARTMAP

Fuzzy-ARTMAP (FAM) uses the Fuzzy–ART algorithm as a building block. FAM was developed by Carpenter et al in 1992 [CGM92]. It is a supervised learning neural network algorithm that utilizes two Fuzzy–ART modules named \( ART_a \) and \( ART_b \) connected by a map field called \( F_{ab} \). For classification purposes the interesting
FA-PERFORMANCE-PHASE($I, \text{templates, } \rho, \beta, \varepsilon$)
1 $T_{\text{max}} \leftarrow 0$
2 for each $w_j$ in templates
3 do if $\rho(I, w_j) \geq \rho$ and $T(I, w_j, \beta) > T_{\text{max}}$
4 then $T_{\text{max}} \leftarrow T(I, w_j, \beta)$
5 $w_{j_{\text{max}}} \leftarrow w_j$
6 return $w_{j_{\text{max}}}$

Figure 2.4: Fuzzy–ART performance phase pseudocode

computations happen in the $ART_a$ module (where the inputs to the network are applied). Most of the notation from Fuzzy–ART will be used here as well but we will use parameters and variables with a superscript $a$ or $b$ or $ab$ to remind us the module that they pertain to.

2.2.1 The Fuzzy-ARTMAP Architecture

In this dissertation we will concentrate on the use of FAM for classification problems. This focus simplifies the FAM architecture. More specifically, the FAM architecture will be composed of four layers or fields of nodes (see Figure 2.5). The layers that are worth describing are the input layer ($F^a_1$), the category representation layer now denoted as ($F^a_2$), and the output layer ($F^b_2$). The input layer of FAM is the layer where an input vector $I$ is applied, and as in FA this input is complement-coded and has a dimensionality $2M_a$. The category representation layer is the layer where compressed representations of the input patterns are formed. Finally, the output layer is the layer that produces the outputs of the network. An output of the network represents
the output to which the input applied at the input layer of FAM is supposed to be mapped to.

In FAM we have two sets of weights worth mentioning. The first set of weights are weights from $F^a_2$ to $F^a_1$, designated as $w^a_{ji}$, $(1 \leq j \leq N_a, 1 \leq i \leq 2M_a)$, and referred to as top-down weights. The vector of weights

$$w^a_j = (w^a_{j1}, w^a_{j2}, \ldots, w^a_{j,2M_a})$$

is called a template. Their purpose is to represent the group of input patterns that chose node $j$ in the category representation layer of FAM as their representative node. The second set of weights, worth mentioning, are the weights that emanate from every node $j$ in the category representation layer to every node $k$ in the output layer. These weights are designated as $W^{ab}_{jk}$, and we refer to them as inter–ART weights. The vector of inter–ART weights emanating from every node $j$ in FAM

$$W^{ab}_j = [W^{ab}_{j,1}, W^{ab}_{j,2}, \ldots, W^{ab}_{j,N_b}]$$

correspond to the output pattern that the node $j$ is mapped to.

FAM can also operate in two distinct phases: the training phase and the performance phase, and it is valid to intermix them since FAM can function in online fashion. The training phase of FAM can be succinctly described as follows: Given a set of input/output pairs:

$$\{(I^1, O^1), \ldots, (I^r, O^r), \ldots, (I^P, O^P)\}$$
Figure 2.5: Block Diagram of the Fuzzy-ARTMAP Architecture.
we want to train FAM to map every input pattern of the training set to its corresponding output pattern. To achieve the aforementioned goal we present the training set to FAM architecture repeatedly. That is, we present $I^1$ to $F_{a1}^a$, $O^1$ to $F_{b2}^b$, $I^2$ to $F_{a1}^a$, $O^2$ to $F_{b2}^b$, and finally $I^P$ to $F_{a1}^a$, and $O^P$ to $F_{b2}^b$. We present the training set to FAM as many times as it is necessary for FAM to correctly classify all these input patterns. The task is considered accomplished (i.e., the learning is complete) when the weights do not change during a training set presentation. The aforementioned training scenario is called off-line learning. The performance phase of FAM works as follows: Given a set of input patterns, such as $\tilde{I}^1, \tilde{I}^2, \ldots, \tilde{I}^P$, we want to find the FAM output produced when each one of the aforementioned test patterns is presented at its $F_{a1}^a$ layer. In order to achieve the aforementioned goal we present the test set to the trained FAM architecture and we observe the network’s output.

The operation of FAM is affected by two network parameters, the choice parameter $\beta_a$, and the baseline vigilance parameter $\bar{\rho}_a$. The choice parameter takes values in the interval $(0, \infty)$, while the baseline vigilance parameter assumes values in the interval $[0,1]$. These parameters affect the number of nodes created in the category representation layer of FAM similarly as the parameters $\rho$ and $\beta$ affect FA. Higher values of $\beta_a$ and $\bar{\rho}_a$ create more nodes in the category representation layer of FAM, and consequently produce less compression of the input patterns. There are two other network parameter values in FAM that are worth mentioning. The vigilance parameter $\rho_a$, and the number of nodes $N_a$ in the category representation layer of FAM. Fortunately both of these parameters are set automatically by the algorithm to their relevant values. The vigilance parameter $\rho_a$ takes value in the interval $[\bar{\rho}_a, 1]$ and its initial value is set to be equal to $\bar{\rho}_a$. The number of nodes $N_a$ in the category
representation layer of FAM increases while training the network and corresponds to the number of committed nodes in FAM plus one uncommitted node.

Prior to initiating the training phase of FAM the top-down weights (the $w_{ji}$’s) are chosen equal to 1, and the inter–ART weights (the $W_{jk}$’s) are chosen equal to 0.

### 2.2.2 The Fuzzy-ARTMAP Learning Algorithm

The FAM learning algorithm presents all the training pairs to the network one by one. For every input pattern $\Gamma \in TrainingSet$ we do

1. Calculation of bottom up inputs for every node $j$ in $F_2^n$, as follows:

   $$T(\Gamma, w_j^a, \beta_a) = \frac{|\Gamma \wedge w_j^a|}{\beta_a + |w_j^a|}$$

   (2.15)

2. after calculation of the bottom up inputs the node $j_{max}$ with the maximum bottom up input is chosen

   $$j_{max} = \max \arg \{T(\Gamma, w_j^a, \beta_a)\}$$

   (2.16)

3. The node $j_{max}$ with the maximum bottom up input is examined to determine whether it passes the vigilance criterion. A node passes the vigilance criterion if for the function (referred to as the vigilance ratio):

   $$\rho(\Gamma, w_j^a) = \frac{|\Gamma \wedge w_j^a|}{|\Gamma|}$$

   (2.17)
the following condition is met:

\[ \rho(\Gamma, \mathbf{w}_j^a) \geq \rho_a \]  

(2.18)

if the vigilance criterion is satisfied we proceed with the next step; otherwise node \( j_{\text{max}} \) is disqualified from the competition and we find the next node in sequence in the \( F_2^a \) layer that maximizes the bottom up input. Eventually we will end up with a node \( j_{\text{max}} \) that maximizes the bottom up input and passes the vigilance criterion.

4. This operation is implemented only after we have found a node \( j_{\text{max}} \) that maximizes the bottom-up input of the remaining nodes in competition and passes the vigilance criterion. Here we determine whether the node \( j_{\text{max}} \) passes the prediction test. The prediction test checks if the inter–ART weight vector emanating from node \( j_{\text{max}} \)

\[ \mathbf{W}_{j_{\text{max}}}^{ab} = [W_{j_{\text{max}}1}^{ab}, W_{j_{\text{max}}2}^{ab}, \ldots, W_{j_{\text{max}},N_b}^{ab}] \]  

(2.19)

matches exactly the desired output vector \( \mathbf{O}^r \) (if it does then it is referred to as passing the prediction test). If the node does not pass the prediction test, the vigilance parameter \( \rho_a \) is increased to the level of

\[ \rho_a \leftarrow \frac{|\Gamma \wedge \mathbf{w}_j^a|}{|\Gamma|} + \varepsilon \]  

(2.20)

where \( \varepsilon \) is a very small positive constant, node \( j_{\text{max}} \) is disqualified, and the next in sequence node that maximizes the bottom-up input and passes the vigilance is chosen.
5. Once we have found a node that passes all the previous tests then the weights in FAM are modified as follows:

\[
\mathbf{w}^a_{j_{\text{max}}} \leftarrow \mathbf{w}^a_{j_{\text{max}}} \wedge I^r
\]  
(2.21)

\[
\mathbf{W}^{ab}_{j_{\text{max}}} \leftarrow \mathbf{O}^r
\]  
(2.22)

FAM training is considered complete if and only if after repeated presentations of all training input/output pairs to the network, where Operations 1-5 are iteratively applied for every input/output pair, we find ourselves in a situation where a complete cycle through all the input/output pairs produced no weight changes. In some databases noise in the data may create over-fitting when we repeatedly present the input/output pairs, so a single pass over the training set may be preferable. This situation also arises when we do online training of the network with an unlimited data source.

In the performance phase of FAM only Operations 1, 2 and 3 are implemented for every input pattern presented to the network. By registering the network output to every test input presented to FAM, and by comparing it to the desired output we can calculate the network’s performance (i.e., network’s misclassification error).

### 2.2.3 Fuzzy-ARTMAP pseudo-code

We will concentrate our analysis on optimizing FAM as it performs a single pass over the training data because this operation is performed in both the online and the offline versions of FAM. We will also, without loss of generality, model every \( \mathbf{O}^r \) of
the \((\mathbf{I}', \mathbf{O}')\) training pair as a unitary vector \(e_i\) of the canonical basis of the form \((0, 0, \ldots, 1, \ldots)\). The pseudo-code in figure 2.6 makes use of these assumptions and refers to \(\text{class}(\mathbf{I}')\) and \(\text{class}(\mathbf{w}_j^a)\) as a function that returns the class (output \(\mathbf{O}'\)) associated with an input pattern \(\mathbf{I}'\) and the class associated with the template \(\mathbf{w}_j^a \in F_2^a\) layer, respectively.

Note that \(T(\mathbf{I}', \mathbf{w}_j^a, \beta_a)\) is defined by equation 2.15 and \(\rho(\mathbf{I}', \mathbf{w}_j^a)\) is defined by equation 2.17. With the online or single pass version of the algorithm we simplify the process deleting the outer \texttt{repeat} loop to get the algorithm in figure 2.7. Finally, the performance phase of FAM is almost identical to the performance phase of FA, but instead of returning the template we return the class label associated with the winning template. The pseudocode for the performance phase of FAM is shown in figure 2.8.

### 2.3 Fuzzy-ARTMAP properties

**Theorem 2.3.1.** The template weights \(\mathbf{w}_j^a\) in the category representation layer always decrease during training in every component and in norm \(|\mathbf{w}_j^a|\).

**Proof.** This is a direct consequence of the weight update operation of equation 2.11 which is the only part in the algorithm were the value of the weights are changed. This operation uses the fuzzy–\(\min\) \(\wedge\) operand which does a component–wise decreasing of the values of the weights. \(\square\)
FAM-TRAINING-PHASE($\{I^1, I^2, \ldots, I^P\}$, epochs, $\bar{p}_a, \beta_a, \varepsilon$)

1. $w_0 \leftarrow (1, 1, \ldots, 1)$
2. templates $\leftarrow \{w_0\}$
3. iterations $\leftarrow 0$
4. repeat
5. \hspace{1em} modified $\leftarrow$ FALSE
6. \hspace{1em} for each $I^r$ in $\{I^1, I^2, \ldots, I^P\}$
7. \hspace{2em} do $\rho_a \leftarrow \bar{p}_a$
8. \hspace{2em} repeat
9. \hspace{3em} $T_{\text{max}} \leftarrow 0$
10. \hspace{3em} for each $w^a_j$ in templates
11. \hspace{4em} do if $\rho(I^r, w^a_j) \geq \rho_a$ and $T(I^r, w^a_j, \beta_a) > T_{\text{max}}$
12. \hspace{5em} then $T_{\text{max}} \leftarrow T(I^r, w^a_j, \beta_a)$
13. \hspace{5em} $w_{j_{\text{max}}} \leftarrow w_j$
14. \hspace{4em} if $w_{j_{\text{max}}} = w_0$ or class($I^r$) = class($w_{j_{\text{max}}}$)
15. \hspace{5em} then status $\leftarrow$ Allocated
16. \hspace{5em} else status $\leftarrow$ TryAgain
17. \hspace{4em} $\rho_a \leftarrow \rho(I^r, w^a_{j_{\text{max}}}) + \varepsilon$
18. \hspace{3em} until status $\neq$ TryAgain
19. \hspace{2em} if $w_{j_{\text{max}}} \neq w_0$
20. \hspace{3em} then
21. \hspace{4em} if $w^a_{j_{\text{max}}} \neq w^a_{j_{\text{max}}} \land I^r$
22. \hspace{5em} then $w^a_{j_{\text{max}}} \leftarrow w^a_{j_{\text{max}}} \land I^r$
23. \hspace{5em} modified $\leftarrow$ TRUE
24. \hspace{3em} else
25. \hspace{4em} templates $\leftarrow$ templates $\cup \{I^r\}$
26. \hspace{4em} iterations $= \text{iterations} + 1$
27. \hspace{3em} until modified $= \text{FALSE}$ or iterations $= \text{epochs}$
28. return templates

Figure 2.6: Fuzzy–ARTMAP off–line training phase pseudocode
FAM-ONLINE-TRAINING-PHASE(\{I^1, I^2, \ldots, I^P\}, \rho_a, \beta_a, \varepsilon)

1. \( w_0 \leftarrow (1,1,\ldots,1) \)
2. \( \text{templates} \leftarrow \{w_0\} \)
3. for each \( I^r \) in \( \{I^1, I^2, \ldots, I^P\} \)
4. do \( \rho_a \leftarrow \rho_a \)
5. repeat
6. \( T_{\text{max}} \leftarrow 0 \)
7. \( \text{status} \leftarrow \text{NoneFound} \)
8. for each \( w_{j}^a \) in \( \text{templates} \)
9. do if \( \rho(I^r, w_{j}^a) \geq \rho_a \) and \( T(I^r, w_{j}^a, \beta_a) > T_{\text{max}} \)
10. then \( T_{\text{max}} \leftarrow T(I^r, w_{j}^a, \beta_a) \)
11. \( w_{j_{\text{max}}} \leftarrow w_{j} \)
12. if \( w_{j_{\text{max}}} = w_0 \) or \( \text{class}(I^r) = \text{class}(w_{j_{\text{max}}}^a) \)
13. then \( \text{status} \leftarrow \text{Allocated} \)
14. else \( \text{status} \leftarrow \text{TryAgain} \)
15. \( \rho_a \leftarrow \rho(I^r, w_{j_{\text{max}}}^a) + \varepsilon \)
16. until \( \text{status} \neq \text{TryAgain} \)
17. if \( w_{j_{\text{max}}} \neq w_0 \)
18. then
19. \( w_{j_{\text{max}}}^a \leftarrow w_{j_{\text{max}}}^a \land I^r \)
20. else
21. \( \text{templates} \leftarrow \text{templates} \cup \{I^r\} \)
22. return \( \text{templates} \)

Figure 2.7: Fuzzy–ARTMAP online training phase pseudocode
FAM-Performance-Phase($I^r$, templates, $\rho_a, \beta_a, \varepsilon$)

1. $T_{max} \leftarrow 0$
2. for each $w^a_j$ in templates do
   3. if $\rho(I^r, w^a_j) \geq \rho_a$ and $T(I^r, w^a_j, \beta) > T_{max}$ then
   4. $T_{max} \leftarrow T(I^r, w^a_j, \beta_a)$
   5. $w_{j_{max}} \leftarrow w^a_j$
6. return $\text{class}(w_{j_{max}})$

Figure 2.8: Fuzzy–ARTMAP performance phase pseudocode

**Theorem 2.3.2.** In fast learning, every template $w^a_j$ is equal to:

\[ w^a_j = I^{k_1} \land I^{k_2} \land \cdots \land I^{k_s} \]  

(2.23)

where

\[ S = \{I^{k_1}, I^{k_2}, \ldots, I^{k_s}\} \]  

(2.24)

is the set of input patterns that chose to be represented by template $w^a_j$.

**Proof.** Let $w^a_j$ be an arbitrary template in the template set and $S$ be the set of templates that are represented by $w^a_j$. We will prove this by induction on $|S|$, the size of the set of patterns that are represented by $w^a_j$.

Let us start with the case $|S| = 1$ since when $|S| = 0$ it means that no templates are represented by $w^a_j$ which is the same as stating that $w^a_j$ is the uncommitted node.

**case n = 1:** If $|S| = 1$ then we have only 1 pattern committed to this template and by the weight update operation in equation (2.21) we have that:

\[ w^a_j \leftarrow w_0 \land I^{s_1} = (1, 1, \ldots, 1) \land I^{s_1} = I^{s_1} \]  

(2.25)
**case n + 1:** By induction hypothesis every template $w_j^a$ with associated set $S$ of input patterns that classify to the template $w_j^a$ with $|S| = n$ will be equal to:

$$w_j^a = I^{k_1} \land I^{k_2} \land \cdots \land I^{k_n}$$

where $I^{k_i}, i = 1, \ldots n$ is any template that belongs to $S$.

Now let's assume that $|S| = n + 1$, this means that on the last weight update operation

$$w_j^a \leftarrow w_j^a \land I^r$$

for some $I^r$, and that before the weight update function the template $w_j^a$ represented $n$ patterns, which means that $|S|$ was equal to $n$. So at the point of this operation, by induction hypothesis, it must be true that

$$w_j^a \leftarrow w_j^a \land I^r \leftarrow (I^{k_1} \land I^{k_2} \land \cdots \land I^{k_n}) \land I^r \Rightarrow$$

$$w_j^a = \underbrace{I^{k_1} \land I^{k_2} \land \cdots \land I^{k_n} \land I^r}_{n+1 \text{ terms}}$$

\[ \square \]

**Theorem 2.3.3.** *In fast learning, the number of different templates that a given training set can create is finite.*

**Proof.** By theorem 2.3.2 every $w_j^a$ is actually equal to

$$w_j^a = I^{k_1} \land I^{k_2} \land \cdots \land I^{k_j} \quad (2.26)$$
where
\[ S = \{ I^{k_1}, I^{k_2}, \ldots, I^{k_s} \} \]  
(2.27)
is the set of input patterns that classify to template \( w^a_j \).

Now since the number of possible sets \( S \) is finite and equal to \( 2^P \) (the cardinality of the power set for the set of training patterns), then the number of possible templates is also finite. \( \square \)

**Theorem 2.3.4.** The template weights \( w^a_j \) in the category representation layer are guaranteed to reach a steady state (training will stop).

**Proof.** By theorem 2.3.1 we know that the entries of the templates of the category representation layer are always decreasing. Also we know that their values cannot be negative, so their lower bound is 0. Any bounded monotone succession of values must have a limit. Now since the number of possible templates and therefore of possible template changes is finite (theorem 2.3.3) then the template must eventually reach it’s limit in a finite number of steps. \( \square \)

**Theorem 2.3.5.** All the patterns \( I^r \) in the training set
\[ \{(I^1, O^1), \ldots, (I^r, O^r), \ldots, (I^P, O^P)\} \]
that have been represented by template \( w^a_j \) have the same activation \( T(I^r, w^a_j, \beta_a) \).

**Proof.** All the patterns \( I^r \) that are represented by template \( w^a_j \) had to be learned by template \( w^a_j \) using equation 2.21. This implies that the template \( w_j \) is component–
wise smaller than all the patterns $I^r$ that chose it. This means that the expression:

$$T(I^r, w_j^a, \beta_a) = \frac{|I^r \land w_j^a|}{\beta_a + |w_j^a|} = \frac{|w_j^a|}{\beta_a + |w_j|}$$

(2.28)

is valid. Since formula (2.28) is independent of the input pattern $I^r$ it is obviously the same for all of them.

**Theorem 2.3.6.** When a template $w_j^a$ is updated using the weight update formula (2.21) the activation $T(I^k, w_j^a, \beta_a)$ of all the patterns in $S$: the set of templates that classify to $w_j^a$

$$S = \{I^{k_1}, I^{k_2}, \ldots, I^{k_s}\}$$

is decreased.

**Proof.** We know from theorem 2.3.5 that the activation of all the patterns that are represented by template $w_j$ is the same. By theorem 2.3.1 we also know that the value of the template size will always decrease in the weight update operation. Since the activation is given by

$$\frac{|w_j^a|}{\beta_a + |w_j^a|} = \frac{x}{\beta_a + x}$$

(2.29)

if we substitute $|w_j^a|$ by $x$, we can then derive equation (2.29) with respect to $x$ and get the expression

$$\frac{\partial \left( \frac{x}{\beta_a + x} \right)}{\partial x} = \frac{(\beta_a + x) - x}{(\beta_a + x)^2} =$$

$$\frac{\beta_a}{(\beta_a + x)^2}$$

which is always positive.
Therefore equation 2.29 is a monotone increasing function and therefore when we update the template and decrease $x$ we also decrease the activation of all the patterns in $\mathcal{S}$ that are represented by template $w^a_j$.

2.4 Geometric interpretation of Fuzzy-ARTMAP

**Definition:** for every template $w^a_j$ defined in equation 2.12 in the category representation layer, let $R(w^a_j) \subset [0, 1]^M_a$ represent the hyper-rectangle denoted by

$$R(w^a_j) = \{ \mathbf{x} : \forall i, w^a_{ji} \leq x_i \leq 1 - w^a_{j,M_a+i} \}$$  \hspace{1cm} (2.30)

**Theorem 2.4.1.** Every input pattern $\mathbf{I}'$ in the training set that is represented by the template $w^a_j$ lies in the region $R(w^a_j)$

$$\mathbf{I}' \in R(w^a_j)$$  \hspace{1cm} (2.31)

**Proof.** Let $\mathbf{I}'$ be an input pattern in the training set that is represented $w^a_j$. By theorem 2.3.5 we know that the entries in $\mathbf{I}'$ are component-wise smaller than the entries in $w^a_j$. By equation 2.1 we know that $\mathbf{I}' = (\mathbf{a}, \mathbf{a}^c)$ and $\mathbf{I}'$ is a complement coded representation of $\mathbf{a}$. Therefore it is sufficient to prove that

$$\mathbf{a} \in R(w^a_j)$$
Since \( \Gamma \) is represented by \( w_j^a \) we know that

\[
    w_{ji}^a \leq a_i
\]  

(2.32)

and that

\[
    w_{j,Ma+i}^a \leq a_i^c \Rightarrow \\
    a_i \leq 1 - w_{j,Ma+i}^a
\]  

(2.33)

By equations (2.32) and (2.33) we get

\[
    \forall i : w_{ji}^a \leq a_i \leq 1 - w_{j,Ma+i}^a \Rightarrow a \in R(w_j^a)
\]  

(2.34)

\[\square\]

**Theorem 2.4.2.** For any hyper-rectangle region \( R \subset [0, 1]^{Ma} \) that includes all the input patterns \( \Gamma^s \) from the training set that are represented by template \( w_j^a \), the following condition is true

\[ R(w_j^a) \subset R \]

*Proof.* Let \( x \) be the vector denoted by

\[
    x = (x_1, x_2, \ldots, x_{Ma})
\]
and $R$ be a hyper-rectangle that includes all the input patterns $\mathbf{I}^r$ that are represented by template $\mathbf{w}_j$, where $R$ is defined by

$$R = \{ \mathbf{x} : \forall i \cdot l_i \leq x_i \leq u_i \}$$

and $l_i$ and $u_i$ are the lower and upper bound respectively of the hyper-rectangle $R$ in dimension $i$.

Let us take an arbitrary $\mathbf{y} \in R(\mathbf{w}_j^a)$. Also let $a^r_i$ be the $i^{th}$ entry in vector $\mathbf{a}$ from $\mathbf{I}^r = (\mathbf{a}, \mathbf{a}^c)$, the $r^{th}$ input patterns of the set of patterns that are represented by $\mathbf{w}_j$.

$$\mathbf{y} \in R(\mathbf{w}_j^a) \Rightarrow$$

$$w_{ji}^a \leq y_i \leq 1 - w_{j,M_i}^a \Rightarrow$$
\[
\min_r \{a_i^r\} \leq y_i \leq 1 - \min_r \{1 - a_i^r\} \\
\min_r \{a_i^r\} \leq y_i \leq 1 - (1 - \max_r \{a_i^r\}) \\
\min_r \{a_i^r\} \leq y_i \leq \max_r \{a_i^r\} \tag{2.35}
\]

we also know that since \( R \) contains all the input patterns \( I^r \) that are represented by \( w_j \) then for every \( r \) and arbitrary \( k \) it is true that

\[
\forall r \cdot l_k \leq a_k^r \Rightarrow l_k \leq \min_r \{a_k^r\}
\]

\[
\forall r \cdot a_k^r \leq u_k \Rightarrow \max_r \{a_k^r\} \leq u_j
\]

in particular it is true that

\[
l_i \leq \min_r \{a_i^r\} \tag{2.36}
\]

\[
\max_r \{a_i^r\} \leq u_i \tag{2.37}
\]

from equations 2.35, 2.36 and 2.37 we can conclude that

\[
l_i \leq \min_r \{a_i^r\} \leq y_i \leq \max_r \{a_i^r\} \leq u_i \Rightarrow
\]

\[
l_i \leq y_i \leq u_i
\]

since this is true for arbitrary \( i \) then it is true that \( y \in R \). \( \square \)
Definition: We will call the size of a region $R(w^a_j)$ associated with a template $w^a_j$ the value defined by

$$size(R(w^a_j)) = \sum_{i=1}^{M_a} 1 - w^a_{ji} - w^a_{j,M_a+i}$$  \hspace{1cm} (2.38)

Theorem 2.4.3. $R(w^a_j)$ complies with the following inequality

$$size(R(w^a_j)) \leq M_a(1 - \bar{\rho}_a)$$ \hspace{1cm} (2.39)

Proof.

$$size(R(w^a_j)) = \sum_{i=1}^{M_a} 1 - w^a_{ji} - w^a_{j,M_a+i} =$$

$$\sum_{i=1}^{M_a} 1 + \sum_{i=1}^{M_a} (-w^a_{ji} - w^a_{j,M_a+i}) =$$

$$\sum_{i=1}^{M_a} 1 - \sum_{i=1}^{M_a} (w^a_{ji} + w^a_{j,M_a+i}) =$$

$$M_a - \sum_{i=1}^{2M_a} w^a_{ji} = M_a - |w^a_j| \Rightarrow$$

$$size(R(w^a_j)) = M_a - |w^a_j|$$ \hspace{1cm} (2.40)

now since every template must meet the vigilance criterion, then it is true that for every input pattern $I^r$ that is represented by template $w^a_j$

$$\frac{|I^r \land w^a_j|}{|I|} \geq \bar{\rho}_a$$
but since $I^r$ is represented by $w^a_j$ then by theorem 2.3.2 it is also true that

$$\frac{|I^r \land w^a_j|}{|I|} \geq \bar{\rho}_a \Rightarrow \frac{|w^a_j|}{|I|} \geq \bar{\rho}_a \Rightarrow \frac{|w^a_j|}{M_a} \geq \bar{\rho}_a \Rightarrow$$

$$|w^a_j| \geq M_a\bar{\rho}_a$$

(2.41)

and combining the results of equation 2.40 and equation 2.41 we get that

$$size(R(w^a_j)) = M_a - |w^a_j| \leq M_a - M_a\bar{\rho}_a = M_a(1 - \bar{\rho}_a) \Rightarrow$$

$$size(R(w^a_j)) \leq M_a(1 - \bar{\rho}_a)$$

(2.42)

\[\square\]

### 2.5 Fuzzy-ARTMAP time complexity analysis

To analyze the time complexity of the FAM algorithm we will first concentrate on the online version of the algorithm first. Modifications applied on the online version of the algorithm have the advantage of being useful for both online and offline learning.

After that we will model the offline version of the algorithm. Our approach requires of making a few assumptions about FAM. Based on these assumptions modifications of FAM are proposed.
2.5.1 Online Fuzzy-ARTMAP complexity

We can see from the pseudocode that the FAM algorithm tests every input pattern $I$ in the training set against each template $w_j$ at least once. Let us call $\Gamma$ the average number of times that the inner repeat loop (lines 5 to 16 of the online training phase algorithm of figure 2.7) is executed for each input pattern, and christen it the matchtracking factor. Then the number of times that a given input pattern $I$ passes through the code will be:

$$\text{Time}(I) = O(\Gamma \times |\text{templates}|) \quad (2.43)$$

Also, under the unrealistic condition that the number of templates does not change during training it is easy to see that the time complexity of the algorithm is:

$$\text{Time}(FAM) = O(\Gamma \times P \times |\text{templates}|) \quad (2.44)$$

Usually for a fixed type of database the FAM algorithm achieves a certain compression ratio. This means that the number of templates created is actually a fraction of the number of patterns $P$ in the training set. Let us call this compression ratio $\kappa$ so that:

$$|\text{templates}| = \kappa P \quad (2.45)$$

and

$$O(FAM) = O(\Gamma P \kappa P) = O(\kappa \Gamma P^2) \quad (2.46)$$
This formula will be useful when we proceed with the data partitioning approaches.

2.5.2 Off line Fuzzy-ARTMAP complexity

When dealing with the offline version of FAM, the considerations taken into account for the online FAM also apply. Here though, we repeat the algorithmic process for the full training set for as many epochs as is necessary to achieve 100% (exact) classification on the training set. There is no way to derive a formula for the number of epochs that the algorithm will execute, and also there is no way of knowing what is going to be the average number of templates for each iteration, obviously this number increases since the number of templates never goes down (unless we do pruning, which is not contemplated in the FAM algorithm). So without any previous assumptions the complexity of the off line FAM algorithm would be:

\[
O\left(\sum_{i=1}^{epochs} \kappa_i \Gamma_i P^2\right)
\]

where \(\Gamma_i\) is the matchtracking factor for the \(i^{th}\) epoch, and \(\kappa_i\) is the compression ratio of the \(i^{th}\) epoch.
2.5.2.1 Fuzzy-ARTMAP off line complexity simplification

We can simplify $A$ by making some assumptions as to the values of $epochs$, $\Gamma_i$ and $\kappa_i$. If, as it may be reasonable, we assume that the number of epochs is dependent on the number of training patterns, then $A$ becomes

$$
A = \sum_{i=1}^{epochs(P)} \kappa_i \Gamma_i P^2 = P^2 \sum_{i=1}^{epochs(P)} \Gamma_i \kappa_i
$$

(2.48)

The next assumption we will make is that $\Gamma_i$ does not vary from iteration to iteration but is a linear function of the compression ratio $\kappa_i$. This assumption leads to a constant $\Gamma$ for all $i$ and the formula:

$$
A = \sum_{i=1}^{epochs(P)} \kappa_i \Gamma P^2 = \Gamma P^2 \sum_{i=1}^{epochs(P)} \kappa_i
$$

(2.49)

Finally we can assume that $\kappa_i = \kappa^i$, which basically means that the amount of templates created decreases geometrically from iteration to iteration. Using these assumptions we can state the formula as:

$$
A = \sum_{i=1}^{epochs(P)} \kappa^i \Gamma P^2 = \Gamma P^2 \sum_{i=1}^{epochs(P)} \kappa^i = \Gamma P^2 \kappa \left( \frac{1 - \kappa^{epochs(P)}}{1 - \kappa} \right)
$$

(2.50)

We know by the assumption on equation [2.45] that the number of templates created on each iteration is:

$$
\kappa_i P = \kappa^i P
$$

(2.51)
This means that the algorithm should stop when no more templates are created and this condition will be met when

\[ \kappa^i P < 1 \Rightarrow \kappa^i < \frac{1}{P} = P^{-1} \Rightarrow \]

\[ \log(\kappa^i) < \log(P^{-1}) \Rightarrow i \log(\kappa) < -\log(P) \Rightarrow \]

\[ i > -\frac{\log(P)}{\log(\kappa)} \Rightarrow i = \left\lfloor -\frac{\log(P)}{\log(\kappa)} \right\rfloor \]  
(2.52)

which gives a final analytical value for the time complexity formula as

\[ \kappa \Gamma P^2 \left( \frac{1 - \kappa \left\lfloor -\frac{\log(P)}{\log(\kappa)} \right\rfloor}{1 - \kappa} \right) \]  
(2.53)

and a number of templates given by the formula

\[ \kappa P \left( \frac{1 - \kappa \left\lfloor -\frac{\log(P)}{\log(\kappa)} \right\rfloor}{1 - \kappa} \right) \]  
(2.54)

### 2.5.2.2 Validity of assumptions

The validity of the previous assumptions can only be verified by doing experimental analysis. Also the value of the parameters \( \Gamma \) and \( \kappa \) are probably not constant but are problem dependent. Not so obvious is the fact that they are probably also algorithm dependent. By that we mean that if we use FA then the amount of templates generated by the algorithm will probably be less than if we used FAM. FAM uses matchtracking that can create new templates in situations that FA does not. What this translates into is that the value of \( \kappa \) will probably be different for FA than for
FAM, so comparison of their time is not a straightforward issue of just taking the part of the formula of interest.

For asymptotic behavior though, it really doesn’t matter what the value of $\kappa$ is, since what we are trying to evaluate is the scalability of the algorithm when the amount of patterns $P$ to classify changes.

A very important parameter that is left out of analysis from this formula is the parameter $\tilde{\rho}_a$. Obviously $\Gamma$ and $\kappa$ are dependent on $\tilde{\rho}_a$, and we know that the larger the value of $\tilde{\rho}_a$, the larger the amount of templates that the algorithm generates in the $F_2$ layer. But a precise characterization of this relationship is being left out of this analysis for lack of sufficient experimentation to predict a behavior associated with this parameter.
CHAPTER 3
PARALLEL PROCESSING

Neural networks, by the very nature of combining neurons into a neural processing unit, are an intrinsically parallel paradigm. We can, if we choose to, naively map each neuron to a single processor and immediately get a parallel processing machine. Some specialized hardware chips provide exactly this functionality, but even in the context of such obvious parallelization, the mapping between network architecture and parallel machine is usually not so trivial. In this chapter we outline the different types of parallelization available and concentrate on the parallel platform of choice: the BEOWULF cluster of workstations.

3.1 Types of Parallel Machines

One way to create a taxonomy of parallel machines is to categorize the way in which memory is used and organized in the parallel system. Shared memory multiprocessors extend the one processor, one single memory address space paradigm of conventional computers to a multi processor, one memory address space scenario. The connection of the processors to the memory is done via some interconnection network specially tailored for this task (see figure 3.1). This paradigm is attractive to programmers because data sharing is the simplest form of communication between processes, and
processes that already run in a uniprocessor will run on the multiprocessor with little modification.

Shared memory multiprocessors require special hardware to implement the common address space and address conflict resolution. Another approach is the use of a message passing interface. This approach also has an advantage of simplicity of implementation, since the message passing protocol can be implemented as much in hardware as in software. Also a message passing interface can be incorporated in a normal network of uniprocessors, which can scale easily since processing elements can be incrementally added to the system.

Since the shared memory can be easier to program and the message passing is usually easier to implement, some researchers have opted for a combination of the two paradigms in the distributed shared memory implementation. Under this scheme a virtual address space is created based on a message passing protocol amongst the distributed processes as shown in figure 3.3.

3.1.1 Flynn’s Classification

A more formal but less specific classification was proposed by Flynn in 1966 \cite{Fly66} in which the level of parallelism is characterized by specifying the relationship and cardinality of instructions against data. This classification is:

- **SISD**: The classic model of computation assigns a memory address space to a processor, this model in Flynn’s classification is the *single instruction stream–single data stream* or SISD machine for short. We can find in this category
microprocessors like the 8086. The more modern Pentium® line of processors would not formally fall into this category since it already has multiple threads of execution with an incorporated mathematical coprocessor, but by most practical standards it is considered a SISD machine.

- SIMD: For some applications, it is common to apply the same operations over a large amount of data, this is true for linear algebra math intensive applications that require a lot of vector and matrix manipulations. The single instruction stream–multiple data stream machine or SIMD machine, has been designed with this type of task in mind and they are common in numerical intensive applications like weather forecasting, computer graphics rendering, numerical simulation and others.

- MISD: This type of computation is provided for completeness of the Flynn classification, this is the multiple instruction stream–single data stream machine. MISD machines are usually not implemented unless we count in this category the pipeline architecture where a stream of data is processed using a sequence of instructions.

- MIMD: The most general model of distributed computation provided is the scenario where a multitude of processors operate independently using each a different set of instructions and operating on different data each. This scenario is the multiple instruction stream–multiple data stream machine or MIMD machine for short. In this category we can find more decoupled systems like networks of workstations and clusters.
Figure 3.1: Shared memory multiprocessor model.

$M_1$ to $M_k$ are memory modules and $P_1$ to $P_n$ are processors.
Figure 3.2: Message passing multiprocessor model.

\( M_1 \) to \( M_n \) are memory modules and \( P_1 \) to \( P_n \) are processors.
Figure 3.3: Distributed shared memory multiprocessor model.

M_1 to M_n are memory modules and P_1 to P_n are processors.
3.2 Message-Passing Architectures and Techniques

All the architecture diagrams presented above (figures 3.1, 3.2, and 3.3) leave unspecified the interconnection network required to do the actual communication between processors. This communication can be done via different network topologies that are the actual ones to provide some sort of physical point to point communication links.

3.2.1 Network Topologies

The simplest but most expensive type of network topology is the fully connected network. This scheme requires a physical link between each processor in the network. Communication can be very fast and uninterrupted but scalability and cost are compromised since on a network of \( n \) nodes the number of links is \( \frac{n(n-1)}{2} \) which is quadratic in the number of nodes and the inclusion of a new node in this network requires the addition of \( n \) new communication links.

A much simpler and cheaper structure is the line or ring network topology depicted in figure 3.5. This type of network reduces the cost to a minimum by eliminating the need for point to point links between machines. Any communication made through non-adjacent nodes will require the retransmission by the intermediary nodes. A Line network structure has only one access path between nodes and is therefore more vulnerable to network link failures. A line structure also has different latency times for nodes that are near the middle of the ring than nodes that are close to the ends of the line, this bias can be eliminated by converting the line into a ring structure.
Figure 3.4: Fully connected Network Architecture.

Figure 3.5: Line Network Architecture.
Another network architecture uses the policy of dedicating one processor to the routing, transmission and network overhead processing and making all the other processors in the network communicate directly to the dedicated node. This arrangement leads to the *star* architecture depicted in figure 3.6

The main disadvantage of star topologies is that they do not scale easily, eventually in the life cycle of the network it is probable that a point is reached where it is no longer feasible to connect all processors to a single dedicated node as the star architecture demands. This is specially true for larger networks, and under these circumstances a hierarchical network called a *tree* network is the preferred choice (see figure 3.7). A balanced tree network with $n$ nodes and a node fan out of $m$ has can another node in the network in $\log_m(n)$ hops in the network.
Another popular network architecture is the mesh. In this mesh processors are arranged in a 2-dimensional grid and communicate with their immediate neighbors. Communication times in the mesh are in the order of $\sqrt{n}$ where $n$ is the number of nodes in the mesh.

Finally we have the hypercube architecture. In this arrangement we number the nodes with a binary index and connect the node with those nodes that have a Hamming distance no greater than 1. This architecture has an access time between nodes of $\log(n)$.

Sometimes it is necessary to simulate a network topology on top of a different physical topology. When this situation happens it is called embedding. In our case, we will embed the pipeline (line) and tree architectures in the topology of the UCF Scerola parallel cluster.
Figure 3.8: Mesh Network Architecture.
Figure 3.9: Four dimensional hypercube network architecture.
3.3 Clusters of Workstations and the Scerola cluster

A cluster of workstations is a network of computers configured to function as a high performance parallel machine. Usually an interface is provided so that the machine is perceived by the user as a single dedicated machine by assigning one node of the network to function as the network’s front end. The type of cluster we will use in this dissertation is the Beowulf parallel platform. This platform is a software layer added to the Linux operating system that combines the network into a single high performance machine. The Beowulf cluster platform was developed at the NASA research laboratories and the Scerola parallel cluster is an implementation of this parallel standard. Communication within the cluster can be achieved using different standard libraries, the most popular being the message passing interface MPI and the parallel virtual machine PVM interface.

The Scerola parallel cluster implements the MPI standard and a queuing system to organize the tasks in the cluster. The Scerola cluster runs on top of a Fast Ethernet 100Mb switching network. This allows for the efficient embedding of different network architectures and good point to point communication. In our particular case we will be interested in embedding a ring and a tree topology on the cluster.

3.4 Speedup, Efficiency and Cost

One of the most commonly used measures for parallel computation benchmarking is the speedup. This measure is calculated as the ratio between the time it takes for a task to run in one processor divided by the time it takes for the same task in \( n \)
processors

\[ S(n) = \frac{T_1}{T_n} \]  \hspace{1cm} (3.1)

If, for example, a task \( K \) takes 1 hour to execute with one processor and takes 15 minutes when executed in 6 processors then the speedup for \( n = 6 \) is calculated as

\[ S(6) = \frac{T_1}{T_6} = \frac{60}{15} = 4 \]

which basically states that the task \( K \) runs four times faster in 6 processors than in one.

When measuring the speedup of parallel algorithms it is always desirable that this measure for the task \( K \) be close to linear. When implementing a task in parallel, if the implementation does exactly the same computations that the sequential one except that it uses \( n \) processors to do them instead of 1, it will not be possible to achieve a speedup greater than the number of processors \( n \) (linear speedup). Under extra ordinary circumstances a task may show speedup greater than linear, this phenomenon is known as supra-linear speedup and usually reflects an inefficient or suboptimal sequential implementation. Under the most common circumstances the reverse is true: linear speedup is not possible to achieve due to intrinsically sequential operations in the algorithm or overhead introduced by the parallel implementation (ie: transmission of information between processors) that did not exist in the sequential code.

To calculate the maximum speedup that a given task can achieve we can assume that the task spends a fraction of time \( f \) on intrinsically sequential code. Using this
assumption, the time it takes for \( n \) processors to run would be

\[
T_n = fT_1 + \frac{(1 - f)T_1}{n}
\]  

(3.2)

which gives a speedup of

\[
S(n) = \frac{T_1}{fT_1 + \frac{(1 - f)T_1}{n}} = \frac{n}{1 + (n - 1)f}
\]

(3.3)

This equation is known as Amdahl’s law \cite{Amd67} and provides an upper bound on the speedup of parallelization since

\[
\lim_{n \to \infty} \frac{T_1}{fT_1 + \frac{(1 - f)T_1}{n}} = \frac{1}{f}
\]

(3.4)

If, for example, the intrinsically sequential fraction of a task \( K \) is \( f = 0.05 \) then the speedup will be upper bounded by

\[
\frac{1}{f} = \frac{1}{0.05} = 20
\]

regardless of the amount of processors we invest in solving the problem.

This seems like a rather pessimistic estimate, experience shows that when we parallelize a problem we don’t necessarily do it to solve the same problem faster, but instead what we are usually interested in is in solving larger problems in a reasonable amount of time. This is specially true in clusters of workstations where the type of problems presented to them, like molecular conformation and numerical simulation, are much larger than what can reasonably be solved on a sequential machine. Under these circumstances Amdahl’s law looses some of it’s validity and it becomes more
reasonable to use Gustafson’s law [Gus88]. Gustafson assumes that the sequential part of a given task $K$ takes a constant amount of time $s$ regardless of the size of the problem and that the rest of the task can be divided in blocks of size $p$ depending on the size of the problem. So by increasing the size of the problem we increase the speedup factor, this is called the \textit{scaled speedup factor} and the formula is

$$S_s(n) = \frac{s + np}{s + p} = s + np = n + (1 - n)s$$ (3.5)

under the assumption that $s$ and $p$ are fractions so that $s + p = 1$.

Other measures of parallel execution performance are:

- \textit{Efficiency}: This is the ratio of the execution time of one processor over the execution time of $n$ processors multiplied by $n$

$$E(n) = \frac{T_1}{nT_n}$$ (3.6)

notice that if the speedup is not supra linear then the efficiency will always be less than or equal to 1. An efficiency of 1 implies optimal conditions.

$$E(n) \leq 1$$

- \textit{Cost}: This corresponds to the execution time multiplied by the number of processors used.

$$\text{Cost}(n) = nT_n = n\frac{T_1}{T_n} = \frac{nT_1}{S(n)} = \frac{T_1}{E(n)}$$ (3.7)
CHAPTER 4
PROPOSAL

4.1 Literature Review

There have been many contributions to the ART/ARTMAP literature over the last
decade. Carpenter and Ross [CR95] modify the ARTMAP architecture and add
spacial and temporal evidence integration for dynamic predictive mapping in the
ART–EMAP algorithm. Williamson modifies the ART architecture by adding Gauss-
sian discriminant classifiers to create Gaussian–ARTMAP [Wil96] and demonstrates
that this variant performs better on noisy data. Carpenter modifies the winner take
all mechanism employed in ART/ARTMAP improving classification performance by
implementing distributed learning in dART [Car97] and dARTMAP [CB98]. Car-
penter and Markuzon propose ARTMAP–IC, another distributed memory scheme
which also makes use of instance counting for prediction [CM98]. Verzi et al, add
tunable misclassification tolerance to ARTMAP in Boosted–ARTMAP [VHG98] to
address the category proliferation problem and improve classification performance on
noisy data. Gómez et al also address the category proliferation problem by using
mutual information and entropy related concepts in μARTMAP [SDI00]. Anagnosto-
topoulos and Georgiopoulos modify category representation in FAM by using an eu-
clidean distance function and templates represented by ellipsoids in multi–dimentional
space in the Ellipsoid-ART/ARTMAP algorithm [AG01]. Anagnostopoulos et al uses
semi-supervised learning to improve classification performance in noisy gaussian data [ABG03].

The above contributions revolve around modifications and enhancements of the original FAM architecture. However there are other, independent developments of similar ART-like structures. Simpson develops a Fuzzy Min-Max neural network [Sim92] that is very similar to FAM but allows contraction of category templates to avoid overlap sacrificing ARTMAP’s formally guaranteed learning stability. Caudell and Healey develop the LAPART-2 architecture which learns in no more than two passes over the training set and doesn’t suffer from the category proliferation problem. Petridis et al use a fuzzy lattice in the $\sigma$-FLNMAP algorithm [PKF01], a neural network developed specifically for text classification.

On the other hand, when we review the algorithms used for the processing of large databases, we can see that most of these tend to be ad-hoc. Agrawal and Srikant address the problem of mining association rules in large sales transaction databases [AS94]. This approach can be used for classification by mining associations of transaction attributes with their class label. They develop the AprioriHybrid algorithm and show that it scales linearly with the number of transactions in the database. King, Fend and Sutherland compare eighteen algorithms on a series of twelve moderately large databases (20,000 to 58,000 patterns) [KFS95] which show that tree based algorithms are usually faster than other approaches although their classification performance is not necessarily better. Mainly for their speed decision tree based algorithms like C4.5 by Quinlan [Qui93] and CART by Breiman et al [BFO84], tend to be very popular for the processing of large databases. Variations of these tree learning algorithms specifically tailored for large database processing have been proposed. The SLIQ algorithm by Mehta, Agrawal and Rissanen modifies the
concept of tree based algorithms to eliminate some of the memory resident restrictions of traditional implementations. The SPRINT algorithm by Shafer, Agrawal and Mehta [SAM96], builds on top of the SLIQ algorithm concept to eliminate all memory resident restrictions that impede the handling of even larger databases, with the added bonus that the index data structures that it uses for tree generation are easily amenable for parallel implementation. Cohen modifies the IREP algorithm [FW94] and proposes the rule extraction RIPPER algorithm [Coh95] that scales in the order of $O(n(\log n)^2)$ where $n$ represents the number of patterns in the training set. This algorithm has comparable classification performance to C4.5 [Die97].

The backpropagation neural network architecture which was originally proposed in [RHW86] and it’s corresponding algorithm used to train these type of networks can be characterized mathematically by matrix and vector multiplications. These mathematical structures have been parallelized with extensive success. Mangasarian and Solodov [MS94] show that parallel backpropagation has the same convergence properties as the sequential one, guaranteeing the correctness of the parallel algorithm. Torresen et al [TT98] propose an implementation of the backpropagation algorithm on a MIMD paradigm based on a 2D-torus mesh network of processors. Their approach permits pipelining, training set and node parallelism. Also Torresen and Tomita [TNT95] provide an extensive overview of parallel implementation strategies and techniques for this particular neural network architecture.

For ART based neural networks [CGR91a] we can find the work of Manolakos [Man98] who implements with very little modification the non-supervised learning ART1 architecture on a transputer ring of processors. To do this Manolakos divides the communication in two bidirectional rings, one for the $F_1$ layer and another for the $F_2$ layer. Learning examples are pipelined through the ring to optimize network
utilization. Experimental results of his study indicate close to linear speed-up as a function of the number of processors. This approach is efficient for ring networks and was originally implemented in a transputer. It’s and open question if it can be extended for FAM. Zhang [Zha98] shows how a fuzzy competitive neural network similar to ARTMAP can be implemented using a systolic array, Asanović [ABK98] et al use the special purpose parallel vector processor SPERT-II to implement back-propagation and Kohonen neural networks.

4.2 Justification

As can be seen in the literature results, neural network algorithms usually have slower training times than other machine learning algorithms, especially when they learn from large databases. Even one of the fastest (in terms of training time) neural network algorithms, the FAM algorithm [CGM92], tends to exhibit slow convergence time as the size of the network increases.

One obvious way in which this problem has been addressed is by the use of parallelization. On this area we find that extensive research has been done on the properties and implementation of parallel feed–forward multi–layer perceptrons trained with the backpropagation algorithm. We also find some work on the parallelization of unsupervised ART structures but no work at all on supervised learning ARTMAP like structures. We speculate that one possible reason for this is that the competitive nature of the network plus its matchtracking mechanism make the parallelization of the process harder to implement than backpropagation or the unsupervised ART learning algorithms.
None of the previous ART/ARTMAP like neural network implementations address the issue of learning with very large data sets. On the other hand, data mining algorithms, as mentioned before, have addressed these issues with success. As the size of electronic databases grow, there is an increasing need to process considerably larger databases. It becomes of primary concern to bring down the complexity of the algorithm to polynomial, or logarithmic time.

There is little overlap between the neural network parallel implementations and the large data base/data mining implementations. As stated previously this is probably because most neural network algorithms, and specially backpropagation, are notoriously slow to converge, making the use of very large databases out of the question.

It is also worth mentioning that most of the above algorithms assume that the training set is off line, finite, and exists prior to training. The ART/ARTMAP like algorithms, on the other hand, are online algorithms, capable of absorbing new information without disrupting previously learned classifications and of learning from an infinite stream of immediately available data. Aside from the obvious benefits that online execution provides, we can, if we choose to, use equivalent performance measures for both types of algorithm, but the obvious performance measure for offline algorithms is the training time over the full data set while the best performance measure of an online algorithm is it’s response time. That is to say: the time it takes for the algorithm to absorb and learn a new pattern. This relaxing of restrictions on the online learning algorithm’s efficiency opens up the possibility of using more complex and sophisticated classification methods.
All the aforementioned ART/ARTMAP like architectures have similar characteristics to the FAM neural network. In this dissertation our focus is to improve the convergence speed of ART-like structures through different partitioning and parallelization approaches. We chose to demonstrate the effectiveness of our proposed approach to FAM, since, in a way, FAM has been the catalyst for the development of most of the aforementioned ART structures. If our approach shows merit for FAM, its extension to other ART structures can be accomplished without a lot of effort.

FAM has desirable characteristics, but requires modifications for the handling of large data sets. FAM has already demonstrated it’s potential for moderately sized databases [Ana00], but FAM’s performance deteriorates when the training set grows. Furthermore, knowing that FAM is already an online algorithm we believe that it is a relevant topic to pursue the study of FAM’s characteristics and it’s parallel variants in large data sets while preserving it’s online properties.

4.3 Contribution

In this dissertation we develop and implement three new FAM based parallel variants. Two of these variants divide the FAM algorithm by doing data partitioning of the training set, these will be explained and tested on three databases in chapter 5. The third variant uses the technique of network partitioning by dividing the templates of the neural network amongst a pipeline of processors and is analyzed and tested in chapter 6.

Using our formal characterization of the computation complexity of FAM we show analytically and experimentally that the data partitioning approaches achieve a sig-
significant quadratic work reduction because the nature of the problem is altered. Ex-
periments show that the processing of large databases in the order of half a million
data points reduced learning time from days to minutes. We also show that the prop-
erties of the FAM algorithm dealing with convergence and classification performance
in these data partitioning variants are preserved.

Concerning the network partitioning approach, we show that it does not alter
significantly the nature of the problem and we show analytically and experimentally
that it achieves close to linear speedup in the number of processors. It is essentially
equivalent to FAM training without matchtracking. We develop and prove fourteen
theorems concerning the formal properties of the network partitioning pipelined FAM.
These theorems are divided into correctness properties and performance properties of
the algorithm. Experiments conducted on one database also show that classification
performance is comparable to FAM.
CHAPTER 5
DATA PARTITIONING APPROACHES

5.1 Introduction

Let us start by recalling the FAM algorithm and the FAM pseudo-code (training portion of the algorithm) that allows us to better understand what really contributes to FAM’s complexity. Based on this understanding we propose the partitioned FAM approach to reduce the complexity of FAM training when it deals with large databases. The FAM pseudocode is shown in figure 5.1.

Since FAM clusters the patterns into boxes in the $M_a$ dimensional space and uses the $L_1$ distance to compare templates with patterns, it is reasonable to assume that a data partitioning scheme that divides the hyperspace into subspaces will produce good results with FAM. Our interest in providing a data partitioning method for the FAM algorithm made us contemplate different options. Well known methods are the use of principal component analysis and sampling of the data to select the initial clusters.

Many different approaches can be proposed to partition the data set between the processors. The key to success of any of these techniques is being able to divide the data set into equally sized data partitions that do not interfere with each other. The difficulty is that there is no fast and easy way to divide the data set into equally sized
FAM-Training-Phase(\{I^1, I^2, \ldots, I^P\}, epochs, \bar{\rho}_a, \beta_a, \varepsilon)

1. \( w_0 \leftarrow \{1, 1, \ldots, 1\} \)
2. templates \leftarrow \{w_0\}
3. iterations \leftarrow 0
4. repeat
5. \( modified \leftarrow \text{FALSE} \)
6. for each \( I^r \) in \( \{I^1, I^2, \ldots, I^P\} \)
7.  do \( \rho_a \leftarrow \bar{\rho}_a \)
8.  repeat
9.  \( T_{max} \leftarrow 0 \)
10. for each \( w^a_j \) in templates
11.  do if \( \rho(I^r, w^a_j) \geq \rho_a \) and \( T(I^r, w^a_j, \beta_a) > T_{max} \)
12.     then \( T_{max} \leftarrow T(I^r, w^a_j, \beta_a) \)
13.     \( w_{j_{max}} \leftarrow w_j \)
14. if \( w_{j_{max}} = w_0 \) or \( \text{class}(I^r) = \text{class}(w^a_{j_{max}}) \)
15.    then \( status \leftarrow \text{Allocated} \)
16.    else \( status \leftarrow \text{TryAgain} \)
17.    \( \rho_a \leftarrow \rho(I^r, w^a_{j_{max}}) + \varepsilon \)
18. until \( status \neq \text{TryAgain} \)
19. if \( w_{j_{max}} \neq w_0 \)
20.  then
21.      if \( w^a_{j_{max}} \neq w^a_{j_{max}} \wedge I^r \)
22.        then \( w^a_{j_{max}} \leftarrow w^a_{j_{max}} \wedge I^r \)
23.        \( modified \leftarrow \text{TRUE} \)
24.  else
25.      templates \leftarrow templates \cup \{I^r\}
26.      \( iterations \leftarrow iterations + 1 \)
27. until \( modified = \text{FALSE} \) or \( iterations = epochs \)
28. return templates

Figure 5.1: Simplified FAM learning phase
partitions without analyzing the data set, and this can be a time consuming task, usually in the order of \( O(P^2) \) where \( P \) is the number of patterns in the training set.

On this chapter we propose 2 data partitioning schemes for the FAM algorithm, the first one we call \textit{Boxed–FAM} and applies a partitioning on a projected subspace of the data. The second partitioning method relies on the clustering properties of the Hilbert space–filling curve to select the partitions.

### 5.2 Data partitioning

If we recall the online FAM time complexity formula in equation 2.46 given by

\[
\text{Time}(\text{FAM}) = O(\kappa \Gamma P^2)
\]

where \( \kappa \) is the compression ratio, \( \Gamma \) is the matchtracking factor and \( P \) is the number of training pairs in the training set, we can see from this formula that the convergence of the FAM algorithm is quadratic with respect to \( P \), the number of patterns in the training set. This relationship suggests that it may be beneficial to use methods that reduce the size of the training set in FAM to speed up the convergence of the algorithm, and that by doing so we may get a quadratic speedup on the time complexity of the algorithm.

If, for example, we divide the training set into \( p \) partitions and train \textit{each partition} independently, and assume that we divide the training set into equally sized partitions, we would reduce the number of patterns in each partition to \( \frac{P}{p} \) and, if the above
equation holds for each one of the partitions, we will also reduce the number of templates in each partition to \( \frac{n^p}{p} \) on average.

We must emphasize, though, that under this mode of operation we are altering the FAM algorithm and can no longer say that we are implementing the same learning scheme. FAM requires that we compare each learning pattern to every template in the network and the partitioned FAM variants will only compare it to a subset of the total amount of templates. This implies that to provide a fair comparison between the algorithms any tests done on the partitioned Fuzzy-ARTMAP variants must not only measure the speedup of the algorithm obtained but must also verify if the performance of the original FAM algorithm is preserved. In particular, we will be interested in preserving FAM’s generalization capability, classification performance and compression ratio while improving upon FAM’s convergence time.

5.2.1 Partitioned FAM speedup

Even though the partitioned FAM is not strictly the same algorithm as FAM, we can still measure the speedup that one algorithm has over the other by using a formula analogous to the speedup formula of equation 3.1. For this purpose we will introduce the following definitions.

- \( T_1 \): the time it takes for Fuzzy-ARTMAP to perform one epoch of training with a training set of size \( P \) on a sequential machine.
• $T_1^p$: the time it takes for a partitioned Fuzzy–ARTMAP variant to perform one epoch of training with a training set of size $P$ on a sequential machine by dividing the problem into $p$ partitions.

• $T_n^p$: the time it takes for a partitioned Fuzzy–ARTMAP variant to perform one epoch of training with a training set of size $P$ on a machine with $n$ processors by dividing the problem into $p$ partitions with a balanced amount of partitions per processor.

• $T_n$: the time it takes for a partitioned Fuzzy–ARTMAP variant to perform one epoch of training with a training set of size $P$ on a machine with $n$ processors with one partition per processor.

In the general case we will leave the size of the training set $P$ implicit in the equations and assume that it is the same training set size for all the formulas. We will use $T_n^p$ to denote the time it takes to perform one epoch training with a training set of size $P$ using $n$ processors and $p$ partitions. For the sequential machine time the number of processors will be $n = 1$, and if we are using a single partition (unmodified FAM) then we will drop the super-index $p$.

5.2.1.1 Partitioned FAM speedup on a sequential machine

Using the previous definitions we can calculate the speedup on a sequential machine as

$$\frac{T_1}{T_1^p}$$

(5.1)
From equation 2.46 we know that the number of patterns learned in the unmodified version of FAM is $P$ while the number of patterns in the modified version will be reduced to $\frac{P}{p}$ per partition. This gives the speedup of

$$\frac{T_1}{T_p} = \frac{\kappa \Gamma P^2}{\kappa \Gamma \left( \frac{p}{P} \right)^2} = p$$

(5.2)

5.2.1.2 Partitioned FAM speedup on a parallel machine

To get the speedup on a parallel machine we first assume that the number of processors is equal to the number of partitions, this leads to the equation

$$\frac{T_1}{T_p}$$

(5.3)

Using the results in equation 2.46 this gives an optimum speedup of

$$\frac{T_1}{T_p} = \frac{\kappa \Gamma P^2}{\kappa \Gamma \left( \frac{P}{p} \right)^2} = p^2$$

(5.4)

If we chose to have different amount of partitions than processors then the only sensible scenario is when the number of partitions is a multiple of the number of processors

$$(p \equiv 0) \mod n \Rightarrow p = nc; \quad c \in \{1, 2, \cdots\}$$
and the speedup becomes

\[
\frac{T_1}{T_p} = \frac{\kappa \Gamma P^2}{n \kappa \Gamma \left( \frac{P}{p} \right)^2} = np = cn^2
\]

5.2.2 Assumptions

The previous speedups in equations \([5.1]\) and \([5.3]\) still hold even if we do not assume that
the number of times that the repeat loop is executed (i.e., the parameter \(\Gamma\)) is fixed. In
that case we expect \(\Gamma\) to get smaller as the number of templates that a pattern has
to go through decreases. So we expect the \(\Gamma\) in the numerator of equations \([5.1]\) and
\([5.3]\) to be larger than the \(\Gamma\) in the denominator of these equations; thus resulting in
higher than the \(p/p^2\) speed-up that equations \([5.1]\) and \([5.3]\) predict. Furthermore, we
also assumed that the parameter \(\kappa\) is fixed. In reality, it is likely that the parameter
\(\kappa\) will become smaller as the size \(P\) and the number of templates is reduced. So it is
reasonable to expect speedups greater than \(p\) for \(p\) partitions run in sequence or \(p^2\)
in parallel.

Furthermore, additional assumptions that were needed to make the speed-up equa-
tions valid are as follows:

1. The partitioning scheme is well balanced and distributes the learning task fairly
amongst the different partitions.

2. The partitioning scheme is not computationally expensive so as to outweigh its
benefits.
Whatever approach is used to partition the data in the training set will lead us into a collection of smaller training sets. Each one of these sets will be used to independently train a different FAM network. The resulting collection of trained FAM partitions is what we refer to as partitioned FAM. In the performance phase of this FAM variant, a test input is presented to the modified FAM and this test input activates only the templates of the partition to which the input pattern belongs. The most active category that passes the vigilance criterion will produce the predicted label of the input pattern. As stated earlier, it is apparent from the above statements, that the modified FAMs and the single FAM trained on the original dataset are two distinct methods for solving the same classification problem. Our intent, with the modification of FAM, in addition to achieving convergence speed-up compared to FAM, is to create a trained network whose size is not larger than the size of FAM, and whose generalization performance is comparable to the generalization performance of FAM.

5.3 Boxed–FAM

Our first partitioning scheme is inspired by the projective clustering approach proposed and implemented in Procopiuc et al [PJA02], and by the natural properties of the FAM algorithm. With this method we project the data contained in the $M_a$-dimensional hypercube to $\hat{M}_a$ dimensions, where $\hat{M}_a \ll M_a$, and we partition the data set by a grid in this projected space. If, for example, we use $\hat{M}_a = 3$ dimensions and a grid size of 10 divisions per dimension, we would divide the 3-dimensional space, on which we projected the original data, into 1000 boxes of side length equal to 0.1. If
each set of data within a box trains a different FAM architecture (partitioned-FAM) we are guaranteed that the number of templates created by each such FAM is not going to be large (especially if the number of boxes is large). It is likely though that the total number of templates created by the partitioned FAM will be larger than the number of templates created by the non-partitioned FAM. Classification performance may be negatively affected by this partitioning approach. To avoid this unwelcome side effect we take advantage of the natural partitioning imposed by the FAM algorithm. We know from theorem 2.4.2 that templates in FAM are geometrically represented by hyper-rectangles.

Furthermore each hyper-rectangle size is restricted by the vigilance parameter $\bar{\rho}_a$ and by the dimensionality $M_a$ of the input patterns. In particular theorem 2.4.2 states that

$$size \left( w^a_j \right) \leq M_a (1 - \rho_a)$$

(5.5)

With this in mind we chose boxes in the projected space of side size $(1 - \bar{\rho}_a)$ by assuming that templates grow more or less evenly across every dimension. This restriction of the template side size to $(1 - \bar{\rho}_a)$ is actually restricting the templates to the size that the FAM algorithm already enforces. This partitioning approach is most effective when the value of the vigilance parameter is large, which also happens to be the case when the number of templates grows the most and tends to slow down the training of the algorithm.
Figure 5.2: A random sample of Forest Covertype data-points. Shown is a random sample of 5,000 data points out of the available 581,012 data-points. The data-points are projected to the first 3 dimensions of the database. Different Colors for the data-points represent different class labels.

Figure 5.3: A Boxed random sample of Forest Covertype data-points. Shown is a random sample of 5,000 out of the available 581,012. The data-points are projected to the first 3 dimensions of the database and partitioned using the boxing scheme with a box side of 0.2.
5.3.1 Projected dimension selection

Since the partitioning of the data into boxes is implemented after the projection of the data on the fewer ($\hat{M}_a$) than the available dimensions ($M_a$) dimensions, choosing the right set of dimensions to project is an issue of importance. We followed two approaches in choosing the fewer dimensions $\hat{M}_a$ to project the data to.

5.3.1.1 Manually selected dimensions

The first scheme we implemented is the simplest and implies manually selecting the dimensions through a training set inspection. In this case, we assume that the manual selection is reasonable and made by a knowledgeable party.

5.3.1.2 Selection of projected dimension by entropy measure

One way of choosing these fewer dimensions on which to project the data is based on the entropy measure criterion. This criterion is used extensively by Quinlan in decision trees [Qui93] for decision attribute selection.

**Definition:** Given a training set

$$\mathcal{S} = \{(I^1, O^1), (I^2, O^2), \ldots, (I^P, O^P)\}$$
we say the relationship $\equiv$ holds between arbitrary input patterns $I^r$, $I^k$ in the training set if:

$$I^r \equiv I^k \iff O^r = O^k$$

Notice that $\equiv$ is a reflexive, commutative, and transitive binary operation, which means that $\equiv$ is an equivalence relation in the set $\mathcal{S}$ and the set

$$\mathcal{S}|_{\equiv} = \{[I^r]: (I^r, O^r) \in \mathcal{S}\}$$

is a partition on $\mathcal{S}$

In particular, if we assume without loss of generalization, that the number of distinct class labels is $d$, then the entropy criterion for a given training set can be calculated by

$$\text{Entropy}(\mathcal{S}) = -\sum_{i=1}^{d} p_i \log(p_i)$$

where $p_i$ corresponds to the probability that a randomly selected input pattern $I^r$ classifies to the $i^{th}$ class and is usually approximated with the unbiased estimator

$$p_i = \frac{\text{number of input patterns of class } i}{|\mathcal{S}|}$$

The entropy measure of equation [5.6] can be considered a measure of randomness and it can be used here to select the dimensions that will be used for the projection of the boxing. What we do here is calculate the entropy of the set $\mathcal{S}$ before we partition the set by a given dimension. Then for every dimension we partition the set $\mathcal{S}$ and calculate a weighted average of the entropy of the subsets. The dimension that maximizes the reduction of the entropy will be the one selected as a projecting
**Dimension-Selection**($\mathcal{S}, \text{available} = \{d_1, d_2, \ldots, d_k\}, \text{dimensions}$)

1. $\text{dimension-set} \leftarrow \{\}$
2. $\text{dims} \leftarrow 0$
3. **while** $\text{dims} < \text{dimensions}$ **do**
   4. **for each** $d$ in available **do**
   5. partition $\mathcal{S}$ using dimensions in $\text{dimension-set} \cup \{d\}$
   6. calculate entropy of partitioned set using equation 5.6
   7. **add the dimension** $d$ to $\text{dimension-set}$ that minimizes the entropy
   8. $\text{dimension-set} \leftarrow \text{dimension-set} \cup \{d\}$
   9. $\text{available} \leftarrow \text{available} - \{d\}$
 10. $\text{dims} \leftarrow \text{dims} + 1$
11. **return** $\text{dimension-set}$

![Figure 5.4: Entropy calculation method](image)

dimension. The pseudocode of such an entropy–based dimension selection is provided in figure 5.4

With any of the previous approaches for dimensionality reduction we can use different boxed sizes depending on the density of points in the different regions of the hypercube. This method allows to even out the number of templates in each partition, giving a better workload balance.
5.4 Hilbert–FAM

5.4.1 Space–Filling Curves

A space–filling curve is a mapping from a unit hypercube $[0, 1]^{M_a}$ to the unit interval $[0,1]$. Mokbel and Aref [MA01] describe them as a “thread that goes through all the points in a space but visiting every point only once.” We will talk about a space filling curve $S_{m}^{M_a}$ as an $m^{th}$–order approximation of the space–filling curve $S$ in the $M_a$–dimensional space. Every $m^{th}$–order space–filling curve approximation has a finite number of segments and connects a finite number of points in the multidimensional space. The grid size $N$ of a space–filling curve will be the number of divisions into which each dimension is split.

There are many space–filling curves available, amongst them we have the Peano curve, the Z curve, the Hilbert curve, the sweep, the scan and the gray curves. An $M_a$–dimensional space–filling curve with grid size $N$ connects $N^{M_a}$ points and has $N^{M_a} - 1$ segments. Figures 5.5 and 5.6 shows the Sweep and Peano space–filling curves respectively. The grid size in these examples is 4, the number of dimensions $M_a = 2$ the number of points that they connect is $4^2 = 16$ and the number of segments is 15.

A curve $S$ is space–filling iff:

$$S \overset{\text{def}}{=} \lim_{m \to \infty} S_{m}^{M_a} = [0, 1]^{M_a}$$ (5.7)

Peano was the first to use space–filling curves, Hilbert generalized the definition to to arbitrary number of dimensions. To be able to characterize their properties
Figure 5.5: $2^{nd}$ Order Sweep space-filling curve.

Graphed with a grid size of $N = 4$, $2^N = 16$ partitions and $2^N - 1 = 15$ line segments.
Figure 5.6: 2\textsuperscript{nd} Order Peano space–filling curve.  
\emph{Graphed with a grid size of $N = 4$, $2^N = 16$ partitions and $2^N - 1 = 15$ line segments.}
[MA01] concentrates on the nature of the segments that connect adjacent points in the space–filling curve by cataloging them as either a *jump*, *contiguity*, *reverse*, *forward* or *still*. Different applications will require different space–filling curves. If, for example, we wanted to access a database by an index in which the order is relevant then a space–filling curve that preserves the order of the dimension will probably be best (low *reverse*, i.e. the Peano curve).

### 5.4.2 The Hilbert space–filling Curve

![1st and 2nd order 2D Hilbert space–filling curve](image)

Figure 5.7: 1st and 2nd order 2D Hilbert space–filling curve.

Moon et al. [MJF01] concentrate on the Hilbert space–filling curve (HSFC) and show that for range queries the HSFC curve produces the least number of splits in an index. This result is a consequence of the property that the HSFC does not have any jumps, is continuous, and that it does not have a bias toward any dimension.
Figure 5.8: 3rd and 4th order of 2D Hilbert space–filling Curve.

We will denote the $m^{th}$–order approximation of the $M_a$–dimensional Hilbert space–filling curve as $\mathcal{H}_{m}^{M_a}$. Examples of the first 4 approximations of the 2–dimensional Hilbert space–filling curve can be seen in figures 5.7 and 5.8. The same first 4 approximations of the 3–dimensional Hilbert space–filling curve can be seen in figures 5.9 to 5.12. The $m^{th}$–order approximation $\mathcal{H}_{m}^{M_a}$ of the HSFC has a grid size of $N = 2^m$. In practice $\mathcal{H}_{m}^{M_a}$ divides the $M_a$–dimensional space into $2^{mM_a}$ boxes and orders them in a contiguous sequence. For a more detailed exposition of the clustering properties of this curve we refer the reader to [MJKF01].

5.4.3 The Hilbert index calculation

There are various algorithms for calculating the Hilbert index of a given $M_a$–dimensional data point. The one we present here is based on [Law00] which is a modification of
Figure 5.9: 1\textsuperscript{st} order 3D Hilbert space–filling curve.

an iterative algorithm originally found in [But71]. This is a table driven algorithm. It
assumes that we are mapping binary numbers into binary numbers, and the precision
of the mapping is limited by the order of the approximation $m$. Any precision can
be achieved by increasing the approximation order $m$. Also this algorithm assumes
that the number of bits in the full representation of the $M_a$–dimensional data point
is the same as the number of bits in the resulting key $r \in [0,1)$. All operations are
fully reversible and this guarantees a one–to–one mapping.

The algorithm requires the following definitions.

- $M_a$ : number of dimensions
- $m$ : the order of approximation
- $N$ : the number of bits in a derived–key, $N = mM_a$
- $r$ : an $N$–Bit binary Hilbert derived–key expressed as a real number in the range $[0,1)$. 

87
Figure 5.10: 2nd order 3D Hilbert space–filling curve.

- **byte**: a word containing $M_a$ bits
- $\gamma^i_j$: $i \in \{1, \ldots, m\}, j \in \{1, \ldots, M_a\}$ a binary digit in $r$ such that
  
  $$r = 0.\gamma^1_1\gamma^1_2 \cdots \gamma^1_{M_a}\gamma^2_1\gamma^2_2 \cdots \gamma^2_{M_a}\gamma^3_1 \cdots \gamma^m_{M_a}$$

- $\gamma^i$: $i^{th}$ binary byte in $r$, $\gamma^i = \gamma^i_1\gamma^i_2 \cdots \gamma^i_{M_a}$
- $a_j$: a coordinate in dimension $j$ of the point

  $$(a_1, a_2, \cdots, a_j, \cdots, a_{M_a})$$

  whose derived–key is $r$, (each $a_j \in [0, 1]$)

- $\alpha^i_j$: a binary digit in a coordinate $a_j$

  $$a_j = 0.\alpha^1_j\alpha^2_j \cdots \alpha^i_j \cdots \alpha^m_j$$

- $\alpha^i$: a concatenation of all the $i^{th}$ entries of the $a_j$’s

  $$\alpha^i = \alpha^i_1\alpha^i_2 \cdots \alpha^i_{M_a}$$
• **principal position**: The least significant position of a byte that is different from the last position of the byte. If all positions are equal then the principal position is $M_a$ (the first or most significant bit is bit 1).

• **parity**: number of bits in a byte that are equal to 1.

Given these definitions we can succinctly state the HSFC mapping as

$$\mathcal{H}_{m}^{M_a}(a_1, a_2, \ldots, a_{M_a}) = 0.\gamma_1^1\gamma_2^2 \cdots \gamma^m$$

where the $\gamma^i$'s are calculated using the algorithm in figure 5.11. In this figure $\oplus$ is taken to mean the bitwise Exclusive–Or (XOR) operation. Notice that all operations can be performed in constant time, and if we fix the order of approximation $m$ we can safely state that $\mathcal{H}_{m}^{M_a}(\cdot)$ is a constant time operation. In practice, the efficiency of the XOR operation makes the time spent in the calculation of the Hilbert index negligible even for large databases.

5.4.4 **The Hilbert space–filling for FAM partitioning**

The FAM algorithm is a distance based algorithm in which all dimensions are treated equally. Fuzzy ART (unsupervised learning), on which FAM is based, is a clustering algorithm that uses the distance function as a means of selecting its templates. Our interest in providing a data partitioning method for the FAM algorithm made us contemplate different options. Naively dividing the space into hyper-boxes has the disadvantage of having to decide which dimensions to select. Using all dimensions is not viable in high dimensional spaces, since then the amount of partitions would be at least $2^{M_a}$ where $M_a$ is the number of dimensions. The entropy measure used in
\textbf{Hilbert}\((\{a_1, a_2, \ldots, a_{M_a}\}, m)\)

1. \(\omega^0 \leftarrow (0, 0, \ldots, 0)\)
2. \(\tilde{\sigma}^0 \leftarrow (0, 0, \ldots, 0)\)
3. \textbf{for} \(i \in \{1, 2, \ldots, m\}\) \textbf{do}
   \begin{enumerate}
   \item \(\omega^i \leftarrow \omega^{i-1} \oplus \tilde{\sigma}^{i-1}\)
   \item \(\bar{\sigma}^i \leftarrow \alpha^i \oplus \omega^i\)
   \item \(\sigma^i \leftarrow \text{shift } \bar{\sigma}^i \text{ left circular } \Sigma_{k=1}^{i-1} (J_k - 1) \text{ times}\)
   \item \(\gamma^i \leftarrow (\gamma_1^i = \sigma_1^i, \gamma_2^i = \sigma_2^i \oplus \gamma_1^i, \ldots, \gamma_{M_a}^i = \sigma_{M_a}^i \oplus \gamma_{M_a-1}^i)\)
   \item \(J_i \leftarrow \text{principal position of } \gamma^i\)
   \item \(\tau^i \leftarrow \text{a byte of } M_a \text{ bits obtained by complementing } \sigma^i \text{ in the } M_a^{th} \text{ position}\)
   \item \(\text{iff it is of odd parity then complement it in the principal position}\)
   \item \(\tilde{\tau}^i \leftarrow \text{shift } \tau^i \text{ left circular } \Sigma_{k=1}^{i-1} (J_k - 1) \text{ times}\)
   \item \textbf{return} \(0.\gamma^1 \gamma^2 \ldots \gamma^m\)
\end{enumerate}

\textbf{Figure 5.11:} Hilbert space–filling curve algorithm.

\textbf{Figure 5.12:} \(4^{th}\) order Hilbert space–filling curve.
decision trees can be used to select dimensions and split points \cite{Qui93} per dimension. However, this approach, which gives priority to some dimensions over others, somehow runs counter to FAM’s learning process in which, using an $L_1$ distance function, all dimensions are treated the same. On the other hand, the Hilbert space filling curve has been successfully used by Lawder \cite{Law00} for distance queries. Moon et al. \cite{MJF01} prove that the Hilbert space–filling curve does not have a bias toward any dimension. We concentrate on the HSFC because its properties make it more compatible with the characteristics of the FAM algorithm. Our claim is that the $M_a$–dimensional distance function is best preserved by a space–filling curve like the HSFC. Points that are close in the index will be close in the $M_a$–dimensional space (the converse is not necessarily true, though).

The approach is the following: we take the set of training pairs $(I^r, O^r)$, apply the Hilbert index $r = \mathcal{H}_m^{M_a}(a)$, where $a$ is the non complement coded part of $I = (a, a^c)$. The resulting values are added to an index file and sorted. Once sorted, the index is split into $p$ contiguous and equally sized partitions, each partition is processed independently. The complexity of the partitioning operation is equal to the complexity of the sorting algorithm used. For any reasonable sorting algorithm this is $O(PT \log(PT))$ and therefore does not add to the complexity of the FAM learning process itself (which was found to be at least $O(PT^2)$).

The $p$ partitions obtained will be completely balanced in the number of patterns they process, although they may have different number of templates depending on the complexity of the classification task in each partition. On the experimental level we found that the calculation of a 5$^{th}$ order Hilbert index for 581,012 patterns of dimensionality 12 of the Forest Covertype database (see experiments) took about 2
seconds. This is a negligible amount of time compared to the training time of the FAM algorithm on the same data.

5.5 Design of Experiments

Experiments for all FAM algorithm variants were conducted on 3 databases: 2 real-world database and 1 artificially-generated databases (Gaussian distributed data). Training set sizes of $1000 \times 2^i, i \in \{0, 1, \ldots, 9\}$, that is 1,000 to 512,000 patterns were used for the training of FAM and partitioned FAM. The test set size was fixed at 20,000 patterns. The number of partitions varied from $p = 1$ (FAM) to $p = 32$ (partitioned FAM). Partition sizes were also increased in powers of 2.

To avoid additional computational complexities in the the experiments (beyond the one that the size of the training set brings along) the values of the ART network parameters $\bar{\rho}_a$, and $\beta_a$ were fixed (i.e., the values chosen were ones that gave reasonable results for the database of focus). For each database and for every combination of $(p, PT) = (\text{partition}, \text{training set size})$ values we conducted 32 independent experiments (training and performance phases), corresponding to different orders of pattern presentations within the training set. As a reminder FAM performance depends on the values of the network parameters $\bar{\rho}_a$, and $\beta_a$, as well as the order of pattern presentation within the training set.

All the tests where conducted on the SCEROLA Beowulf cluster of workstations of the Institute for Simulation and Training. This cluster consists of 64 900MHz machines running with 250MBytes of RAM each. This configuration guaranteed identical conditions on all runs, parallel and sequential. Since there is no
communication between processors the sequential time can be calculated as the addi-
tion of all the partition times, the parallel time is be the maximum of all the partition
times.

The metrics used to measure the performance of our partitioning approach (hFAM) were:

1. Classification performance of partitioned FAM compared with the classification
   performance of FAM (Higher classification performance is better).

2. Size of the trained partitioned FAM compared against the size of FAM (smaller
   size is better).

3. Speedup of partitioned FAM versus FAM, calculated using formulas 5.1 and 5.3
   for the sequential and parallel implementations respectively.

To calculate the speedup we used 2 measures, the first one is total CPU time
of each test and the second one is the total number of iterations of the FAM main
loop. This approach allowed us to check how closely the wall clock speedup values
are correlated with the number of computations performed by the algorithm. A
discrepancy between these measures would mean that we might be obtaining speedup
from other implementation dependent sources (like cache non-linearities or operating
system dependent issues).

A special mention should be made of the letters recognition database from the UCI
machine learning repository. This database was used to benchmark preliminary tests
on the algorithm modification techniques. This database consists of 20,000 points
of 16 dimensional data. It’s relative small size make it a useful benchmark of the
proposed algorithm modifications, but most of the comprehensive testing was done
on the other larger databases. Some results pertaining to boxed FAM are reported for the letters recognition database.

5.5.1 Forest CoverType Database

The first database used for testing hFAM was the Forest CoverType database provided by Blackard [Bla99], and donated to the UCI Machine Learning Repository [Uni03]. The database consists of a total of 581,012 patterns each one associated with 1 of 7 different forest tree cover types. The number of attributes of each pattern is 54, but this number is misleading since attributes 11 to 14 are actually a binary tabulation of the attribute Wilderness-Area, and attributes 15 to 54 (40 of them) are a binary tabulation of the attribute Soil-Type. The original database values are not normalized to fit in the unit hypercube. Thus, we transformed the data to achieve this. There are no omitted values in the data.

For the purpose of generating the Hilbert index the binary attributes, Soil-Type and Wilderness-Area where re-packed and the database was treated as if it consisted of only 12 real valued attributes. For the purposes of FAM learning the patterns were trained with the dimensionality of 54. Patterns 1 through 512,000 were used for training. The test set for all trials were patterns 561,001 to 581,000. A visualization of the first 3 dimensions of the Forest Covertype database can be seen in figure 5.2. Different tones correspond to different classes. As it can be seen from the figure the class boundaries are quite complex. Classification performance of different machine learning algorithms for this database has been reported in the range of 75%.
5.5.2 Gaussian Databases

The Gaussian data was artificially generated using the polar form of the Box–Muller transform with the R250 random number generator by Kirkpatrick and Scholl [KS81]. We generated 2-class, 16 dimensional data. All the dimensions are identically distributed with the same mean $\mu$ and variance $\sigma^2$ except one. The discriminating dimension has offset means so that the overlap between the Gaussian curves is $\alpha$. The value of $\alpha$ was set at 5% for one database and at 15% for the other. 532,000 patterns were generated for each Gaussian database. 512,000 patterns were used for training; the remaining 20,000 patterns were used for testing.

5.6 Boxed Partitioning Experimental Results

To prove the feasibility of our approach a number of tests were conducted using 2 databases from [Uni03]: the letter recognition database from David Slate and the Covertype database by Jock A. Blackard [Uni03]. On both databases the size of the data used for training was increased by a factor of 2 starting from 1000 data points and ending with 16000 data points for the letters database and starting with 1000 data points and ending with 512,000 for the CoverType data. Classification performance was evaluated with a fixed set of 4,000 patterns for the letters database and with fixed set of 20,000 patterns for the CoverType data.

Comparisons of the training performance of the partitioned FAM approach and the non-partitioned FAM approach on the aforementioned databases were conducted. The training performance was based on two measures. Time that it took for the net-
works to undergo one epoch of training and generalization performance of the trained
networks on the chosen test sets (see Figures 5.19 to 5.20). The dimensions to project
the data in the partitioned-FAM approach were chosen manually by simply observing
the range, variation of the values of the datasets across the chosen dimensions. Au-
tomatically choosing the dimensions on which to project using the entropy measure
and histogram selection was also investigated but the results were not that different
from the hand-picked approach.
Figure 5.13: Forest Covertype database error rate by training set size.
Figure 5.14: Forest Covertype database speedup.
Figure 5.15: Gaussian %5 overlap database speedup.
Figure 5.16: Classification performance in the Forest Covertype database.
Figure 5.17: Number of templates generated in the Forest Covertype database.
For the letters database a baseline vigilance parameter value of 0.85 was used. This creates a partitioning scheme where the reduced input space of 3 dimensions was partitioned into boxes of side length equal to 0.15; this results in a total of $343 = 7^3$ boxes. For the CoverType database a vigilance value of 0.96 was used in the training. This gives a partitioning scheme with boxes of side size of 0.04 or $25^3 = 15,625$ boxes in the three dimensions that we used to project the data.

Patterns can see templates in the contiguous boxes. This creates overlap and reduces the number of templates created.

Some of the observations from the results (error rate and training time) of the FAM and partitioned FAM with these databases are: (a) the partitioned FAM reduces the training time compared to FAM, and at times significantly, especially when the size of the training set is large (e.g., CoverType database where the training time is reduced
by a factor of 78 for the largest training set size) (b) the generalization performance of the partitioned FAM is inferior to FAM especially for training sets of smaller size. This effect can be countered by allowing each FAM in the partitioning approach to be influenced by the boxed regions that are neighbors to its associated region (referred to in the figures as the boxed with neighbors approach).
Figure 5.19: Letters database error rate by training set size.
Figure 5.20: Forest Covertype training duration for different training set sizes.
This creates an overlapping FAM approach that is feasible in the sequential machine but creates communication issues in the parallel approach. Figure 5.18 shows a representation of boxing with neighbors in 2 dimensions. The improvement obtained in generalization of the partitioned FAM with neighbors is made at the expense of increasing its computational complexity. A comparison of the classification performance of FAM, Boxed–FAM and Boxed–FAM with neighbors in the letters database can be seen in figure 5.19.

### 5.7 Hilbert Partitioning Experimental Results

The first set of results that we produced and we are reporting here evaluate whether the size of the resulting hFAM architecture (the one that relies on Hilbert partitioning and training of many FAMs) is larger or smaller than the size of the FAM architecture (trained on the entire training dataset). In figure 5.21 we show a bar graph of the number of templates on the Z axis, the training set size on the X axis increasing from left to right (in thousands of patterns), and the number of partitions of the training dataset on the Y axis increasing from front to back. All the following graphs unless otherwise stated have this format with the measured variable of interest in the Z dimension and the controlled variables training set size and number of partitions in the X and Y dimensions, respectively. It is evident from this graph that the number of data partitions up to the quantity tested does not significantly affect the size of hFAM. In other words we see that the ratio of the number of templates (representing the size of the ART architecture) in hFAM versus the number of templates in FAM is slightly higher than 1. Nevertheless, it is worth noting that hFAM consistently
creates larger size ART architectures compared to the sizes of the architectures that FAM creates. The fact that the number of data partitions has little effect on the size of the hFAM architecture created is also true for the Gaussian data, and confirmed in figure 5.22.

The generalization performance of the forest type database can be seen on figure 5.23. The classification performance of hFAM is very similar with the classification performance of FAM. In fact, the classification performance curve for the hFAM is smoother than the one attributed to FAM. Also the hFAM attained a slightly better classification than FAM, reaching a peak of 76.63% for 32 data-partitions and 512,000 patterns. The Tree Covertype database classification performance consistently improves up to 512,000 patterns. This phenomenon clearly indicates that the database is complex enough to be needing a large training set size to achieve high classification performance levels. As expected, this behavior is not observed with the Gaussian artificial data (5% or 15% data), whose classification performance peaks at 32,000 patterns (figure 5.24). Beyond this number of training input patterns the classification performance graph is flat. Also classification improved considerably by partitioning regardless of the amount of patterns used for training. This is a consequence of the artificial nature of the data since the best split point for the classes coincides with the first split point obtained by Hilbert partitioning.

The turn–around time of the Forest Covertype data is presented in figure 5.25. We can compare this graph with the graph counting the number of iterations in figure 5.26 and corroborate that their shape is almost identical. Also the difference in training time for the algorithm with 512,000 patterns and 32 partitions (26 seconds) and FAM (4 hrs 7 min) is very dramatic.
Figure 5.27 shows the speedup in turn–around time of hFAM with $p$ partitions running in parallel using equation 5.3. The best speedups obtained were in the order of 565. If we look at the table of this figure for the speedup values when the database size is 512,000 we can see that the numbers seem to increase quadratically as the number of partitions $p$ is increased from 1 up to 16. At $p = 32$ there seems to be an anomaly and a dip in the speedup. The speedup measured using number of iterations can be seen in figure 5.28, here the best speedup is in the order of 100 and is not obviously quadratic. Nevertheless, we provide the log–log graph of the same data on figure 5.29, this figure clearly shows a slope on the graph close to 2 for the largest (and most representative for our purposes) training set size, which indicates that the speedup in terms of the number of iterations is as predicted close to $p^2$ for the parallel implementation. We speculate that the difference in speedup obtained between these 2 measures is due to platform restrictions that slow the sequential FAM algorithm when the size of the database is too large.

The speedup measured in iterations for the same data using a sequential processing machine can be seen in figure 5.29. We can see from this figure that the speedup for an single processor is in the order of $p$. Again the speedup observed from 16 to 32 partitions does not follow the same progression for this data.
Figure 5.21: Number of templates of hFAM and FAM for the Covertype data.  
X axis shows thousands of training patterns, Y axis shows number of partitions and Z axis shows number of templates.
Figure 5.22: Number of templates in hFAM and FAM for the 5% overlap data. 

*X axis shows thousands of training patterns, Y axis shows number of partitions and Z axis shows number of templates.*
Figure 5.23: Classification of hFAM and FAM for the Forest Cover data.

X axis shows thousands of training patterns, Y axis shows the % of correctly classified patterns in the test set.

111
Figure 5.24: Classification of hFAM and FAM for 5% Overlap Data.

X axis shows thousands of training patterns, Y axis shows the % of correctly classified patterns in test set.
Figure 5.25: hFAM versus FAM parallel partitioning elapsed time. 
Results are for Covertype data. X axis shows thousands of training patterns, Y axis shows the number of seconds.
Figure 5.26: hFAM versus FAM parallel partitioning number of iterations. Results are for Covertype data. X axis shows thousands of training patterns, Y axis shows the number of iterations.
Figure 5.27: hFAM versus FAM parallel partitioning time speedup.

Results are for Covertype data. Speedup is calculated using elapsed time in seconds. X axis shows thousands of training patterns. Y axis shows number of partitions and Z axis shows the parallel speedup.
Figure 5.28: hFAM versus FAM parallel partitioning iterations speedup.

Results are for Covertype data. Number of iterations are used to calculate speedup; X axis shows thousands of training patterns, Y axis shows number of partitions and Z axis shows the parallel speedup.

<table>
<thead>
<tr>
<th></th>
<th>1</th>
<th>2</th>
<th>4</th>
<th>8</th>
<th>16</th>
<th>32</th>
<th>64</th>
<th>128</th>
<th>256</th>
<th>512</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>16</td>
<td>10.818057</td>
<td>13.3242399</td>
<td>21.9105562</td>
<td>34.0871443</td>
<td>46.2637323</td>
<td>65.7674264</td>
<td>85.2711204</td>
<td>94.8683111</td>
<td>91.9605498</td>
<td>101.164108</td>
</tr>
<tr>
<td>32</td>
<td>10.818057</td>
<td>13.3242399</td>
<td>33.9263073</td>
<td>40.1593525</td>
<td>46.3923976</td>
<td>67.6085845</td>
<td>88.8247713</td>
<td>100.753257</td>
<td>92.9185287</td>
<td>102.063429</td>
</tr>
</tbody>
</table>
Figure 5.29: hFAM versus FAM parallel partitioning speedup (2).

Speedup is calculated using the number of iterations on Forest covertype database. Data is graphed using a LOG–LOG scale to highlight $O(P^2)$ complexity where $P =$ number of patterns. $X$ axis shows the logarithm of the number of partitions, and $Y$ axis shows the $\log_2$ (speedup). Database sizes are represented as $1000 \times 2^i, i \in \{0, 2, \ldots, 9\}$. 

117
Figure 5.30: hFAM versus FAM sequential partitioning speedup (3).

Speedup is calculated using the number of iterations for the Forest Covertype Database. Data is graphed using a LOG–LOG scale to highlight $O(P^2)$ complexity for $P =$ number of patterns. X axis shows thousands of training patterns, Y axis shows the speedup.
CHAPTER 6
NETWORK PARTITIONING APPROACHES

6.1 Introduction

Network partitioning relies on the technique of separating the neural network into different processors. On our implementation of FAM the nodes of the network are represented by the templates $w^a_j$. Therefore in our context, network partitioning implies dividing the templates between the processors in the cluster. Recall FAM’s online complexity formula in equation 2.46 given by

$$Time(FAM) = O(\kappa \Gamma P^2)$$

where $\kappa$ is the compression ratio, $\Gamma$ is the matchtracking factor and $P$ is the number of patterns in the training set. In this equation the number of templates generated in one epoch training was assumed to be

$$|templates| = \kappa P$$

(6.1)

Obviously if we divide the templates between $n$ processors and assume that the division is well balanced then the amount of templates in a processor $i$ will be

$$|templates(i)| = \frac{\kappa P}{n}$$

(6.2)
and when processing each partition in parallel, as long as the overhead is not significant we should expect a speedup in the order of $O(n)$.

The problem though, is not trivially parallelized. Patterns $\mathbf{I}$ in the training set must be compared against every template, which means that either the pattern must travel to all the processors and be compared to the templates that each processor holds, or the templates must travel to where the pattern is. Also, FAM makes the templates $\mathbf{w}_j$ compete in a winner-take-all contest, so the processors must communicate information between them. Once the winner is chosen we must modify the winning template, and this responsibility must be assumed by some processor in the cluster. To optimize the function of FAM in the parallel version it would be beneficial to train more than one input pattern at a time, but the code of the FAM algorithm is intrinsically sequential: a new input pattern is not learned until the previous one has modified it’s corresponding template. All these issues must be addressed by a network partitioning parallel FAM implementation.

This chapter is organized as follows: First we recall how the FAM code works and state explicitly the parameters and notation of the pseudocode. After that we present a no-matchtracking modification of the FAM algorithm developed by Anagnostopoulos [Ana03] that is particularly amenable to parallelization. We then proceed to propose a line structure parallel implementation of Anagnostopoulos no-matchtracking FAM variant. After that we prove useful properties of the Anagnostopoulos FAM variant culminating in a worst workload balance bound. After this proposal we modify the no-matchtracking FAM to accept matchtracking by adding a small variant to the FAM code and we observe that the formal properties of the no-matchtracking FAM version are preserved in the parallel matchtracking FAM, although at the cost of a small digression from the original FAM specification. We conclude the chapter with
experimentation on the no-matchtracking parallel FAM that shows a close to linear speedup while maintaining FAM’s compression ratio and generalization capabilities.

6.2 The FAM pseudo-code

This section contains, for completeness, an algorithmic description of FAM, for a more thorough description of the architecture we refer the reader to [CGM92] and [GFB96] or to chapter 2 in this dissertation.

The FAM algorithm needs the following parameters:

- **Patterns**: This is the set of learning patterns used to train the algorithm, each $\Gamma \in \text{Patterns}$ is of the form $\Gamma = (a, a^c)$ where $a \in [0, 1]^M$ and $M$ represents the dimensionality of $a$.

- $\tilde{\rho}_a$: The baseline vigilance, a learning parameter that controls the size of the category templates, $\tilde{\rho}_a \in [0, 1]$.

- $\beta_a$: Also a learning parameter. The only restriction is that $\beta_a > 0$, but it is common to set it to a small value close to 0.

- **epochs**: This parameter represents the maximum number of times the algorithm is exposed to the training data set. It is common to set $epochs = \infty$ or to set $epochs = 1$. In the first case the algorithm will iterate through the training set until there are no more modifications on the templates. In the second case it will only do one pass over the data set.
• $\varepsilon$: As its notation reveals this is a very small number $\varepsilon > 0$. It is used to guarantee that ineligible templates are excluded from competition during the matchtracking process.

• $w^a_j$: Weight vector in the FAM neural network that emanates from the category representation layer and converges to the input layer.

• $T(\Gamma', w^a_j, \beta_a)$: Activation of node with template $w^a_j$ due to the presentation of input pattern $\Gamma'$.

$$T(\Gamma', w^a_j, \beta_a) = \frac{|\Gamma^a \wedge w^a_j|}{|w^a_j| + \beta_a}$$  \hspace{1cm} (6.3)

• $\rho(\Gamma', w^a_j)$: Vigilance ratio, represents the level of match between an input pattern $\Gamma'$ and the template $w^a_j$.

$$\rho(\Gamma', w^a_j) = \frac{|\Gamma^a \wedge w^a_j|}{|\Gamma'|}$$  \hspace{1cm} (6.4)

The FAM algorithm (learning phase) is shown in figure 6.1. The process we call matchtracking is being performed in the loop of lines 8 through 22. Once a template has won competition, we check if it is of the correct category (line 18) and if not, we increase the vigilance parameter $\rho$ (line 21) so that the winning node is no longer eligible.

Also notice that we repeat the learning process (lines 4 to 30) until we reach a maximum number of iterations or we perform a complete presentation of all the input patterns and no modifications are made to the network. In figure 6.2 (online version of FAM) input patterns are presented once in the neural network architecture.

The performance phase of the algorithm is much simpler. In the performance phase we return the class associated with the template that wins the competition for the input pattern. It is common in this phase to set the parameter $\rho$ to 0 to assure
FAM-LEARNING-PHASE(\{I^1, I^2, \ldots, I^P\}, \hat{\rho}, \beta_a, \text{epochs}, \varepsilon)

1  \text{w}_0 \leftarrow (1, 1, \ldots, 1) \quad \text{\(2M_a\)}

2  \text{templates} \leftarrow \{\text{w}_0\}

3  \text{iterations} \leftarrow 0

4  \text{repeat}

5  \quad \text{modified} \leftarrow \text{FALSE}

6  \quad \text{for each} \ I^r \text{ in} \ \{I^1, I^2, \ldots, I^P\}

7  \quad \quad \text{do} \ \hat{\rho} \leftarrow \rho

8  \quad \quad \text{repeat}

9  \quad \quad \quad \text{T}_{\text{max}} \leftarrow 0

10 \quad \quad \quad \text{status} \leftarrow \text{none}

11 \quad \quad \quad \text{for each} \ \text{w}_a^j \text{ in} \ \text{templates}

12 \quad \quad \quad \quad \text{do if} \ \rho(I^r, \text{w}_a^j) \geq \rho \text{ and} \ T(I^r, \text{w}_a^j, \beta_a) > \text{T}_{\text{max}}

13 \quad \quad \quad \quad \quad \text{then}

14 \quad \quad \quad \quad \quad \quad \text{T}_{\text{max}} \leftarrow T(I^r, \text{w}_a^j, \beta_a)

15 \quad \quad \quad \quad \quad \quad \hat{j}_{\text{max}} \leftarrow j

16 \quad \quad \quad \text{if} \ \text{w}_a^j_{\text{max}} \neq \text{w}_0

17 \quad \quad \quad \quad \text{then if} \ \text{class}(I^r) = \text{class}(\text{w}_a^j_{\text{max}})

18 \quad \quad \quad \quad \quad \text{then} \ \text{status} \leftarrow \text{Allocated}

19 \quad \quad \quad \quad \text{else} \ \text{status} \leftarrow \text{Matchtracking}

20 \quad \quad \quad \quad \quad \rho \leftarrow \rho(I^r, \text{w}_a^j_{\text{max}}) + \varepsilon

21 \quad \quad \text{until} \ \text{status} \neq \text{Matchtracking}

22 \quad \text{if} \ \text{status} = \text{Allocated}

23 \quad \quad \text{then if} \ \text{w}_a^j_{\text{max}} \neq (\text{w}_a^j_{\text{max}} \land I^r)

24 \quad \quad \quad \text{then} \ \text{w}_a^j_{\text{max}} \leftarrow \text{w}_a^j_{\text{max}} \land I^r

25 \quad \quad \quad \quad \text{modified} \leftarrow \text{TRUE}

26 \quad \quad \quad \text{else} \ \text{templates} \leftarrow \text{templates} \cup \{I^r\}

27 \quad \quad \quad \quad \text{modified} \leftarrow \text{TRUE}

28 \quad \text{iterations} \leftarrow \text{iterations} + 1

29 \text{until} \ (\text{iterations} = \text{epochs}) \text{ or} \ (\text{modified} = \text{FALSE})

30 \text{return} \ \text{templates}

Figure 6.1: Restatement of FAM algorithm.
FAM-ON-LINE-LEARNING(\{I_1, I_2, \ldots, I_P\}, \bar{\rho}, \beta_a, \varepsilon)

1. \(w_0 \leftarrow (1, 1, \ldots, 1)^{2M_a}\)
2. \(\text{templates} \leftarrow \{w_0\}\)
3. for each \(I_r\) in \(\{I_1, I_2, \ldots, I_P\}\)
4. do \(\rho \leftarrow \bar{\rho}\)
5. repeat
6. \(T_{\text{max}} \leftarrow 0\)
7. \(\text{status} \leftarrow \text{NoneFound}\)
8. for each \(w_a^j\) in \(\text{templates}\)
9. do if \([\rho(I_r, w_a^j) \geq \rho]\) and \([T(I_r, w_a^j, \beta_a) > T_{\text{max}}]\)
10. then
11. \(T_{\text{max}} \leftarrow T(I_r, w_a^j, \beta_a)\)
12. \(j_{\text{max}} \leftarrow j\)
13. if \(w_{j_{\text{max}}}^a \neq \text{uncommitted}\)
14. then if \(\text{class}(I_r) = \text{class}(w_{j_{\text{max}}}^a)\)
15. then \(\text{status} \leftarrow \text{Allocated}\)
16. else \(\text{status} \leftarrow \text{Matchtracking}\)
17. \(\rho \leftarrow \rho(I_r, w_{j_{\text{max}}}^a) + \varepsilon\)
18. until \(\text{status} \neq \text{Matchtracking}\)
19. if \(\text{status} = \text{Allocated}\)
20. then
21. \(w_{j_{\text{max}}}^a \leftarrow w_{j_{\text{max}}}^a \land I\)
22. else
23. \(\text{templates} \leftarrow \text{templates} \cup \{I_r\}\)
24. return \(\text{templates}\)

Figure 6.2: Restatement of FAM online algorithm.
FAM-Performance-Phase($\mathbf{I}^r$, $\mathbf{templates}$, $\rho$, $\beta$)
1 $T_{\text{max}} \leftarrow 0$
2 $j_{\text{max}} \leftarrow \text{NIL}$
3 \textbf{for each} $w^a_j$ \textbf{in} $\mathbf{templates}$
4 \textbf{do}
5 \hspace{1em} \textbf{if} $\rho(\mathbf{I}^r, w^a_j) \geq \rho$ \textbf{and} $T(\mathbf{I}^r, w^a_j, \beta) > T_{\text{max}}$
6 \hspace{1em} \textbf{then}
7 \hspace{2em} $T_{\text{max}} \leftarrow T(\mathbf{I}^r, w^a_j, \beta)$
8 \hspace{2em} $j_{\text{max}} \leftarrow j$
9
10 \textbf{if} $w^a_{j_{\text{max}}} \neq w_0$
11 \hspace{1em} \textbf{then} \textbf{return} \text{class}($w^a_{j_{\text{max}}}$)
12 \hspace{1em} \textbf{else} \textbf{return} \text{NIL}

Figure 6.3: Restatement of FAM performance phase.

that the network will produce a predicted label (classification) for every input pattern (albeit sometimes erroneous). The FAM performance phase is shown in figure 6.3.

6.3 Anagnostopoulos “No Matchtracking” FAM

Another simplification that can be applied to the FAM algorithm is the elimination of the matchtracking process. This modification was originally proposed by Anagnostopoulos [Ana03] and although it may seem rather imprudent to do this we see that it can actually improve the classification performance of FAM on some databases. Our interest in using this FAM variant lies in that it simplifies the FAM algorithm and allows us to concentrate on the parallelization of the competition loop. Later we will see that we can drop the requirement of eliminating matchtracking if we accept
FAM-NO-MATCHTRACKING-LEARNING($\{I^1, I^2, \ldots, I^P\}$, $\rho_a$, $\beta_a$)

1. $w_0 \leftarrow (1, 1, \ldots, 1)$

2. $templates \leftarrow \{w_0\}$

3. for each $I^r$ in $\{I^1, I^2, \ldots, I^P\}$

4. do $T_{max} \leftarrow 0$

5. $w_{max} \leftarrow \text{none}$

6. for each $w^a_j$ in templates

7. do if $[\rho(I^r, w^a_j) \geq \rho_a]$ and $[T(I^r, w^a_j, \beta_a) > T_{max}]$

8. then

9. $T_{max} \leftarrow T(I^r, w^a_j, \beta_a)$

10. $w_{max} \leftarrow w^a_j$

11. if $w_{max}^a \neq w_0$ and $\text{class}(I^r) = \text{class}(w_{max}^a)$

12. then $w_{j_{max}} \leftarrow w_{j_{max}} \land I^r$

13. else $templates \leftarrow templates \cup \{I^r\}$

14. return $templates$

Figure 6.4: Anagnostopoulos No-matchtracking FAM.

A slight digression from the Fuzzy–ARTMAP sequential behavior. This no-match tracking FAM is depicted in figure 6.4.

6.4 The Beowulf parallel platform

The Beowulf cluster of workstations is a network of computers where processes exchange information through the network’s communications hardware. In our case, it consisted of 64 AMD T-Bird 900MHz processors with 256MB of RAM. The nodes are connected through a Fast Ethernet network.
In general, the Beowulf cluster configuration is a parallel platform that has a high latency. This implies that to achieve optimum performance communication packets must be large and kept to a small number. Parallelization techniques in this platform are radically different from shared memory or vector machines. Also communication between nodes in the cluster is done by consent from all the parties involved; that is all communicating entities must agree to send/receive information in compatible formats. This has an impact on the design of the algorithm because receiving entities must know before-hand that they are going to receive information in order to be prepared to accept it. There is no central coordinating entity and protocols must be based on listening/polling schemes and must dispense of any interrupt driven communication.

We have two choices for parallelization design. We can request from each node in the network to process a different input pattern. Or we can request that each node processes the same input patterns at the same time. If we want the parallel implementation to work equivalently to the sequential one the first design will lead to a pipelined approach where each node computes a stage in the pipeline. The second approach will lead to a star master/slave topology where all nodes communicate to a gathering master node. We chose to follow the pipelined approach because in this scenario we are only doing point to point communication, which is a constant time operation in a Fast Ethernet switched network. The star approach tends to degrade communication performance as the size of the gather operation increases.

Our design is based on fixed packet size communication through the network. No network bandwidth would be gained by using variable sized packets since packets are more efficient when they are large and to find out the size of a packet a receiving process would have to incur an extra (and expensive) communication.
580,000 of Covertype data patterns of dimensionality 55 over a pipeline of varying packet size and number of processors.
To find the optimum packet size for our experiments, we selected the SCEROLA parallel cluster of the Institute for Simulation and Training (IST) as a benchmark, and used a large database to transmit information in a pipeline using different packet sizes. Figure 6.5 shows the transition times of 580,000 patterns of dimensionality 55 (reals) from the Forest Covertype database across a pipeline of processors with different packet sizes (in vectors). As can be seen on the table, the lowest transmission times regardless of number of processors in the pipeline occurred when the packet size was in between 64 to 128 patterns. Equating this number to memory size we get an optimum packet size for this cluster in the vicinity of:

\[(64\ldots128) \times 55 \times 4 = (14080\ldots28160)\text{Bytes}\quad (6.5)\]

6.5 Parallel, no matchtracking, FAM implementation

Anagnostopoulos’ FAM variant is particularly amenable to a production-line style pipeline parallel implementation since patterns can be evenly distributed amongst the nodes in the pipeline. A depiction of the pipeline is shown in figure 6.6. The elimination of matchtracking makes the learning of a pattern a one-pass over the pipeline procedure and different patterns can be processed on the different pipeline steps to achieve optimum parallelization. For the implementation we will introduce the following definitions:

- \(n\): number of processors in the pipeline.
- \(k\): index of current process, \(k \in \{0, 1, \ldots, n - 1\}\).
Figure 6.6: Pipeline Structure.
- \( p \): packet size, number of patterns sent downstream, \( 2p = \) number of templates sent upstream.

- \( i^i \): input pattern \( i \) of current packet in the pipeline. \( i \in \{1, 2, \ldots, p\} \).

- \( w^i \): current best candidate template for input pattern \( i^i \).

- \( T^i \): current maximum activation for input pattern \( i^i \).

- \( myTemplates \): set of templates that belong to the current processor.

- \( nodes \): variable local to the current processor that holds the total number of templates the process is aware of (it’s own plus the other processors).

- \( myShare \): amount of templates that the current process should have.

- \( w^i_{k-1} \): template \( i \) coming from previous process in the ring.

- \( w^i_{k+1} \): template \( i \) coming from next process in the ring.

- \( w^i \): template \( i \) going to next process in the ring.

- \( w^i_{to(k-1)} \): template \( i \) going to previous process in the ring.

- \( I.class \): class label associated with a given input pattern.

- \( w.class \): class label associated with a given template.

- \( \text{index}(w) \): sequential index assigned to a template

- \( newNodes \): number of created nodes on a given iteration to communicate upstream in the pipeline.
Figure 6.7: Initialization procedure for pipelined parallel FAM implementation.

- $new\,Nodes_{k+1}$: number of created nodes on a given iteration communicated from processor $k + 1$ in the pipeline.

The algorithm itself is shown in figure 6.8 and the initialization procedure is shown in figure 6.7.

The pseudocode of Process is the main heart of the algorithm. Each element of the pipeline will execute this procedure. The input parameter $k$ tells the process which stage of the pipeline it is, where the value $k$ varies from 0 to $n - 1$. After initializing most of the values as empty (figure 6.7) we enter the loop of lines 2 through 35. This loop continues execution until there are no more input patterns to process. The first activity of each process is to create a packet of excess templates to send back (line 12 to 14). Lines 7 to 10 correspond to the information exchange between contiguous nodes in the pipeline. The functions Send-Next and Recv-Next on lines 7 and 8 respectively don’t do anything if the process is the last in the pipeline ($k = n - 1$). The same is true for the function Send-Prev when the

\begin{verbatim}
INIT($p$)
1 nodes ← 0
2 myTemplates ← {}
3 $\forall_{i=1}^{2^p} \ (w_{i, to(k-1)} ← \text{none})$
4 $\forall_{i=1}^{p} \ (w_{i} ← \text{none})$
5 $\forall_{i=1}^{p} \ (F_{i} ← \text{none})$
6 myShare ← 0
7 newNodes ← 0
8 newNodes_{k+1} ← 0
9 continue ← TRUE
\end{verbatim}
Process\((k, n, \rho_a, \beta_a, p)\)
1. \text{Init}(p)
2. \textbf{while} continue
3. \textbf{do}
4. \quad \textbf{while} |myTemplates| > myShare
5. \quad \textbf{do}
6. \quad \quad \text{Extract-Template} \left( myTemplates, \{w^i_{to(k-1)} \} \right)
7. \quad \quad \text{Send-Next} \left( k, n, \{(w^i, I^i, T^i) : i = 1, 2, \ldots, p \} \right)
8. \quad \quad \text{Recv-Next} \left( k, n, \{w^i_{k+1} : i = 1, 2, \ldots, 2p \}, \text{newNodes}_{k+1} \right)
9. \quad \quad \text{Send-Prev} \left( k, \{(w^i_{to(k-1)} : i = 1, 2, \ldots, 2p \}, \text{newNodes} \right)
10. \quad \quad \text{Recv-Prev} \left( k, \{(w^i_{k-1}, I^i_{k-1}, T^i_{k-1}) : i = 1, 2, \ldots, p \} \right)
11. \quad \text{newNodes} \leftarrow \text{newNodes}_{k+1}
12. \quad \quad \mathcal{S} \leftarrow \{w^i_{k+1}\}
13. \quad \textbf{for each} \ i \ \textbf{in} \ \{1, 2, \ldots, p\}
14. \quad \quad \text{Find-Winner}(I^i, w^i, T^i, \rho_a, \beta_a, \mathcal{S})
15. \quad \quad \text{myTemplates} \leftarrow \text{myTemplates} \cup \mathcal{S}
16. \quad \textbf{if} I^i_{k-1} = EOF
17. \quad \quad \text{continue} \leftarrow \text{FALSE}
18. \quad \textbf{else}
19. \quad \quad \mathcal{S} \leftarrow \{w^i_{to(k-1)}\}
20. \quad \quad \textbf{for each} \ i \ \textbf{in} \ \{1, 2, \ldots, p\}
21. \quad \quad \quad \text{Find-Winner}(I^i, w^i_{k-1}, T^i_{k-1}, \rho_a, \beta_a, \mathcal{S})
22. \quad \quad \quad (I^i, w^i, T^i) \leftarrow (I^i_{k-1}, w^i_{k-1}, T^i_{k-1})
23. \quad \quad \textbf{for each} \ i \ \textbf{in} \ \{1, 2, \ldots, p\}
24. \quad \quad \quad \text{Find-Winner}(I^i, w^i, T^i, \rho_a, \beta_a, \text{myTemplates})
25. \quad \quad \textbf{if} \ k = n - 1
26. \quad \quad \quad \textbf{then} \textbf{if} \ \text{class}(I^i) = \text{class}(w^i)
27. \quad \quad \quad \quad \text{then} \text{myTemplates} \leftarrow \text{myTemplates} \cup \{I^i \land w^i\}
28. \quad \quad \quad \textbf{else} \ \text{newTemplate} \leftarrow I^i
29. \quad \quad \quad \text{index}(\text{newTemplate}) \leftarrow \text{newNodes} + \text{nodes}
30. \quad \quad \quad \text{myTemplates} \leftarrow \text{myTemplates} \cup \{I^i, w^i\}
31. \quad \quad \quad \text{newNodes} \leftarrow \text{newNodes} + 1
32. \quad \quad \textbf{if} \ \text{newNodes} > 0
33. \quad \quad \quad \text{then} \ \text{nodes} \leftarrow \text{nodes} + \text{newNodes}
34. \quad \quad \quad \ \text{myShare} \leftarrow \lceil \frac{\text{nodes}}{n} \rceil$
35. \quad \text{Send-Next} \left( k, n, \{(\text{none}, \text{none}, 0)\} \right)
36. \quad \text{Recv-Next} \left( k, n, \{w^i_{k+1} : i = 1, 2, \ldots, 2p \}, \text{newNodes}_{k+1} \right)
37. \quad \text{myTemplates} \leftarrow \text{myTemplates} \cup \{w^i_{k+1} : i = 1, 2, \ldots, 2p \}

Figure 6.8: Pipelined FAM ring implementation for parallel processing.
**Find-Winner**($I, w, T, \rho_a, \beta_a, S = \{w^i\})

1. $idx \leftarrow -1$
2. for each $w^i$ in $S$
3. do if $[\rho(I, w^i) \geq \rho_a]$
4. then
5. if $[T(I, w^i, \beta_a) > T]$
6. then
7. $T \leftarrow T(I, w^i, \beta_a)$
8. $idx \leftarrow i$
9. else if $[T(I, w^i, \beta_a) = T]$ and $index(w^i) < index(w)$
10. then $T \leftarrow T(I, w^i, \beta_a)$
11. $idx \leftarrow i$
12. if $idx \neq -1$
13. then
14. $\text{Extract}(w^{idx}, S)$
15. $\text{Add}(w, S)$
16. $w \leftarrow w^{idx}$
17. return TRUE
18. else
19. return FALSE

**Figure 6.9: Utility function**

*Used to find best candidate template in a template list. Needed by parallel FAM ring implementation*
process is the first in the pipeline \((k = 0)\). On the other hand, the function \texttt{Recv-Prev} reads learning patterns from the input stream if the process is the first in the pipeline. These fresh patterns will be paired with an uncommitted node \((1, 1, \cdots, 1)\) with index \(\infty\) as their best representative so far. On all other cases these functions do the obvious information exchange between contiguous processes in the pipeline. We assume that all communication exchange between contiguous processes in the pipeline. We can achieve this in an MPI environment by doing non-blocking sends and using an \texttt{MPI-Waitall} for the receives.

On line 30 we add 2 templates to the template set \texttt{myTemplates}. This is because a new template was created and the current candidate winner \(w\) is not of the correct category and has to be inserted back into the pool of templates.

The function \texttt{Find-Winner} is also important. This function searches through a set of templates \(S\) to find if there exists a template \(w^i\) that is a better choice (using FAMs criteria) for representing \(I\) than the current best representative \(w\). If it finds one it swaps it with \(w\), leaving \(w\) in \(S\) and extracting \(w^i\) from it. By sending the input patterns downstream in the pipeline coupled with their current best representative template we guarantee that the templates are not duplicated amongst different processors and that we do not have multiple-instance consistency issues.

Also when exchanging templates between nodes in the pipeline we have to be careful that patterns that are sent downstream do not miss the comparison with templates that are being sent upstream. This is the purpose of lines 12 to 15 (communication with the next one in the pipeline) and lines 18-21 of \texttt{PROCESS}. In line 12 we set \(S\) to represent the set of templates that have been sent upstream to node \(k\) by node \(k + 1\). We loop through each pattern, template pair \((I, w)\) (lines 13-15) to see if
one of the templates sent upstream has a higher activation that the ones that were sent downstream, if this is true then the template will be extracted from $S$. The net result of this is that $S$ ends up containing the templates that lost the competition, and therefore the ones that process $k$ should keep (line 15). The converse process is done on lines 18 to 21. On line 18 we set $S$ to represent the set of templates that are sent upstream to the previous node $k - 1$ in the pipeline. On lines 19 to 20 we compare the pattern, template pairs $(I_{k-1}^i, w_{k-1}^i)$ that $k - 1$ sent with the templates in $S$ sent to process $k - 1$. On line 21 we set our current pattern, template pairs to the winners of this competition. The set $S$ is discarded since it contains the losing templates and therefore the templates that process $k - 1$ keeps.

Finally, in line 30 of FIND-WINNER we add both the input pattern $I^i$ and the template $w^i$ to the set of templates. This does the obvious myTemplates update except when the template $w^i$ happens to be the uncommitted node in which the addition is ignored.

### 6.6 Properties of the Parallel, no matchtracking, FAM Algorithm

In this section we provide a series of theorems that state the properties of the no-matchtracking parallel algorithm. For convenience we have divided the theorems in two tables. Table 6.6 lists the theorems and their names dealing with the correctness of the algorithm. Table 6.6 lists the theorems dealing with the performance of the algorithm.
### Table 6.1: No-matchtracking FAM CORRECTNESS theorems

<table>
<thead>
<tr>
<th>Theorem</th>
<th>Name</th>
</tr>
</thead>
<tbody>
<tr>
<td>6.6.1</td>
<td>Non-duplication</td>
</tr>
<tr>
<td>6.6.4</td>
<td>Upstream bundle size sufficiency</td>
</tr>
<tr>
<td>6.6.10</td>
<td>Overflow avoidance</td>
</tr>
<tr>
<td>6.6.12</td>
<td>Partial evaluation correctness</td>
</tr>
</tbody>
</table>

### Table 6.2: No-matchtracking FAM PERFORMANCE theorems

<table>
<thead>
<tr>
<th>Theorem</th>
<th>Name</th>
</tr>
</thead>
<tbody>
<tr>
<td>6.6.2</td>
<td>Template awareness delay</td>
</tr>
<tr>
<td>6.6.3</td>
<td>Weak upstream migration precondition</td>
</tr>
<tr>
<td>6.6.5</td>
<td>Strong upstream migration precondition</td>
</tr>
<tr>
<td>6.6.6</td>
<td>Strong upstream migration postcondition</td>
</tr>
<tr>
<td>6.6.7</td>
<td>Template ownership delay</td>
</tr>
<tr>
<td>6.6.8</td>
<td>Network size lower bound</td>
</tr>
<tr>
<td>6.6.9</td>
<td>Template ownership bound</td>
</tr>
<tr>
<td>6.6.11</td>
<td>Pipeline depth invariance</td>
</tr>
<tr>
<td>6.6.13</td>
<td>Workload balance variance bound</td>
</tr>
</tbody>
</table>
Definition 6.6.1. A template $w^n_j$ is in transit if the template has been received from the previous processor in the pipeline and the current process has not made the decision yet of whether to send it to the next process, previous process, or keep it. Templates in transit are stored in the $w^i$’s array.

Definition 6.6.2. A template $w^n_j$ is owned by a processor $i$ in the pipeline if it is stored in the myTemplates array of process $i$.

Theorem 6.6.1. Non-duplication

A template $w$ will either be owned by a single processor, or it will be in transit on a single processor (i.e. only one copy of the template exists in the system).

Proof. First lets notice that templates start their existence in process $n - 1$ in line 30 of PROCESS. Here they are immediately added to the templates of process $n - 1$, so they start belonging to a single process.

Also, templates only change location when

1. They are compared with a given input pattern $I^r$ and selected to represent it, in which case they are deleted from the template list owned by the process and added to the templates in transit.

2. They are in transit and lose competition to another template, in which case they are removed from the templates in transit and added to the owned templates of the process.

3. They are sent upstream or sent downstream as in transit templates.
The only possible situation where the templates may be in two places at once is in situation 3 when they are exchanged between processors in the pipeline, since that is the only scenario where 2 processors hold a copy of the same template.

So the only possible problem will arise when 2 consecutive processes exchange templates. Now a template that is sent downstream on line 7 of PROCESS by a process $k - 1$ is received by process $k$ in line 10 of PROCESS. Every template $w$ that is sent downstream is tagged along with an input pattern $I$. Process $k$ will keep the template in transit if it is the best candidate for input pattern $I$. To verify this, process $k$ will compare template $w$ against the templates that process $k$ sent upstream. If a template $w'$ that was sent upstream is a better candidate than $w$ for the input pattern $I$ (lines 19–21) then process $k$ will discard the template $w$ and keep the same template $w'$.

Concurrently, process $k - 1$ will check the template $w$ and input pattern $I$ pair it sent to process $k$ and compare them against the templates that it receives from process $k$. If a template $w'$ that was received from process $k$ is a better candidate than $w$ for input pattern $I$ (lines 12–15) then process $k - 1$ will keep template $w$ and discard template $w'$.

As we can see these concurrent operations guarantee that a template that was sent downstream not only will not reside in 2 places at the same but will also be compared against all the input patterns that flow through the pipeline.

**Theorem 6.6.2. Template awareness delay**

*The total number of templates (nodes) that a process $k = 0, 1, \ldots, n - 1$ in the pipeline is aware of is equal to the number of templates that existed in the system $n - k - 1$ iterations ago.*
Proof. Consider the last process in the pipeline \((n - 1)\). This process knows immediately when a template is created so it knows how many templates exist \(n - 1 - k = n - 1 - (n - 1) = 0\) iterations ago.

The number of templates created per iteration is sent upstream to the previous process in the variable \(newNodes\). This variable is received by process \(n - 2\) one iteration after the templates have been created, by process \(n - 3\) two iterations after the templates have been created and in general by process \(i\), \(n - 1 - i\) iterations after the templates have been created. This means that a process \(k\) always receives on the current iteration the value of the variable \(newNodes\) that was created \(n - k - 1\) iterations ago, and this implies that process \(k\) is aware of the amount of templates that existed \(n - k - 1\) iterations ago.

\[\Box\]

**Theorem 6.6.3. Weak upstream migration precondition**

A process \(k\) in the pipeline sends templates upstream only if on the current iteration:

\[|myTemplates| > myShare\]  \hspace{1cm} (6.6)

**Proof.** It will suffice to say that PROCESS creates the packet of templates to be sent upstream in lines 12 through 14. Looking at line 13 of the Process pseudocode we can see that templates are packed to be sent upstream only when condition \((6.6)\) is met.

\[\Box\]

**Theorem 6.6.4. Bundle size sufficiency**

The excess templates for a process \(k \neq 0\), at any given time, always fits in the packet size \(2p\) to be sent upstream.

140
Proof. By an excess of templates we mean the number of templates over \( \text{myShare} \). What we need to prove then, is that no process in the pipeline except the first one can have at any point in time an excess of templates greater than \( 2p \). Formally, we have to prove that it is impossible for a process in the pipeline to reach a situation where

\[
|\text{myTemplates}| > \text{myShare} + 2p
\]  

(6.7)

Also let us notice that at the beginning of execution there are no templates in transit and that all the processes have their \( \text{myShare} = 0 \) templates, in other words they comply with the condition 6.8

\[
|\text{myTemplates}| \leq \text{myShare}
\]  

(6.8)

Now let us consider the process \( n-1 \), the last in the pipeline. If this process complies with the equation 6.8 and receives \( p \) templates from the previous process, it would have a total of at most \( p + \text{myShare} \) templates. In the worst case scenario all of the \( p \) templates that have been sent are not of the correct category and will force the creation of another \( p \) templates giving a maximum total of \( 2p + \text{myShare} \) templates where \( 2p \) are in transit. At the beginning of the next iteration, the process will pack \( 2p \) templates to be sent upstream to the previous process in the pipeline (assuming its variable \( \text{myShare} \) does not increase) and will receive \( p \) templates from the previous process. Notice that the \( p \) templates extra that it ended up with are not part of it’s fair share because they are templates in transit, that may or may not produce new templates and again will be sent upstream in the pipeline. It’s actual number of templates \( |\text{myTemplates}| \) has not exceeded \( \text{myShare} \).
Now consider any other process that is not the last or the first in the pipeline and assume (as it does when it starts) that it complies with equation 6.8. This process can receive in the worst case scenario a total of \( p \) templates from the previous process in the pipeline and \( 2p \) templates from the next process in the pipeline. Now the \( p \) templates brought from the previous process in the pipeline will continue their journey to the next process (maybe not the same ones but at least that quantity), so they will not increase the total number of templates that the process owns. The excess of the \( 2p \) templates coming from the next process over \( \text{myShare} \) will be packed and sent to the previous process.

\[ \text{Theorem 6.6.5. Strong upstream migration precondition} \]

If a process \( k \in \{0, 1, \ldots, n-1\} \) in the pipeline sends templates back, then it is true that:

- 1 iteration ago process \( k + 1 \) complied with condition 6.6 and sent templates back.
- 2 iterations ago process \( k + 2 \) complied with condition 6.6 and sent templates back.
  
  \[ \vdots \]

- \( n - 1 - k \) iterations ago process \( n - 1 \) complied with condition 6.6 and sent templates back.

\[ \text{Proof.} \] If process \( k \) sends back templates then by theorem 6.6.3 it complies with condition 6.6. But by the reasoning in theorem 6.6.4 all excess templates fit in the packet size so they are sent upstream on the next iteration that they received. This
means that the excess templates where received from process $k + 1$ one iteration ago. Recursively, if process $k + 1$ sent templates back one iteration ago then by theorem 6.6.3 process $k + 1$ must have complied with condition 6.6 two iterations ago, and this can only happen if 2 iterations ago process $k + 2$ sent templates back. By repeating this argument we can corroborate that process $k + i$ complied with condition 6.6 and sent templates back $i$ iterations ago.

**Theorem 6.6.6. Strong upstream migration postcondition**

If a process $k \in \{0, 1, \ldots, n - 1\}$ in the pipeline sends templates back, then it is true that:

1. in this iteration process $k$ keeps myShare templates.
2. 1 iteration ago process $k + 1$ kept myShare templates.
3. 2 iterations ago process $k + 2$ kept myShare templates.

\[ \vdots \]

4. $n - 1 - k$ iterations ago process $n - 1$ kept myShare templates.

2. All of the values of myShare are the same for all the processes.

3. The templates that each processor keeps are distinct.

**Proof.** First lets notice that by theorem 6.6.2

- on the current iteration process $k$ is aware of the templates that existed in the system $n - k - 1$ iterations ago.
• 1 iteration ago process $k + 1$ was aware of the templates that existed in the system $n - k - 1$ iterations ago.

• 2 iterations ago process $k + 2$ was aware of the templates that existed in the system $n - k - 1$ iterations ago.

  :

• $n - k - 1$ iterations ago process $n - 1$ was aware of the templates that existed in the system $n - k - 1$ iterations ago.

This means that all the processes were aware of the same amount of templates and therefore their values for $myShare$ are all the same. It is evident by looking at lines 12 to 14 of PROCESS that the process keeps $myShare$ templates when it sends templates back. We also know by theorem 6.6.5 that they all sent templates on the corresponding iterations. Now for any pair of processes $k + i$ and $k + j$ where $i < j$, the templates that process $k + i$ keeps $i$ iterations ago cannot be the ones that process $k + j$ keeps $j$ iterations ago since it takes at least $(j - i)$ iterations to transmit templates from $j$ to $i$ and process $k + j$ kept them $j$ iterations ago, which means that they cannot reach process $i$ by $j - (j - i) = i$ iterations ago. □

**Theorem 6.6.7.** Template ownership delay

The templates that on the current iteration a process $k$ has were created at least $n - k - 1$ iterations ago

*Proof.* This is obvious since templates are created in process $n - 1$ in line 30 of the code of PROCESS. These templates are sent back in the pipeline one step of the pipeline per iteration. The distance from $k$ to process $n - 1$ is equal to $n - k - 1$ so the templates that $k$ has must have been created at least $n - k - 1$ iterations ago. □
Theorem 6.6.8. Network size lower bound

If a process $k$ sends templates back on a given iteration, then the number of templates $N$ that existed in the system $n - 1 - k$ iterations ago complies with the next condition:

$$N > (n - k)\text{myShare}$$

(6.9)

Proof. Notice that if process $k$ sends templates back then it complies with condition 6.6 and by theorem 6.6.6 all processes from $k$ on-wards kept myShare templates and these templates are all distinct. Also by theorem 6.6.7 all these templates where created at least $n - k - 1$ iterations ago. So the number of templates that existed in the system $n - k - 1$ iterations ago is at least:

$$N \geq |myTemplates| + (n - 1 - k)\text{myShare}$$

$$> \text{myShare} + (n - 1 - k)\text{myShare} = (n - k)\text{myShare}$$

(6.10)

Theorem 6.6.9. Template ownership bound

A process $k$ in the pipeline cannot own more than myShare templates and cannot own less than $\max(0, \text{myShare} - p(2(n - 1 - k) - 1))$

Proof. That it cannot exceed myShare of templates has already been shown by theorem 6.6.4. That it cannot own less that 0 templates is also obvious. What needs to be proven then is that if $\text{myShare} > p(2(n - k) - 1)$ Then the number of templates will never be less than $\text{myShare} - p(2(n - k) - 1)$ templates.

To prove this let us assume a steady state in the pipeline where node $k$ has myShare templates, and the worst case possible scenario. In this scenario process $k$
would receive from process $k-1$ packets of $p$ pattern/template $(I^i, w^i)$ pairs where the $w^i$ would be the uncommitted node, and would send to the next process packets of $p$ pattern/template pairs where the $w^i$ no longer is the uncommitted node. This means that on each iteration process $k$ would be losing $p$ patterns to the next processes in the pipeline.

Patterns lost to the next processes in the pipeline in a worst case scenario will travel all the way to the last process in the pipeline and afterwards find their way back to process $k$. If this is the situation then process $k$ will have to wait $n-1-k$ units of time, for the patterns sent, to reach process $n-1$ and then wait another $n-1-k$ iterations for the patterns to come back. This is a total of $2(n-1-k)$ iterations before a packet of $p$ templates sent downstream by process $k$ is seen again by process $k$, which implies that $2(n-1-k) - 1$ iterations will pass without seeing the patterns that it sent downstream. If during these $2(n-1-k) - 1$ iterations process $k$ has the bad luck of sending $p$ templates of it’s own templates downstream on each iteration then during that time process $k$ would have lost $p(2(n-1-k) - 1)$ templates and would posses a total of $myShare - p(2(n-1-k) - 1)$ templates. □

Theorem 6.6.10. Overflow avoidance

The first process in the pipeline will always be able to absorb the templates that have been sent to it from the next process downstream.

Proof. Let us assume by contradiction that it cannot absorb the templates it has been sent from the next process downstream. This means that process 0 complies with condition [6.6] and that is has to send templates back. By theorem [6.6.8] the number of templates $N$ that existed in the system $n-1$ iterations ago complies with equation [6.9]. But by line 35 of PROCESS we have:
\[ N > n \times myShare = n \left\lceil \frac{\text{nodes}}{n} \right\rceil \geq n \left( \frac{\text{nodes}}{n} \right) = \text{nodes} \quad (6.11) \]

This means that the number \( N \) of templates that existed in the system \( n - 1 \) iterations ago is greater than \( \text{nodes} \): the number of templates that process 0 is aware of, which implies that there are templates that existed since \( n - 1 \) iterations ago which process 0 is not aware of, and this is a contradiction of theorem 6.6.2. \qed

**Theorem 6.6.11. Pipeline depth invariance**

The difference in the number of myShare that 2 arbitrary processes in the pipeline have cannot exceed \( p + 1 \) where \( p \) is the packet size (the difference in number of templates is independent of the pipeline size \( n \)).

*Proof.* First, by theorem 6.6.2 we know that a process \( k \) is aware of the number of templates that existed \( n - 1 - k \) iterations ago. Also, the largest difference in the number of templates that two process are aware of is found in the difference between process 0 and process \( n - 1 \). Now, lets assume that process 0 is aware of \( \text{nodes}_0 \) templates. Since this amount of templates existed \( n - 1 \) iterations ago and we can create a maximum of \( p \) templates per iteration then the maximum number of templates that process \( n - 1 \) can be aware of is \( \text{nodes}_0 + (n - 1)p \). This means that the value of myShare for process 0 is

\[ myShare_0 = \left\lceil \frac{\text{nodes}_0}{n} \right\rceil \geq \frac{\text{nodes}_0}{n} \quad (6.12) \]

and the value of myShare for process \( n - 1 \) is at the most

\[ myShare_{n-1} = \left\lfloor \frac{\text{nodes}_0 + (n - 1)p}{n} \right\rfloor \leq \frac{\text{nodes}_0 + (n - 1)p}{n} + 1 \quad (6.13) \]
We also know that the number of templates that each processor \( k \) owns is less than or equal to \( myTemplates_k \) so the maximum amount of templates between 2 processors is less than or equal to

\[
myShare_{n-1} - myShare_0 = \left\lfloor \frac{nodes_0 + (n-1)p}{n} \right\rfloor - \left\lfloor \frac{nodes_0}{n} \right\rfloor \leq \frac{nodes_0 + (n-1)p}{n} + 1 - \frac{nodes_0}{n} = \frac{(n-1)p}{n} + 1 \leq p + 1
\]

\[\square\]

**Theorem 6.6.12. Partial evaluation correctness**

*If we make the packet size \( p \) of **PROCESS** equal to the size of the training set and set the number of processes to \( n = 1 \), then the parallel algorithm presented here is equivalent to Anagnostopoulos no Matchtracking FAM.*

**Proof.** Let us start by noting that if the number of process is \( n = 1 \) then the functions **Recv-Next** and **Send-Prev** do not perform any computation and can be omitted. This implies that the variables exchanged in these processes also do not hold any information and can be eliminated too. These variables are the set of templates \( \{w^{i}_{k+1}\} \) coming from the next process in the pipeline and the set of variables \( \{w^{i}_{lo(k-1)}\} \) going to the previous process in the pipeline. By eliminating these lines of code and doing partial evaluation and eliminating unnecessary variables we get the code of figure 6.10

Notice that the only differences with Anagnostopoulos no matchtracking FAM are that 1) the set of patterns doesn’t come as a parameter 2) We are using the function **FIND-WINNER** to find the winner node and 3) templates are being extracted and
Process($\rho_a, \beta_a$)

1. $myTemplates \leftarrow \emptyset$
2. $\forall i=1^p I_i \leftarrow \text{none}$
3. $newNodes \leftarrow 0$
4. $continue \leftarrow \text{TRUE}$
5. $\text{Recv-Prev}(k, \{(w^i_{k-1}, I_{k-1}^i, T_{k-1}^i) : i = 1, 2, \ldots, p\})$
6. $newNodes \leftarrow 0$
7. for each $i$ in $\{1, 2, \ldots, p\}$
8. do $\text{Find-Winner}(I^i, w^i, T^i, \rho_a, \beta_a, myTemplates)$
9. if $\text{class}(I^i) = \text{class}(w^i)$
10. then
11. $myTemplates.\text{Add}([I^i \land w^i])$
12. else
13. $newTemplate \leftarrow I^i$
14. $\text{index}(newTemplate) \leftarrow newNodes$
15. $myTemplates.\text{Add}([I^i, w^i])$
16. $newNodes \leftarrow newNodes + 1$
17.

Figure 6.10: Partial evaluation of parallel no matchtracking FAM using $p = 1$
reinserted in the template set. To guarantee that the first templates created receive priority over newer templates we number the templates when created with a sequential index and use the index to determine who wins competition in case of a tie between templates.

Theorem 6.6.13. Workload balance variance bound

In a pipeline with an arbitrary number of processors and a downstream packet size $p$, the standard deviation of the number of templates that each processor owns cannot exceed

$$\frac{p}{2\sqrt{3}}$$  \hspace{1cm} (6.14)

Proof. Given that in the parallel FAM algorithm there are many templates in transit we cannot know exactly how many templates each process possesses. We can though, approximate a worst case workload balance scenario if we assume, as will usually be the case, that the number of comparisons that a given process does on each iteration will be proportional to the number of templates that it is allowed to posses or $\text{myShare}$. In a worst case scenario, on every iteration the network will be creating $p$ new templates so process $k$ will have a value of

$$\text{nodes}_k = \text{nodes}_0 + kp$$

The expected value of $\text{myShare}$ for this worst case scenario will be

$$\text{Avg}(\text{myShare}) = \frac{\sum_{k=0}^{n-1} \text{nodes}_0 + kp}{n} =$$

$$\frac{\text{nodes}_0 + \frac{p}{n} \sum_{k=0}^{n-1} k}{n} =$$

150
\[ \frac{\text{nodes}_0 + \frac{p}{2}(n - 1)}{n} = \]

\[ \frac{\text{nodes}_0}{n} + \frac{p}{2n}(n - 1) \]

and the variance will be

\[ \frac{1}{n} \sum_{k=0}^{n-1} \left( \frac{\text{nodes}_0 + kp}{n} - \frac{\text{nodes}_0}{n} - \frac{p}{2n} (n - 1) \right)^2 = \]

\[ \frac{1}{n} \sum_{k=0}^{n-1} \left( \frac{kp}{n} - \frac{p}{2n} (n - 1) \right)^2 = \]

\[ \frac{1}{n} \sum_{k=0}^{n-1} \left( \frac{kp}{n} \right)^2 - \left( \frac{p(n - 1)}{2n} \right)^2 = \]

\[ \frac{p^2}{n^3} \sum_{k=0}^{n-1} k^2 - \left( \frac{p(n - 1)}{2n} \right)^2 = \]

\[ p^7 \frac{n(n - 1)(2n - 1)}{6n^3} - p^2 \frac{(n - 1)^2}{4n^2} = \]

\[ p^2 \frac{(n - 1)(2n - 1)}{6n^2} - p^2 \frac{(n - 1)^2}{4n^2} = \]

\[ 2p^2 \frac{(n - 1)(2n - 1)}{12n^2} - 3p^2 \frac{(n - 1)^2}{12n^2} = \]

\[ p^2(n - 1) \left( \frac{2(n - 1)}{12n^2} - \frac{3(n - 1)}{12n^2} \right) = \]

\[ p^2(n - 1) \left( \frac{4n - 2}{12n^2} - \frac{3n - 3}{12n^2} \right) = \]

\[ p^2(n - 1) \left( \frac{4n - 2 - 3n + 3}{12n^2} \right) = \]

151
\[ p^2(n - 1) \left( \frac{n + 1}{12n^2} \right) = \]
\[ \frac{p^2n^2 - 1}{12n^2} \]

and finally this gives us a standard deviation of

\[ \sqrt{\frac{p^2n^2 - 1}{12n^2}} = \frac{p}{2\sqrt{3}} \sqrt{1 - n^{-2}} < \frac{p}{2\sqrt{3}} \]

(6.15)

\[ \square \]

If, for example, we use a packet size of 64 patterns, then the worst possible standard deviation in the value of \textit{myShare} would not exceed

\[ \frac{64}{2\sqrt{3}} = \frac{32}{\sqrt{3}} = 18.4752 \]

\textit{regardless} of the pipeline size \( n \).

### 6.7 Matchtracking Parallel FAM implementation

All the previous discussion assumes that we are implementing Anagnostopoulos no-matchtracking FAM variant. We can with little modification implement a matchtracking FAM ring implementation. To do so we have to make the following assumptions.

1. When the last process \( n - 1 \) in the pipeline sends patterns to the next process, they will be received by the first process 0.
2. When the first process 0 in the pipeline receives patterns from the previous process, they will be received from the last process \( n - 1 \), if less than \( p \) patterns are reviewed from process \( n - 1 \) then training patterns are read from the input source to fill up the packet.

3. All input patterns carry with them the current vigilance criterion.

Notice that none of the behavior concerning the movement of templates is different in the matchtracking FAM compared to the no-matchtracking FAM. This means that all the theorems concerning the properties of Anagnostopoulos no-matchtracking FAM are valid here, as well, are the conclusions we reached concerning the workload balance.

6.8 Experiments

Experiments for all Fuzzy-ARTMAP algorithm variants were conducted on the Forest Covertype database. Training set sizes of \( 1000 \times 2^i, i \in \{0, 1, \ldots, 9\} \), that is 1,000 to 512,000 patterns were used for the training of no matchtracking Fuzzy-ARTMAP and pipelined no matchtracking Fuzzy-ARTMAP. The test set size was fixed at 20,000 patterns. The number of processors in the pipeline varied from \( p = 1 \) to \( p = 32 \). Pipeline sizes were also increased in powers of 2.

To avoid additional computational complexities in the the experiments (beyond the one that the size of the training set brings along) the values of the ART network parameters \( \tilde{p}_a \), and \( \beta_a \) were fixed (i.e., the values chosen were ones that gave reasonable results for the database of focus). For each database and for every combination
Process($k, n, \rho, \beta, p$)

1. Init($p$)
2. While continue do while $|my\text{Templates}| > my\text{Share}$ do
3. Extract-Template($my\text{Templates}, \{w^i_{to(k-1)}\}$)
4. Send-Next($k, n, \{(w^i, I^i, T^i) : i = 1, 2, \ldots, p\}$)
5. Recv-Next($k, n, \{w^i_{k+1} : i = 1, 2, \ldots, 2p\}, newNodes_{k+1}$)
6. Send-Prev($k, \{w^i_{to(k-1)} : i = 1, 2, \ldots, 2p\}, newNodes$)
7. Recv-Prev($k, \{w^i_{k-1, I^i_{k-1}, T^i_{k-1}} : i = 1, 2, \ldots, p\}$)
8. newNodes ← newNodes$_{k+1}$
9. $S ← \{w^i_{k+1}\}$
10. For each $i$ in $\{1, 2, \ldots, p\}$
11. do Find-Winner($I^i, w^i, T^i, \rho, \beta, S$)
12. myTemplates.Add-Templates($S$)
13. continue ← $I^i_{k-1} = \text{EOF}$
14. if continue then $S ← \{w^i_{to(k-1)}\}$
15. For each $i$ in $\{1, 2, \ldots, p\}$
16. do Find-Winner($I^i, w^i, T^i, \rho, \beta, my\text{Templates}$)
17. $(I^i, w^i, T^i) ← (I^i_{k-1}, w^i_{k-1}, T^i_{k-1})$
18. For each $i$ in $\{1, 2, \ldots, p\}$
19. do Find-Winner($I^i, w^i, T^i, \rho, \beta, my\text{Templates}$)
20. if $k = n - 1$
21. then if class($I^i$) = class($w^i$)
22. then myTemplates.Add($\{I^i \wedge w^i\}$)
23. else if $w^i = w_0$
24. then index(newTemplate) ← newNodes + nodes
25. myTemplates.Add($\{I^i\}$)
26. newNodes ← newNodes$_{k+1}$
27. else myTemplates.Add($w^i$)
28. $\rho(I^i) ← \rho(I^i, w^i) + \varepsilon$
29. send $I^i$ to the first processor in the pipeline
30. if newNodes > 0
31. then myShare ← $[\text{nodes} - \text{nodes} + \text{newNodes}] / n$
32. Send-Next($k, n, \{(\text{none}, \text{none}, 0)\}$)
33. Recv-Next($k, n, \{w^i_{k+1} : i = 1, 2, \ldots, 2p\}, newNodes_{k+1}$)
34. myTemplates.Add-Templates($\{w^i_{k+1} : i = 1, 2, \ldots, 2p\}$)

Figure 6.11: Parallel Matchtracking pipelined FAM pseudocode

154
\textsc{Find-Winner}(I, w, T, \rho_a, \beta_a, S = \{w^i\})

\begin{verbatim}
1  idx \leftarrow -1
2  for each \(w^i\) in \(S\)
3    do if \([\rho(I, w^i) \geq \rho(I)]\)
4      then
5        if \([T(I, w^i, \beta_a) > T]\)
6          then
7            \(T \leftarrow T(I, w^i, \beta_a)\)
8            \(idx \leftarrow i\)
9        else if \([T(I, w^i, \beta_a) = T]\) and \(\text{index}(w^i) < \text{index}(w)\)
10           then \(T \leftarrow T(I, w^i, \beta_a)\)
11           \(idx \leftarrow i\)
12    if \(idx \neq -1\)
13      then
14        \text{Extract}(w^{idx}, S)
15        \text{Add}(w, S)
16        w \leftarrow w^{idx}
17        return TRUE
18    else
19      return FALSE
\end{verbatim}

Figure 6.12: Utility function
of \((p, PT) = (\text{partition, training set size})\) values we conducted 12 independent experiments (training and performance phases), corresponding to different orders of pattern presentations within the training set. As a reminder, Fuzzy–ARTMAP performance depends on the values of the network parameters \(\rho_a, \text{ and } \beta_a\), as well as the order of pattern presentation within the training set.

All the tests were conducted on the SCEROLA Beowulf cluster of workstations [Mic03] of the Institute for Simulation and Training. This cluster consists of 64 900MHz machines running with 250MBytes of RAM each. This configuration guaranteed identical conditions on all runs, parallel and sequential. Since there is no communication between processors the sequential time can be calculated as the addition of all the partition times, the parallel time is the maximum of all the partition times.

The metrics used to measure the performance of the pipelined approach were:

1. Classification performance of pipelined no matchtracking FAM.

2. Speedup of pipelined no matchtracking FAM versus FAM.

To calculate the speedup we used 2 measures, the first one is total CPU time of each test and the second one is the total number of iterations of the FAM main loop. This approach allowed us to check how closely the wall clock speedup values are correlated with the number of computations performed by the algorithm. A discrepancy between these measures would mean that we might be obtaining speedup from other implementation dependent sources (like cache non-linearities or operating system dependent issues).
6.8.1 Forest Covertype Database

The Forest Covertype database was provided by Blackard [Bla99], and donated to the UCI Machine Learning Repository [Uni03]. The database consists of a total of 581,012 patterns each one associated with 1 of 7 different forest tree cover types. The number of attributes of each pattern is 54, but this number is misleading since attributes 11 to 14 are actually a binary tabulation of the attribute Wilderness-Area, and attributes 15 to 54 (40 of them) are a binary tabulation of the attribute Soil-Type. The original database values are not normalized to fit in the unit hypercube. Thus, we transformed the data to achieve this. There are no omitted values in the data.

![Figure 6.13: A random sample of Forest Covertype data-points. 5,000 points out of the available 581,012 are shown. The data-points are projected to the first 3 dimensions of the database. Different Colors for the data-points represent different class labels.](image)

Patterns 1 through 512,000 were used for training. The test set for all trials were patterns 561,001 to 581,000. A visualization of the first 3 dimensions of the Forest Covertype database can be seen in figure 6.13. Different tones correspond to different
Table 6.3: Training time in seconds of Forest Covertype database

<table>
<thead>
<tr>
<th>Pipeline Size</th>
<th>32,000</th>
<th>64,000</th>
<th>128,000</th>
<th>256,000</th>
<th>512,000</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>9.277</td>
<td>28.780</td>
<td>96.095</td>
<td>357.011</td>
<td>1367.924</td>
</tr>
<tr>
<td>2</td>
<td>4.767</td>
<td>14.365</td>
<td>49.684</td>
<td>155.427</td>
<td>552.710</td>
</tr>
<tr>
<td>4</td>
<td>2.492</td>
<td>7.878</td>
<td>22.110</td>
<td>81.561</td>
<td>273.200</td>
</tr>
<tr>
<td>8</td>
<td>1.420</td>
<td>4.216</td>
<td>11.032</td>
<td>51.534</td>
<td>140.210</td>
</tr>
<tr>
<td>16</td>
<td>0.885</td>
<td>2.030</td>
<td>7.552</td>
<td>23.376</td>
<td>76.473</td>
</tr>
<tr>
<td>32</td>
<td>0.863</td>
<td>1.604</td>
<td>4.515</td>
<td>14.647</td>
<td>38.319</td>
</tr>
</tbody>
</table>

Table 6.4: Classification performance of no-matchtracking Fuzzy–ARTMAP

<table>
<thead>
<tr>
<th>Number of Patterns</th>
<th>Classification %</th>
</tr>
</thead>
<tbody>
<tr>
<td>32,000</td>
<td>68.97</td>
</tr>
<tr>
<td>64,000</td>
<td>72.87</td>
</tr>
<tr>
<td>128,000</td>
<td>70.66</td>
</tr>
<tr>
<td>256,000</td>
<td>73.28</td>
</tr>
<tr>
<td>512,000</td>
<td>75.23</td>
</tr>
</tbody>
</table>

classes. As it can be seen from the figure the class boundaries are quite complex. Classification performance of different machine learning algorithms for this database has been reported in the range of 75%.

Results for the training time in seconds for this database can be seen in table 6.3. The speedup for the same data is shown in figure 6.14. We can see that the speedup is close to linear. For large training set sizes the speedup becomes slightly supra-linear. This behavior is to be expected since in large database sizes the sequential version of the algorithm will probably not fit in processor cache. The parallel version of the algorithm on the other hand reduces the amount of memory each processor consumes and therefore it is likely that each subtask will fit in cache.
Classification performance is shown on table 6.8.1. Here we can see that classification of the no-matchtracking FAM variant is comparable to the unmodified FAM algorithm.
Figure 6.14: Forest CoverType database speedup in seconds.
6.9 conclusions

We present a pipelined Fuzzy-ARTMAP variant that eliminates matchtracking. We demonstrate experimentally that it’s classification performance is comparable with Fuzzy-ARTMAP. We also show formally that the algorithm is well behaved and has good workload balancing properties. The algorithm exhibited linear speedup when the number of processors was increased. No parallel Fuzzy-ARTMAP implementation was found in the literature. To the best of our knowledge, this is the first implementation of a Fuzzy-ARTMAP like classification algorithm on a parallel machine.
CHAPTER 7
CONCLUSIONS

We observed that FAM’s training time tends to slow considerably when FAM is applied to large classification problems. To attack this problem we developed equation 2.46 to characterize FAM’s online sequential time complexity, and equation 2.48 to characterize FAM’s offline sequential time complexity. Using some reasonable simplifying assumptions we develop in equation 2.53 a closed form formula for the maximum number of epochs for achieving FAM convergence and in equation 2.54 a close form formula for the number of templates that a network creates. This formal characterization of the algorithm is used to conjecture that data partitioning can speedup the algorithm quadratically in the number of partitions and that network partitioning can speedup the algorithm linearly in the number of processors. More specifically, partitioning the data set we would theoretically be able to reach a speedup of $p$ in the sequential machine and $p^2$ in an efficient parallel machine with sufficient number of processors.

We proposed the use of Hilbert space–filling curves to enforce some sort of data partitioning (the resulting scheme was called hFAM). This approach has not been found in the literature in the context of neural network parallelization or work reduction. Experimental results of hFAM on 3 databases confirmed our expectations: the classification performance of hFAM is not affected, as compared to the classification performance of FAM. The size of the resulting hFAM is slightly higher than the cor-
responding FAM structure. Finally, and most importantly, the convergence time of hFAM is improved linearly on the sequential machine and quadratically on the parallel machine just as our FAM complexity characterization predicted. Convergence times on a database of 512,000 patterns was reduced from the order of days to train to the order of minutes even when we did not use a parallel machine.

We also proposed the use of a processor pipeline to achieve parallelization and network partitioning. An algorithm is developed for this pipeline that guarantees consistency and equivalence to the sequential algorithm no-matchtracking Anagnostopoulos’s FAM variant. We demonstrate formally that workload balance is guaranteed if the transmission packet size is small compared to the number of templates in the network. Experimental results on one database show that the algorithm is close to optimal since it achieves very close to linear speedup on the pipeline size and identical classification results as the sequential version.

Nevertheless, there are lots of pending problems to be solved related to this work. Analysis of the workload balance in the parallel machine for the data partitioning hFAM indicated that not all processes are being utilized to the maximum level. This is because in hFAM even though the number of patterns processed by each partition is the same, the number of templates varies considerably from one processor to another depending on the complexity of the region it has to classify.

We foresee, and are currently pursuing, different avenues of research. Combining our data-partitioning approach with a networking partitioning approach will help us achieve optimal workload balance in the parallel implementation of the data partitioning hFAM algorithm. Dynamically configuring the processor pipeline can also improve the speedup of the pipelined FAM and eliminate a parameter in the algo-
rithm. This has already been theoretically analyzed in appendix B but requires the use of MPI2 to permit dynamic process allocation in the cluster.

The properties of the pipelined network partitioning approach developed here are general enough to be applied to other types of competitive classifiers like the closely related ARTMAP, Fuzzy Min–MAX, Kohonen, and even Bayesian classifiers. Extending our results to these other classifiers is part of the work we intend to develop.

Also, even though the pipeline provided good results, it is of interest to us to develop other parallelization schemes of the neural network. Specifically we are interested in implementing a tree structure parallel neural network. We have unpublished results that prove formally that on batch database processing the improvement of the tree structure approach over the pipeline is negligible, but in online processing it is reasonable to expect that the response time will be reduced from $p$ to $\log(p)$ where $p$ is the number of processors.

Another venue of research lies in the area of dimensionality reduction. Analysis of the fractal dimension of the databases used shows that their fractal dimension is usually much lower that the actual dimension of the database. We conjecture that using the Hilbert space–filling curve to reduce the dimensionality of the database to it’s fractal dimension will produce very similar classification results as classifying on the full dimensionality of the data set.

In the area of applications we have found in the literature dimensionality reduction techniques applied with self organizing maps for the analysis of gene microarray data [HYA03]. The fact that the Hilbert space–filling curve can map an $n$-dimensional point to a 1-dimensional line preserving properties of the distance function makes us think that it can be useful for problem representation in genetic algorithms. We are currently
researching the use of Hilbert space-filling curves for the mapping of 2-3 dimensional problems into a genetic algorithm 1-dimensional structure.
APPENDIX A
PARAMETER ESTIMATION
The optimum number of processing elements in the chain will depend on the size of the packet (number of templates) sent across the pipeline, to derive a formula for this number lets define:

- $p$: number of patterns sent on a packet between processors in the pipeline, this number is assumed fixed since there is an optimum packet size that minimizes transmission time.

- $XMT(p)$: time it takes to transmit and exchange a packet of $p$ $(I, w)$ pairs across 2 processes in the pipeline.

- $c$: time it takes to compare a pattern $I$ against a template $w$ to check if it passes vigilance ($\rho(I, w) \geq \rho_a$) and if it maximizes activation $T(I, w, \beta_a)$.

- $\Pi$: total number of templates.

- $n$: number of processors in the pipeline.

- $P$: number of training patterns in the training set.

Then the time it takes to compare a single pattern against all the templates is:

$$\frac{c\Pi}{n}$$  \hspace{1cm} (A.1)

and the time it takes to transmit all the patterns through the pipeline is

$$XMT(p) \left( \left\lceil \frac{P}{p} \right\rceil + n - 1 \right)$$  \hspace{1cm} (A.2)
this transmission cost means that by pattern the total time becomes

\[
\frac{1}{P} XMT(p) \left( \left\lceil \frac{P}{p} \right\rceil + n - 1 \right) \sim XMT(p) \left( \frac{1}{p} + \frac{n-1}{P} \right) \tag{A.3}
\]

and the total processing cost including transmission for each pattern in the database would be

\[
\frac{c \Pi}{n} + XMT(p) \left( \frac{1}{p} + \frac{n-1}{P} \right) \tag{A.4}
\]

Tests where conducted on the cluster to review the feasibility of the previous formula, results indicate that it is best to synchronize the processors so that they all communicate at the same time.

From equation [A.4] we can get the optimum amount of processors given an amount of templates and a packet size by:

\[
\frac{d \left( \frac{c \Pi}{x} + XMT(p) \left( \frac{1}{p} + \frac{x-1}{P} \right) \right)}{dx} = 0 \Rightarrow
\]

\[
\frac{d \left( \frac{c \Pi}{x} + \frac{XMT(p)}{p} + XMT(p) \frac{x-1}{P} \right)}{dx} = 0 \Rightarrow
\]

\[
\frac{d \left( \frac{c \Pi}{x} \right)}{dx} + \frac{d \left( XMT(p) \frac{1}{x} \right)}{dx} + \frac{d \left( XMT(p) \frac{x-1}{P} \right)}{dx} = 0 \Rightarrow
\]

\[
c \Pi \frac{d \left( \frac{1}{x} \right)}{dx} + XMT(p) \frac{d (1)}{dx} + XMT(p) \frac{d (x-1)}{P} = 0 \Rightarrow
\]

\[- \frac{c \Pi}{x^2} + \frac{XMT(p)}{P} = 0 \Rightarrow
\]

\[
\frac{XMT(p)}{P} = \frac{c \Pi}{x^2} \Rightarrow
\]

168
It is noticeable that the assumption of linearity of the transmission time with respect to the number of processors in the pipeline is consistent with the experimental evidence and assuming the existence of some randomness in the transmission times.

using complement coding would restrict the amount of patterns that can be transmitted but would standardize the notation. Also we need to take into consideration that we have to be able to transmit more templates backwards than in the pipeline chain than forward. Nevertheless since packets that are sent forward consist of pattern, template \((I, w)\) pairs and packets sent back consist only of templates then with equal sized packets in both directions we achieve our objective.

The following rationale was applied to the tested databases:

- Send forward a total of \(p = 32\) patterns at a time with their corresponding 32 templates.
- Send backward on the chain a total of \(2p = 64\) templates, this takes up the space of the 128 possible patterns.
- Synchronize and communicate amongst the processors at the start of each loop.
APPENDIX B
SEQUENTIAL MODIFICATIONS
The modifications presented on this chapter are based on equation 2.47. We will develop methods in the chapter on data partitioning techniques that will also speedup FAM under sequential execution, but since the main principle behind them is a data partitioning approach that can also be used on a parallel setting we will postpone their presentation until chapter 5.

If we observe equation 2.47 and we assume, as is usually the case, that the size $P$ of the dataset is given and that the amount of templates to be created is also given, then the parameters from that equation that we can work with are $\Gamma$, and $epochs$. The fact that FAM time is linear with respect to these parameters indicates that by modifying the section of the algorithm that deals with their computations we should be able to improve its performance linearly.

The $\Gamma$ parameter concerns the matchtracking, so to reduce $\Gamma$ we need to reduce the amount or type of matchtracking that the algorithm is incurring in.

$epochs$ represents the number of iterations, so in order to reduce the parameter $epochs$ we can opt for either of 2 things:

1. Only do 1 epoch of the algorithm or

2. Reduce the time spent on subsequent epochs.

These approaches intend to make the time it takes for multi-epoch FAM to execute comparable to that of 1 epoch FAM.

The following proposals intend to address these time problems, under the assumption that the matchtracking parameter $\Gamma$ is significant and/or the number of epochs are relevant for the quality of the classification of FAM. Of course, these assumptions can only be verified by extensive experimentation.
Expensive--FAM

This Fuzzy-ARTMAP variant tries to reduce the cost associated with multiple epoch FAM. The rationale for its modifications lies on the observation that after the first epoch of FAM training very few templates misclassify or need to be re-learned. If we can efficiently pinpoint which are the templates that need to be reviewed and re-learned then the time complexity of subsequent FAM epochs can be substantially reduced.

To analyze this let us consider what happens when a pattern \( I' \) is presented to the network. When we do this, only three things can happen:

1. The pattern \( I' \) might be classified correctly on to the appropriate template \( w_i \) without any modification to the template (\( I' \) already lies inside the box encompassed by \( w_i \))

2. The pattern \( I' \) selects a template \( w_i \) that has to be expanded to include \( I' \). By theorem 2.3.6 this implies that \(|w_i|\) becomes smaller so the \( T_j \)'s of all the templates that classify to \( w_i \) will be smaller. This might produce a process of *bleeding* where the template loses patterns that had previously classified to it. These patterns will need re-learning.

3. The pattern \( I' \) selects the un-committed node, this will create a new point template \( w = I' \) that being new might *steal* patterns from neighboring templates.

We can create an algorithm, that although expensive computationally, is smarter that FAM, this algorithm will check what happens with the templates every time a pattern is presented for learning and label those patterns that need to be presented again for the next epoch, for this purpose we will call a pattern \( I' \) *clean* if we can
guarantee that it has been learned correctly, and conversely it will be called dirty if we cannot guarantee that it classified to the correct category. This version of the algorithm we will call Expensive–Fuzzy–ARTMAP or Expensive–FAM for short is depicted in figure B.4.

This version of the algorithm has the advantage of reducing the number of patterns that we retrain with since some patterns will not be flagged as dirty on subsequent epochs of the algorithm. The problem is that this comes at the cost of higher computational complexity: every time that a new template is created it draws the attention of all the patterns and reviews them for vigilance. Also every time a category is expanded all the patterns that previously classified to it will be re–checked.

*Heuristic–FAM*
the excessive computational complexity of the previous FAM variant is due to the rechecking of all the patterns every time a new template has been created. We can use the following heuristic to minimize the amount of patterns that need to be reclassified. On Expensive–FAM we flag the pattern as dirty when it’s template changes, this behavior depends on certain patterns that are expanding the templates or creating new ones, what if instead of flagging as dirty all the patterns of the template we flag as dirty the patterns that provoked modification either by template expansion or changing template. We will call this version of the algorithm Heuristic–FAM s because this technique is not guaranteed to produce the same results as the FAM or Expensive–FAM and might overlook the reclassification of certain patterns, but it is also likely that the patterns that are producing template expansion and
Figure B.3: Case 3 — The pattern creates a new template.

*the new template competes with neighboring templates for the patterns and may absorb patterns from other templates (stealing).*

changing template are the ones that lie in borderline areas that need reclassification.

The *Heuristic*–FAM algorithms pseudocode is shown in figure [B.5](#).

As stated earlier, this version of the algorithm flags as dirty a lot less patterns from the training set but has the disadvantage that we can no longer guarantee 100% convergence on the training set. It’s appeal would lie in situations where multiple FAM convergence produces better results than single iteration FAM, and where the *Heuristic*–FAM’s results would be close to the FAM multiple epoch classification ratio but without the multiple epoch FAM time complexity.

*Queue*–FAM

Looking back at the complexity formula given for the algorithm we can see that one of the parameters is the average number of matchtracking per pattern. It is rea-
EXPENSIVE-FAM-LEARNING-PHASE(\{I^1, O^1\}, \ldots, (I^P, O^P)\), \rho_a, \beta_a, epochs, \varepsilon)

1 templates ← \{\} ; iterations ← 0
2 for each I^r in \{I^1, \ldots, I^P\}
3 do I^r.dirty ← TRUE ; I^r.template ← NONE
4 repeat
5      modified ← FALSE
6      for each (I^r, O^r) in \{(I^1, O^1), (I^2, O^2), \ldots, (I^P, O^P)\}
7      do if I^r.dirty
8        then \rho_a ← \bar{\rho}_a
9        I^r.dirty ← FALSE
10       repeat
11          choice ← \frac{M_a}{M_a + \beta_a} ; status ← NoneFound
12          for each w_j in templates
13            do if (\rho(I^r, w_j) \geq \rho_a) and (T(I^r, w_j, \beta_a) > choice)
14                                    then choice ← T(I^r, w_j, \beta_a) ; \ j_0 ← j
15                                    status ← GotOne
16            if status = GotOne
17               then if O^r = w_{j_0}.class
18                          then status ← ThisIsIT
19                         else status ← Matchtracking
20                          \rho_a ← \rho(I^r, w_j) + \varepsilon
21         until status \neq Matchtracking
22         if status = Allocated
23            then if I^r.template \neq w_{j_0}
24            then modified ← TRUE
25            if w_{j_0} \neq (w_{j_0} \land I^r)
26             then w_{j_0} ← (w_{j_0} \land I^r)
27             for each I_k \neq I^r in \{I^1, \ldots, I^P\}
28          do if I_k.template = w_{j_0}
29             then I_k.dirty ← TRUE
30          else w_k ← I^r
31          templates ← templates \cup \{w_k\}
32          modified ← TRUE ; I^r.template ← w_k
33          for each I_j \neq I^r in \{I^1, \ldots, I^P\}
34          do if \left(\frac{|I_j|}{M_a} \geq \bar{\rho}_a\right)
35             then I_j.dirty ← TRUE
36 until (++iterations = epochs) or (modified = FALSE)
37 return templates

Figure B.4: Expensive-FAM learning phase
Heuristic-FAM-Learning-Phase(\{ (I^1, O^1), \ldots, (I^P, O^P) \} , \bar{\rho}_a, \beta_a, epochs, \varepsilon)

1. templates ← \{\}; iterations ← 0
2. for each \( \Gamma' \) in \{\( I^1 \), \ldots, \( I^P \)\}
3. do \( \Gamma'.dirty ← \text{TRUE} \quad \Gamma'.template ← \text{NONE} \)
4. repeat
5. modified ← \text{FALSE}
6. for each \((\Gamma', O^r)\) in \{\( (I^1, O^1) \), \( (I^2, O^2) \), \ldots, \( (I^P, O^P) \)\}
7. do
8. if \( \Gamma'.dirty \)
9. then \( \rho_a ← \bar{\rho}_a \)
10. repeat
11. choice ← \( \frac{M_a}{2M_a + \beta_a} \); status ← NoneFound
12. for each \( w_j \) in templates
13. do if \( (\rho(\Gamma', w_j) \geq \rho_a) \) \text{ and } \( T(\Gamma', w_j, \beta_a) > \text{choice} \)
14. then choice ← \( T(\Gamma', w_j, \beta_a) \)
15. \( j_0 ← j \); status ← GotOne
16. if status = GotOne
17. then if \( O^r = w_{j_0}.\text{class} \)
18. then status ← Allocated
19. else status ← Matchtracking
20. \( \rho_a ← \rho(\Gamma', w_j) + \varepsilon \)
21. until status ≠ Matchtracking
22. if status = Allocated
23. then if \( O^r = w_{j_0} \)
24. then \( \Gamma'.dirty ← \text{FALSE} \)
25. else modified ← \text{TRUE} \quad \Gamma'.dirty ← \text{TRUE} \quad w_{j_0} ← (w_{j_0} \land \Gamma')
26. else \( w_k ← \Gamma' \)
27. templates ← templates \cup \{w_k\} \quad \Gamma'.template ← w_k \quad modified ← \text{TRUE} \quad \Gamma'.dirty ← \text{TRUE}
28. until \((++\text{iterations} = \text{epochs})\) \text{ or } (\text{modified} = \text{FALSE})
29. return templates

Figure B.5: Heuristic–FAM learning phase
sonable to assume that matchtracking can be a factor in slowing down the algorithm, this would happen specially in data sets that classify to many categories (the more categories we have the higher the likelihood that the winning template is not of the correct category and therefore the higher the likelihood of matchtracking) So it is reasonable to point out that trying to bring down the cost of matchtracking might very likely improve performance.

If we analyze the matchtracking loop on the no modifications

This gives us the version of the algorithm that we will call Queue–FAM shown in figure [B.6]

Notice that in this version of the algorithm the template list is transversed only once. On this transversal we create a Queue data structure that holds the templates that pass vigilance ordered by their match criteria \( \rho(w, I) \). On matchtracking we only have to access the queue at the top, and this can be done in constant time or \( O(\lg(templates)) \) time depending on whether we have to restructure the queue or not. In general, the construction of the heap and the extraction of values will take \( O(n \lg(n)) \) time where \( n = \overline{templates} \)

Buffered–FAM

When we try to minimize the matchtracking factor, if the value of \( \rho_a \) is high we can assume that the number of templates that pass vigilance is going to be a small fraction of the total number of templates. This means that for large values of \( \rho_a \) we can reduce the cost of matchtracking considerably by buffering the templates that pass the vigilance criterion. This strategy is the underlying motive of the Buffered–FAM implementation variant that is shown in figure [B.7]
Queue-FAM-Learning-Phase(\{(I^1, O^1), \ldots, (I^P, O^P)\}, \rho_a, \beta_a, \text{epochs}, \varepsilon)

1. templates ← {} 
2. iterations ← 0 
3. w_0 ← \{1,1,\ldots,1\} 

4. repeat
   5. modified ← FALSE
   6. for each (I^r, O^r) in \{(I^1, O^1), (I^2, O^2), \ldots, (I^P, O^P)\} do
      7. \rho_a ← \tilde{\rho}_a
      8. Passed ← {}
      9. QUEUE(Passed, w_0, priority = \frac{M_a}{2M_a + \beta_a})
     10. for each w_j in templates do
          11. do if (\rho(I^r, w_j) ≥ \rho_a) and \left( T(I^r, w_j, \beta_a) > \frac{M}{2M_a + \beta_a} \right) then
               12. QUEUE(Passed, w_j, priority = T(I^r, w_j, \beta_a))
          13. repeat
        14. repeat
           15. w_{j_0} ← POP(Passed)
           16. until (Passed = {}) or (\rho(I^r, w_{j_0}) ≥ \rho_a)
        17. if w_{j_0} = w_0 then
            18. status ← NewTemplate
            19. else if O^r = w_{j_0}.class then
                20. status ← ThisIsIT
            21. else status ← Matchtracking
        22. \rho_a ← \rho(I^r, w_{j_0}) + \varepsilon
     23. until status ≠ Matchtracking
    24. if status = Allocated then
        25. if w_{j_0} ≠ (w_{j_0} \land I^r) then
            26. w_{j_0} ← w_{j_0} \land I^r
            27. modified ← TRUE
        28. else templates ← templates \cup \{I^r\}
        29. modified ← TRUE
    30. iterations ← iterations + 1
   31. until (iterations = \text{epochs}) or (modified = FALSE)

32. return templates

Figure B.6: Queue–FAM learning phase
Buffered-FAM-Training-Phase(\{ (I^1, O^1), (I^2, O^2), \ldots, (I^P, O^P) \} \cup \rho_a, \beta_a, \varepsilon)

1. \textbf{w}_0 \leftarrow \{1, 1, \ldots, 1\}^{2M_a}
2. \text{templates} \leftarrow \{\textbf{w}_0\}
3. \textbf{repeat}
4. \hspace{1em} \text{modified} \leftarrow \text{FALSE}
5. \hspace{1em} \textbf{for each} (\Gamma', O') \textbf{ in } \{ (I^1, O^1), (I^2, O^2), \ldots, (I^P, O^P) \}
6. \hspace{2em} \textbf{do} \hspace{1em} \rho_a \leftarrow \tilde{\rho}_a \hspace{1em} T_{\text{max}} \leftarrow 0
7. \hspace{2em} \text{buffer} \leftarrow \{\}
8. \hspace{2em} \textbf{for each} \textbf{w}_j^a \textbf{ in templates}
9. \hspace{3em} \textbf{do if } \rho(\Gamma', \textbf{w}_j^a) \geq \rho_a
10. \hspace{4em} \text{then buffer} \leftarrow \text{buffer} \cup \{\textbf{w}_j^a\}
11. \hspace{4em} \textbf{if } T(\Gamma', \textbf{w}_j^a, \beta_a) > T_{\text{max}}
12. \hspace{5em} \text{then } T_{\text{max}} \leftarrow T(\Gamma', \textbf{w}_j^a, \beta_a)
13. \hspace{4em} \textbf{w}_{j_{\text{max}}} \leftarrow \textbf{w}_j
14. \hspace{2em} \textbf{repeat}
15. \hspace{3em} \textbf{if } \textbf{w}_{j_{\text{max}}} = \textbf{w}_0 \textbf{ or } O' = \textbf{w}_{j_{\text{max}}}^{\text{class}}
16. \hspace{4em} \text{then } \text{status} \leftarrow \text{Allocated}
17. \hspace{3em} \textbf{else} \hspace{1em} \text{status} \leftarrow \text{TryAgain}
18. \hspace{4em} \rho_a \leftarrow \rho(\Gamma', \textbf{w}_{j_{\text{max}}}^a) + \varepsilon
19. \hspace{4em} T_{\text{max}} \leftarrow 0
20. \hspace{4em} \textbf{for each} \textbf{w}_j^a \textbf{ in buffer}
21. \hspace{5em} \textbf{do if } \rho(\Gamma', \textbf{w}_j^a) < \rho_a
22. \hspace{6em} \text{then buffer.DELETE(\textbf{w}_j^a)}
23. \hspace{5em} \textbf{else if } T(\Gamma', \textbf{w}_j^a, \beta_a) > T_{\text{max}}
24. \hspace{6em} \text{then } T_{\text{max}} \leftarrow T(\Gamma', \textbf{w}_j^a, \beta_a)
25. \hspace{6em} \textbf{w}_{j_{\text{max}}} \leftarrow \textbf{w}_j
26. \hspace{2em} \textbf{until status} \neq \text{TryAgain}
27. \hspace{2em} \textbf{if } \textbf{w}_{j_{\text{max}}} \neq \textbf{w}_0
28. \hspace{3em} \textbf{then if } \textbf{w}_{j_{\text{max}}}^a \neq \textbf{w}_{j_{\text{max}}}^{\text{class}} \wedge \Gamma'
29. \hspace{4em} \text{then } \textbf{w}_{j_{\text{max}}}^a \leftarrow \textbf{w}_{j_{\text{max}}}^{\text{class}} \wedge \Gamma'
30. \hspace{4em} \textbf{modified} \leftarrow \text{TRUE}
31. \hspace{3em} \textbf{else}
32. \hspace{4em} \text{templates} \leftarrow \text{templates} \cup \{\Gamma'\}
33. \hspace{2em} \textbf{until modified} = \text{FALSE}
34. \textbf{return templates}

Figure B.7: Buffered-FAM pseudocode
Even though experimental results shed light on a very low matchtracking level on the overall run of the algorithm.
LIST OF REFERENCES


185


