DESIGN AND ANALYSIS OF A SYSTOLIC ARRAY FOR NEURAL COMPUTATION

THÈSE N° 1264 (1994)

PRÉSENTÉE AU DÉPARTEMENT D'INFORMATIQUE

ÉCOLE POLYTECHNIQUE FÉDÉRALE DE LAUSANNE

POUR L'OBTENTION DU GRADE DE DOCTEUR ÈS SCIENCES

PAR

MARC VIREDAZ
Ingénieur en microtechnique diplômé EPF
originaire de Crissier (VD) et Genève (GE)

acceptée sur proposition du jury:

Prof. J.-D. Nicoud, rapporteur
Dr M. Annaratone, corapporteur
Prof. U. Ramacher, corapporteur
Prof. E. Sanchez, corapporteur

Lausanne, EPFL
1994

à mes parents

Acknowledgements

The first person, I wish to express my gratitude to, is my advisor, Prof. Jean-Daniel Nicoud, for having trusted me and giving me the opportunity to complete this thesis. I would like to acknowledge his scientific and technical support as well as his friendship, both during the months I spent in his laboratory back in 1988 to complete my graduation project and during the last two and a half years.

Of capital importance to a thesis are the members of the examining committee. First, I would like to thank Prof. André Schiper for presiding them. I am very grateful to Dr. Marco Annaratone from DEC, to Prof. Ulrich Ramacher from the TU Dresden (formerly at Siemens), and to Prof. Eduardo Sanchez from the EPFL for having accepted to take part in the committee. I am particularly indebted to Marco Annaratone, with whom I work two and a half years on the K2 project at the ETH-Zurich, and from whom I learned most of what I know on parallel computers.

Special thanks go to Dr. François Blayo and Dr. Christian Lehmann, whose theses are the direct ancestors of mine. Their work has provided the necessary background on which the MANTRA I machine, presented here, is built. I wish to thank them for their friendship and for having introduced me to the fascinating world of artificial neural networks.

A special attention should be dedicated to Paolo Ienne, with whom I shared work and office, as well as most problems and successes, during the last two and a half years. His thorough implementation of the GACD1 and GENES IV integrated circuits and several novel ideas greatly helped the success of the MANTRA I machine. I would like to acknowledge his scientific and technical abilities and to thank him for his friendship and for bringing his Italian good mood in the office.

Next, I desire to express my gratitude all my colleagues who helped me building the hardware or writing software. I'm thankful to Christophe Marguerat, who designed the processor board, for his advise and help with the TMS320C40 microprocessor. I wish to thank the application-software designers, Dr. Thierry Cornu and Giorgio Caset, and recognize their courage to accept programming the machine I have designed. Their work is based on the initial study from Bruno Buggiani. The program loader written by Thierry Conus was also very helpful. The outstanding

work of Georges Vaucher and Peter Brühlmeier during the placement and routing of the printed circuit boards has proved to be a key factor to the success of the project. Finally, André Guignard should be thanked for having spent weeks packaging the VLSI custom circuits.

During the redaction of a thesis, the feed-back of proofreaders is an essential contribution to the final result. The pertinent comments from Dr. Thierry Cornu, Dr. Marco Fillo, Jelena Godjevac, Paolo Ienne, Dr. Christian Lehmann, and Patrick Thiran substantially helped improving the quality of this document.

Reiterated thanks go to my former colleague and very dear friend Marco Fillo for having spent many hours reviewing my thesis, while unfamiliar with the project. I'm grateful for everything I learned about digital electronics, while working with him after my graduation. I want finally to thank him for his friendship, *grazie*. I am also glad to thank my friend Jelena Godjevac, who accepted to read and correct my thesis, although working in a very different domain. For this work and for unforgettable *gibanice*, хвала много.

I wish also to thank all my colleagues from the LAMI and the MANTRA group for forming one of the nicest team in which to work. In particular, I would like to express my gratitude to the steering committee of the MANTRA group, formed by the Professors Alain Germond, Martin Hasler, Jean-Daniel Nicoud, Eric Vittoz, and Dominique de Werra, for having set up an infrastructure well suited for research in the domain of artificial neural networks and in this way having given me the possibility to complete this thesis.

Not to be forgotten are the secretaries, without whom no institute or laboratory could ever exist. Their work, too often neglected, is nevertheless essential, and no time is tougher than when they are absent. I'm very thankful to Marie-José Pellaud from the LAMI and to Monique Dubois from the MANTRA group.

Enfin, I would like to thanks those whom I owe everything, my parents, as much for their love and affection as for having given me the opportunity of choosing the job I like and always encouraged to do so. They can be assured of my eternal gratitude and affection. Finally, I would like to thank all my relatives and friends for their constant moral support, with a special thought for my brother Yves-Daniel and for my friend Sandrine. *Merci!*

Abstract

Research on *artificial neural networks (ANNs)* has been carried out for more than five decades. A renewed interest appeared in the 80's with the finding of powerful models like J. Hopfield's *recurrent networks*, T. Kohonen's *self-organizing feature maps*, and the *back-propagation rule*. At that time, there was no platform that was at the same time versatile enough for any ANN model to be implemented and fast enough to solve large problems. Super-computers were the sole exception to this rule, but were prohibitively expensive for most applications. However, both research scientists and application engineers clearly identified the need for such a computing power. This triggered many projects in the field.

In parallel, research on *multi-processor systems* started during the 60's. *Systolic arrays* have been proposed in 1979 as a means to fully exploit the possibilities of VLSI. Two previous theses, by F. Blayo and C. Lehmann, have studied the use of bi-dimensional systolic arrays for neural computation. At first, the presented system, called *GENES*, has been designed for the Hopfield model. Extensions to other ANNs have also been proposed.

The goal of the present thesis is to study, design, build, and analyze an efficient accelerator for neural computation. In a first step, the GENES architecture has been extended towards generality and efficiency. This includes a thorough analysis of ANN models, of other neural computers, and of previous GENES implementations. The result of this work is the *GENES IV* integrated circuit, whose architecture has been co-designed by P. Ienne and the author. The main part of this thesis discusses the architecture, the design and the analysis of the *MANTRA I* machine, a neural computer based on a GENES IV array with up to 40×40 *processing elements (PEs)*. The delta rule (and hence the Perceptron and Adaline rules), the back-propagation rule, the Hopfield model, and the Kohonen model can be implemented on this system. Although not a generic system, such a machine may be regarded as a *multi-model neural computer*. A prototype has been running for a year and is used daily by software designers.

Several novel features distinguish the MANTRA I machine from other neural systems. First, it belongs to the few existing neural computers, contrary to the majority of implementations, which are specific to an application or to an algorithm. The machine does not hard-wire any algorithm,

but provides the necessary primitives to implement the target models. This is a key feature for research, since several algorithms or versions of an algorithm can be tested on a problem. It is an important aspect for applications as well, because different ANN models are often cascaded to solve a problem.

The GENES IV array—that is, the computing core of the MANTRA I machine—features synapse-level parallelism (i.e., one real or virtual PE is allocated per synapse or neural connection), while most other systems exploit only neuron-level parallelism (i.e., one PE per neuron). Hence, this system aims at a much finer parallelism grain and is well suited for massively parallel architectures.

The problem size that can be computed by a neural accelerator should not be limited by the hardware (except for memory size). Therefore, it is essential to support time-sharing of PEs. On the MANTRA I machine, this is achieved by the concept of *virtual arrays*. Matrices are divided into sub-matrices that can be mapped onto the physical array, which is then time-shared among them. An efficient mechanism has been implemented to swap sub-matrices in background, while some other computation is performed.

Since systolic arrays are pipelined systems, it is important to avoid emptying and re-filling them too often, in order to keep the hardware *utilization rate* high. Therefore, a *systolic instruction flow* has been implemented, so that each instruction follows the data for which it has been issued.

Like any SIMD system, the MANTRA I machine is composed of a *parallel* or *SIMD module* and a *control module*. The SIMD module consists of the GENES IV array and a set of dedicated units designed for the computation that scales with the number of neurons and would poorly fit on a bi-dimensional array. A complex system of FIFOs and memories sustains the required input/output streams for the systolic array.

The control module is a complete SISD system. Its tasks are (1) to control the SIMD module by dispatching instructions, (2) to manage data input and output, (3) to communicate with the external world, and (4) to perform data pre- and post-processing. The SIMD instructions are of the *very long instruction word (VLIW)* type. Synchronization between the two modules is achieved by an instruction FIFO.

The performance of the MANTRA I machine has been analyzed using the delta rule. Measurements show that the sustained performance is very close to its peak value, as long as the problem fits in the memory banks connected to the GENES IV array. Experiments have also been run to investigate the impact of the constraints imposed by the hardware on the convergence of algorithms.

Finally, the use of systolic arrays as neural accelerators is discussed in the light of the experience acquired with the GENES IV array and the MANTRA I machine. The weaknesses of the machine are analyzed, and several solutions are proposed to avoid them in a future design. A general discussion of the future of neural computers concludes this thesis.

Résumé

La recherche dans le domaine des *réseaux de neurones artificiels (RNA)* s'étend sur plus de cinq décennies. Elle a suscité un regain d'intérêt dans les années 80 avec la mise au point de modèles tels que les *réseaux récurrents* de J. Hopfield, les *cartes auto-organisatrices* de T. Kohonen et la *règle de la rétro-propagation du gradient*. A cette époque, il n'existait aucun système à la fois suffisamment souple pour que n'importe quel modèle de RNA puisse être implanté et suffisamment rapide pour résoudre des problèmes importants. Les super-ordinateurs étaient la seule exception à cette règle, mais leur prix était prohibitif pour la plupart des applications. Cependant, tant les chercheurs que les ingénieurs ont clairement identifié le besoin d'une telle puissance de calcul. Ce fut le point de départ de nombreux projets dans ce domaine.

En parallèle, la recherche dans le domaine des *ordinateurs multi-processeurs* a débuté dans les années 60. Les *tableaux systoliques* ont été proposés en 1979 comme moyen d'exploiter au mieux les possibilités du VLSI. Dans deux thèses précédentes, F. Blayo and C. Lehmann ont étudié l'utilisation de tableaux systoliques bi-dimensionnels pour le calcul neuronal. Le système presenté, appelé *GENES*, a d'abord été conçu pour le modèle de Hopfield. Des extensions à d'autre RNA ont également été proposées.

Le but de la présente thèse est d'étudier, de concevoir, de construire et d'analyser un accélérateur de calcul neuronal efficace. Dans un premier temps, l'architecture de GENES a été étendue en vue d'offrir un champ d'application plus vaste et une plus grande efficacité. Cette étape comprend une analyse approfondie des modèles de RNA, d'autres ordinateurs neuronaux et des implantations précédentes de GENES. Le résultat de ce travail est le circuit intégré *GENES IV*, conçu conjointement par P. Ienne et l'auteur. La partie principale de cette thèse étudie l'architecture, la réalisation et l'analyse de la machine *MANTRA I*, un ordinateur neuronal basé sur un tableau GENES IV d'au plus 40×40 *processeurs élémentaires (PE)*. La règle delta (et donc les règles du Perceptron et de l'Adaline), la règle de la rétro-propagation du gradient, le modèle de Hopfield et le modèle de Kohonen peuvent être implantés sur ce système. Bien que n'étant pas un système générique, une telle machine peut être qualifiée d'*ordinateur neuronal multi-modèles*. Un prototype est fonctionnel depuis une année et utilisé quotidiennement par les concepteurs du logiciel.

Différentes caractéristiques distinguent la machine MANTRA I d'autres systèmes neuronaux. D'abord, elle fait partie des quelques ordinateurs neuronaux existants, au contraire de la majorité des réalisations, qui sont spécifiques à une application ou à un modèle. Aucun algorithme n'est câblé dans la machine, mais celle-ci fournit les primitives nécessaires à l'implantation des modèles visés. Il s'agit là d'une caractéristique essentielle pour la recherche, puisque plusieurs algorithmes ou versions d'un algorithme peuvent être testés sur un problème. C'est également un point important pour les applications, puisque différents modèles de RNA sont souvent utilisés conjointement pour résoudre un problème.

Le tableau systolique GENES—c'est-à-dire le cœur de la machine MANTRA I—vise un parallélisme au niveau des synapses (c'est-à-dire qu'un PE réel ou virtuel est alloué par synapse ou connexion vers un neurone), alors que la plupart des autres systèmes exploitent seulement le parallélisme au niveau des neurones (un PE par neurone). Ce système vise donc un grain de parallélisme beaucoup plus fin et convient donc bien comme architecture massivement parallèle.

La taille des problèmes qui peuvent être calculés par un accélérateur neuronal ne devrait pas être limitée par le système (mis à part la taille de la mémoire). Il est donc essentiel de permettre le partage des PE dans le temps. Avec la machine MANTRA I, ceci est possible grâce au concept de *tableaux virtuels*. Les matrices sont divisées en sous-matrices de la taille du tableau physique, qui est alors partagé entre elles dans le temps. Un mécanisme efficace permet l'échange de sous-matrices en arrière-plan, tandis que d'autres calculs sont effectués.

Comme les tableaux systoliques sont des systèmes en *pipeline*, il est important d'éviter de les vider et remplir trop souvent, afin de maintenir un *taux d'utilisation* élevé. À cet effet, un *flux systolique d'instructions* a été implanté, afin que chaque instruction suive les données pour lesquelles elle a été prévue.

Comme tout système SIMD, la machine MANTRA I est composée d'un *module parallèle* ou *SIMD* et d'un *module de contrôle*. Le module SIMD est formé du tableau GENES IV et d'une série d'unités spécialisées conçues pour les opérations qui sont proportionnelles au nombre de neurones et qui auraient été mal adaptées à une mise en œuvre au moyen d'un tableau bi-dimensionnel. Un système complexe de FIFOs et de mémoires fournit les flux d'entrée et de sortie nécessaires au tableau systolique.

Le module de contrôle est un système SISD complet. Ses tâches sont (1) de contrôler le module SIMD en générant ses instructions, (2) de gérer l'entrée et la sortie de données, (3) de communiquer avec le monde extérieur et (4) d'effectuer le pré- et post-traitement des données. Les instructions pour la partie SIMD sont codées *horizontalement* (VLIW). La synchronisation entre les deux modules est assurée par une FIFO d'instructions.

Les performances de la machine MANTRA I ont été analysées à l'aide de la règle delta. Les mesures montrent que celles-ci sont très proches de leurs valeurs de pointe, pour autant que le problème ne dépasse pas la capacité des banques de mémoire connectées au tableau GENES IV. Une autre série d'expériences a permis d'évaluer l'impact des contraintes imposées par la machine sur la convergence des algorithmes.

Finalement, l'utilisation de tableaux systoliques comme accélérateurs neuronaux est discutée à la lumière de l'expérience acquise avec le tableau GENES IV et la machine MANTRA I. Les faiblesses de la machine sont analysées et plusieurs solutions sont proposées pour les éviter dans une nouvelle version. Une discussion générale sur le futur des ordinateurs neuronaux clôt cette thèse.

Contents

List of figures

List of tables

List of algorithms

A computer is a large room full of equipment, clicking and clanking, whirring and flashing, satisfying the data processing needs of a large organization. It monitors inventory, bills customers, pays creditors, edits and types reports, sends out personalized advertising notices, communicates with other computers, and, most important, prints payroll checks every second Wednesday. On some days it seems like the computer does everything but cook lunch.

John F. Wakerly, *Microcomputer Architecture and Programming*

Introduction

The *artificial neural network (ANN)* domain and computer science have co-existed in the past five decades. It is interesting to notice that the pioneer work by W. McCulloch and W. Pitts on artificial neurons, published in 1943 [McC43], preceded the first electronic computer, the ENIAC, and is contemporary of electrical and electro-mechanical computing devices. The ENIAC machine, built at the University of Pennsylvania's Moore School of Electronical Engineering, became operational during the end of World War II, but, for military reasons, was publicly disclosed only in 1946: J. von Neumann, having heard from this machine, arranged for the Manhattan Project at Los Alamos to be its first major user.

J. von Neumann—whose name is associated with the "classical computer architecture[1]" often opposed to alternate computing paradigms such as ANNs—was himself very interested in the ANN domain. He was one of the brightest scientist of this century, who excelled in several domains—as different as mathematics, physics, and economics (he held a degree in chemistry and a Ph.D. in mathematics)—before establishing the foundations of computer science. In his "First Draft of a Report on the EDVAC" [Neu45] and in the following "Preliminary Discussion of the Logical Design of an Electronic Computing Instrument" [Bur47], he adopted a formalism and vocabulary borrowed to W. McCulloch and W. Pitts. Later, ANNs became an important part of his "General and Logical Theory of Automata" [Neu51].

The work of F. Rosenblatt on the Perceptron in 1958 [Ros58] represents a milestone, in the ANN field. A large scientific audience was immediately attracted by this type of device. Interest in

[1] Credit for the establishment of the stored-program concept, known as *von Neumann architecture*, has often been disputed. The first project that adopted this paradigm was the EDVAC (it underwent several delays and the EDSAC machine, though started later, became operational earlier). The confusion comes from the "First Draft of a Report on the EDVAC" [Neu45], written by J. von Neumann for internal purposes, that was distributed throughout the USA and UK without his knowledge. Although, most ideas came from common discussions with J. P. Eckert and J. W. Mauchly, the senior designers of the ENIAC and EDVAC systems, or from their previous work, J. von Neumann should get the credit of having scientifically analyzed an engineer project and clearly distinguished the major architectural concepts from implementation details. However, justice should also be done to J. P. Eckert and J. W. Mauchly, without whom there would have been no computer at all.

the domain increased steadily, until the publication of "Perceptrons" by M. Minsky and S. Papert in 1969 [MIN69]. This book has often been accused of having "killed" Perceptrons or even ANNs. In fact, it is only a thorough analysis of these systems that clearly shows the limits of Perceptrons. The reason, why almost nothing has then been produced in the field during more than a decade, is less because of this book than because powerful mathematical models were lacking and computing technology was not ready. The ANN domain entered a new era in the 80's, with the finding of models such as J. Hopfield's *recurrent networks* and T. Kohonen's *self-organizing feature maps* in 1982, and the *back-propagation rule* in 1985–1986.

Research on *multi-processing* started during the 60's with the pioneer work on the SOLOMON computer [SLO62] and the ILLIAC IV machine [BAR68, BOU72]. However, multi-chip processors are ill-suited for this type of machine, which remained confined to a very restricted niche of scientists until the 80's when the microprocessor (introduced by Intel in 1971 [FAG92]) had become mature enough to be used in parallel computers. Meanwhile, a very big research effort has been put on another form of parallelism: pipelining. The first pipelined machines were developed in the late 50's and early 60's with, among others, the IBM 7030 or Stretch [BLO59, BUC62]. This class of computers reached its apogee with vector processors—introduced in the 70's (CDC STAR-100 [HIN72], TI ASC [WAT72])—that remained the only super-computers until a few years ago.

Systolic arrays, invented by H. T. Kung and C. E. Leiserson in 1979 [KUN79], are direct descendants of pipelined processors. The former class may even be considered as a generalization of the latter, since, in systolic architectures, data streams are both multi-dimensional and multi-directional. Unlike other concepts discussed so far, systolic arrays were invented as a means of fully exploiting the possibilities of the VLSI technology of the time. Therefore, their success was immediate.

In the late 80's and early 90's, besides super-computers, there was no platform that was at the same time versatile enough to implement any ANN model and fast enough to be used on large problems. However, scientists required such a computing power to evaluate and improve their models, using very large data sets. Application engineers had the same computing needs, to repeatedly train an ANN in order to tune the model's parameters. Moreover, many possible applications would have required the processing power of super-computers without being able to afford such an expensive system. This state of things triggered many projects in the field. This is clearly illustrated by the organization of conferences like the "International Workshops on VLSI for Neural Networks and Artificial Intelligence," the "International Conferences on Microelectronics for Neural Networks," or the "IFIP Workshops on Silicon Architectures for Neural Nets," which are solely dedicated to this area.

The aim of the present thesis is to study efficient accelerators for ANN computation. For this purpose, a neural computer, the *MANTRA I* machine has been designed, built, and analyzed. This work is based on the *GENES* systolic arrays proposed by F. Blayo and C. Lehmann in two previous theses. These systems are bi-dimensional processor arrays dedicated to neural computation. A first prototype has been build for the Hopfield network. The first part of this thesis consisted in extending the GENES architecture towards generality (implementation of multiple models) and efficiency. This led to the *GENES IV* circuit. The architecture of this chip has been co-designed by P. Ienne and the author, P. Ienne having taken care of the implementation. The core of this thesis concerns the design and evaluation of the MANTRA I machine: an ANN computer based on the GENES IV array. The delta rule (and hence the Perceptron and Adaline rules), the back-propagation rule, the Hopfield model, and the Kohonen model can be implemented by this system.

In the GENES IV array, one *processing element (PE)* or virtual PE is allocated for each synapse (i.e., each connection between an input and a neuron or between two neurons). Therefore, this system features synapse-level parallelism, while most other commercial or research machines exploit only neuron-level parallelism. Therefore, it is well suited for fine-grain and massively parallel architectures.

To avoid limiting the size of applications by the size of a neural accelerator, time-sharing of PEs among different synapses should be possible. On the GENES IV array, this is implemented by dividing a *virtual matrix* into sub-matrices of the array's size. The system is then time-shared among them. Sub-matrices can be exchanged in background through a dedicated path. GENES IV operations have been designed so that cascading sub-matrix iterations does not result in any overhead.

The GENES IV array has been designed to maximize the hardware *utilization rate*. In particular, to avoid emptying and re-filling the pipeline at each operation change, a *systolic instruction flow* has been implemented. This means that instructions are dispatched to the first PE and then flow systolically through the array, following the corresponding data.

The MANTRA I machine is an SIMD neural computer based on a GENES IV array with up to 40×40 PEs. It consists of a *parallel* or *SIMD module* and a *control module*. The former consists essentially of the systolic array. In addition, it contains several units dedicated to operations that would have been poorly mapped on a bi-dimensional array. For instance, the activation function of neurons is implemented by a double look-up table scheme, featuring a *coarse-grain table* to map the whole input space with reduced precision and a *fine-grain* one to map a small window with high precision. A selection mechanism is used to position the latter table in the input space. It is also possible to modify its precision by trading window width for precision.

Distinct units are used to memorize the different categories of inputs and outputs. Hardware FIFO queues — simply referred to as FIFOs in this document — are used to enter and retrieve data to/from the SIMD module, while temporary results are stored in static RAMs. This system has been designed to meet the data bandwidth of the GENES IV array.

The control module of the MANTRA I machine is a microprocessor-based system that dispatches instructions to the SIMD module and handles data input and output. In addition, it is also in charge of the communication with the external world and may perform data pre- and post-processing. SIMD instructions are transferred through a FIFO. If instructions are lacking, the SIMD module is frozen, thus achieving synchronization. A counter makes it possible to repeat an instruction up to 128 times.

This thesis is divided into eight chapters. In chapter 1, the ANN models targeted by the MANTRA I machine are presented. A common formalism is adopted to help outlining the similarities between models. The goal of this chapter is not to provide a complete theory of ANNs — not even a summary it — but just to give an overview of the algorithms that can be implemented on the machine. For the sake of conciseness, no demonstrations or derivations of formulae have been included. Only a few of them are presented in appendix A.

Chapter 2 gives a short overview of multi-processor systems, followed by a survey of the most important neural computers found in the literature. The first part puts the emphasis on systolic arrays and related systems. Neural machines are succinctly described (without figures or block diagrams).

Chapter 3 is dedicated to the presentation of the work of F. Blayo and C. Lehmann on the GENES array. This work led to the integration of three VLSI circuits and to the implementation

of a small test system.

The latest chip in the GENES family, the GENES IV circuit, is presented in chapter 4. In the first part of this chapter, the requirements for this device are analyzed and the functional principles derived. The second part is devoted to the description of the implementation.

The architecture of the MANTRA I machine is described in chapter 5. For each unit, before describing the chosen implementation, the requirements are first analyzed and different solutions are evaluated. The integration of the machine in its environment is briefly addressed.

In chapter 6, two issues related to the programming of the machine are discussed: the mapping of algorithms onto the hardware and the use of integer data representation. The latter point is of capital importance, because the majority of ANN applications require a floating-point interface while, for the sake of hardware simplicity, integer arithmetic is often chosen for neural computers. Therefore, its implications on the theoretical models (including data re-scaling and conversion) should be neither neglected nor underestimated. Appendix B presents the mapping onto the machine of a few additional algorithms.

The performance of the machine is then analyzed in chapter 7. After a general discussion of metrics and peak performance, the effective performance of the machine is measured as a function of different parameters. The influence of algorithmic modifications imposed by the hardware (i.e., discrete values of the learning coefficient, epoch updating, and integer data representation) is then experimentally studied. This makes it possible to weight the former results with the latter. The benchmarks used for these experiments are described in appendix C.

Finally, chapter 8 discusses the use of systolic arrays as ANN accelerators, considering the experience gained with the GENES IV and MANTRA I projects. Weaknesses of the machine are analyzed and several solutions to avoid them in a re-design are proposed. A general discussion of the future of systolic arrays in the neural-computer domain concludes this chapter.

References

[AND88] J. A. Anderson and E. Rosenfeld (eds.). *Neurocomputing: Foundations of Research*. MIT Press, Cambridge, MA (USA), 1988.

[ASP87] W. Aspray and A. Burks (eds.). *Papers of John von Neumann on Computing and Computer Theory*, vol. 12 of *The Charles Babbage Institute Reprint Series for the History of Computing*. MIT Press, Cambridge, MA (USA), 1987.

[BAR68] G. H. Barnes, R. M. Brown, M. Kato, D. J. Kuck, D. L. Slotnick, and R. A. Stokes. *The ILLIAC IV Computer. IEEE Transactions on Computers*, 17(8):746–757, August 1968.

[BLO59] E. Bloch. *The Engineering Design of the Stretch Computer*. In *Proceedings of the Fall Joint Computer Conference*, pp. 48–59, 1959.

[BOU72] W. J. Bouknight, S. A. Denenberg, D. E. McIntyre, J. M. Randall, A. H. Sameh, and D. L. Slotnick. *The Illiac IV System. Proceedings of the IEEE*, 60(4):369–388, April 1972. Invited paper.

[BUC62] W. Bucholtz. *Planning a Computer System: Project Stretch*. McGraw-Hill, New York, NY (USA), 1962.

[BUR47] A. W. Burks, H. H. Goldstine, and J. von Neumann. *Preliminary Discussion of the Logical Design of an Electronic Computing Instrument.* Report for the U.S. Army Ordnance Department under contract W-36-034-ORD-7481, part I, vol. I, 2nd edition, Institute for Advanced Study, September 1947. Reprinted in [ASP87].

[FAG92] F. Faggin. *The Birth of the Microprocessor. Byte*, 17(3):145–150, March 1992.

[HIN72] R. G. Hintz and D. P. Tate. *Control Data STAR-100 Processor Design.* In *Proceedings of the IEEE COMPCON*, pp. 1–4. IEEE Computer Society, September 1972.

[KUN79] H. T. Kung and C. E. Leiserson. *Systolic Arrays (for VLSI).* In I. S. Duff and G. W. Stewart (eds.), *Sparse Matrix Proceedings 1978*, pp. 256–282. Society for Industrial and Applied Mathematics, Philadelphia, PA (USA), 1979.

[MCC43] W. S. McCulloch and W. Pitts. *A Logical Calculus of the Ideas Immanent in Nervous Activity. Bulletin of Mathematical Biophysics*, 5:115–133, 1943. Reprinted in [AND88].

[MIN69] M. Minsky and S. Papert. *Perceptrons.* MIT Press, Cambridge, MA (USA), 1969. Introduction reprinted in [AND88].

[NEU45] J. von Neumann. *First Draft of a Report on the EDVAC.* Report for the U.S. Army Ordnance Department under contract W-670-ORD-4926, 1945. Reprinted in [ASP87].

[NEU51] J. von Neumann. *General and Logical Theory of Automata.* In L. A. Jeffrees (ed.), *Cerebral Mechanisms in Behavior*, pp. 1–31. John Wiley & Sons, New York, NY (USA), 1951. Proceedings of the 1948 Hixon symposium.

[ROS58] F. Rosenblatt. *The Perceptron: A Probabilistic Model for Information Storage and Organization in the Brain. Psychological Review*, 65:386–408, 1958. Reprinted in [AND88].

[SLO62] D. L. Slotnick, W. C. Borck, and R. C. McReynolds. *The SOLOMON Computer.* In *Proceedings of AFIPS Fall Joint Computer Conference*, vol. 22, pp. 97–107. Spartan Books, 1962.

[WAT72] W. J. Watson. *The TI ASC–A Highly Modular and Flexible Super Computer Architecture.* In *Proceedings of the AFIPS Fall Joint Computer Conference*, vol. 41, pp. 221–228, Anaheim, CA (USA), December 1972. AFIPS Press.

1

Artificial Neural Networks

To those who do not know mathematics it is difficult to get across a real feeling as to the beauty, the deepest beauty, of nature. [...] I really think that those two cultures separate people who have and people who have not had this experience of understanding mathematics well enough to appreciate nature once.

Richard P. Feynman, *The Character of Physical Law*

This chapter presents a brief survey of the *artificial neural network (ANN)* models addressed in this thesis. A unique formalism is defined, and models are described accordingly. This outlines similarities between models and clarifies the architectural requirements of a multi-model neural computer. Since the mathematical study of ANN algorithms is at the frontier of this thesis, these presentations are rather succinct. The unfamiliar reader is referred to some general introductory book, for instance, the one by J. Hertz, A. Krogh, and R. Palmer [HER91].

1.1 Notations and terminology

In this document, matrices are denoted by upright bold characters. Vectors are treated as single-column matrices and are denoted with the usual horizontal arrow above the name. The elements of a matrix are referred to by the corresponding italic non-bold character and indices. For instance, $M_{i,j}$ is the element on the i^{th} row and j^{th} column of matrix $\mathbf{M}$, and v_i is the i^{th} element of vector $\vec{\mathbf{v}}$. A similar notation is used for matrix rows and columns, $\mathbf{M}_i$ being the i^{th} row of matrix $\mathbf{M}$, and $\vec{\mathbf{M}}_j$ its j^{th} column.

The terminology used in the ANN domain has been mostly borrowed from biology. The following list presents the original definition of the most important terms, along with the meaning adopted in this thesis:

Neuron

A grayish or reddish granular cell with specialized processes that is the fundamental functional unit of nervous tissues [WEB91]. In this thesis, this term denotes an elementary unit capable of computing a weighted sum or a distance and applying an arbitrary function to the previous result.

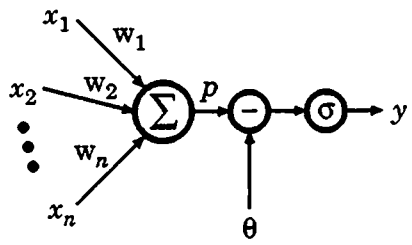


Figure 1.1: *The McCulloch and Pitts neuron.*

Synapse
The point at which a nervous impulse passes from one neuron to another [WEB91]. Here, this term refers to a connection between an input and a neuron or between neurons. A weight is associated to each synapse.

Axon
A usually long and single nerve-cell process that usually conducts impulses away from the cell body [WEB91]. In this document, no ANN meaning is associated with this term.

1.2 Artificial neurons

This section introduces some general concepts on isolated artificial neurons, while the remaining of this chapter presents properties and algorithms related to neural networks.

1.2.1 The McCulloch and Pitts neuron

The most popular artificial neuron has been proposed by W. McCulloch and W. Pitts in 1943 [McC43]. This device, shown in figure 1.1, has n *inputs* x_j, a *synaptic weight* w_j being associated to each of them. The weighted sum of these inputs is the *potential* p. The neuron's *output* y is given by the application of an arbitrary *activation function* σ to the potential from which an optional threshold θ has been subtracted:

$$y = \sigma(p - \theta) = \sigma\left(\sum_{j=1}^{n} w_j \cdot x_j - \theta\right) \tag{1.1}$$

Though this neuron is slightly more general than the original model,[1] it is referred to as *McCulloch and Pitts neuron* in this document. Appendix A.1 discusses some common activation functions.

1.2.2 Distance-based artificial neurons

In some more elaborate artificial neuron models—such as that used in the Kohonen algorithm (see section 1.10)—the weighted sum of McCulloch and Pitts neurons is replaced by an arbitrary distance between the input vector $\vec{\mathbf{x}} = (\ x_1\ \ x_2\ \ \dots\ \ x_n\ )^{\mathrm{T}}$ and the weight single-row matrix

[1] W. McCulloch and W. Pitts restricted possible activation functions to all-or-none responses (see appendix A.1).

$\mathbf{w} = (\ w_1 \quad w_2 \quad \ldots \quad w_n\ )$. Common distances are the *Euclidean* and *Manhattan distances*, given by equations (1.2) and (1.3) respectively:

$$p = \Delta(\mathbf{w}^{\mathrm{T}}, \vec{\mathbf{x}}) = \sqrt{\sum_{j=1}^{n} (x_j - w_j)^2} \tag{1.2}$$

$$p = \Delta(\mathbf{w}^{\mathrm{T}}, \vec{\mathbf{x}}) = \sum_{j=1}^{n} |x_j - w_j| \tag{1.3}$$

1.3 Learning

One of the fundamental properties of ANNs is their learning ability. Networks learn by modifying their parameters (synaptic weights and thresholds) and/or their structure. The scope of this thesis extends only to fixed-size networks, and the second type of learning is not considered. In this chapter, each network consists of m neurons. It is connected to n inputs and produces m_L outputs. In single-layer networks, each neuron corresponds to an output of the ANN (i.e., $m_L = m$), while, in multi-layer networks, only the neurons on the last layer are available as outputs.

Since this thesis aims at digital implementations, the time is considered as discretized. Though called "time" because of its physics origin, the discretized time may also be considered as a loop index in an iterative process. For convenience, two such indices are distinguished: the update time t at which network parameters are updated and the iteration number τ of recurrent networks.

The learning or testing process is based on a set of S input vectors called *prototypes*:

$$\vec{\mathbf{x}}(s) = (\ x_1(s) \quad x_2(s) \quad \ldots \quad x_n(s)\ )^{\mathrm{T}} \qquad \text{for } s = 1, 2, \ldots, S$$

Usually, a training set of S_{lrn} prototypes is defined to learn an application, and a distinct testing set of S_{tst} elements is used to evaluate the quality of the ANN solution.

Since the weights and thresholds are functions of the update time t, the potential and output vectors[2] depend on the prototype s, the time t, and the iteration number τ:

$$\vec{\mathbf{p}}(t,s,\tau) = (\ p_1(t,s,\tau) \quad p_2(t,s,\tau) \quad \ldots \quad p_m(t,s,\tau)\ )^{\mathrm{T}} \qquad \text{for } s = 1, 2, \ldots, S$$

$$\vec{\mathbf{y}}(t,s,\tau) = (\ y_1(t,s,\tau) \quad y_2(t,s,\tau) \quad \ldots \quad y_m(t,s,\tau)\ )^{\mathrm{T}} \qquad \text{for } s = 1, 2, \ldots, S$$

The learning mechanism is either *supervised* or *unsupervised*. In the former case, a desired-output vector is associated with each prototype:

$$\vec{\mathbf{d}}(s) = (\ d_1(s) \quad d_2(s) \quad \ldots \quad d_{m_L}(s)\ )^{\mathrm{T}} \qquad \text{for } s = 1, 2, \ldots, S$$

1.4 ANN topologies

This section discusses the three most common ANN topologies: single-layer, recurrent, and multi-layer feed-forward networks.

[2] With multi-layer networks, the ANN state is usually given as a set of L pairs of potential and output vectors $\vec{\mathbf{p}}^{[k]}$ and $\vec{\mathbf{y}}^{[k]}$ (where L is the number of layers), rather than a single pair as in this general definition.

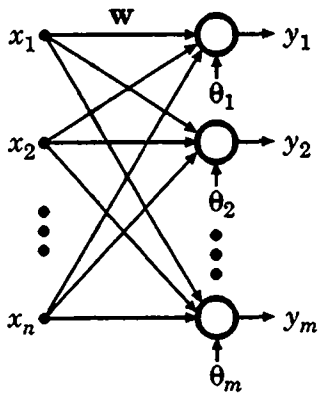


Figure 1.2: *Fully-connected single-layer network.*

1.4.1 Single-layer networks

The simplest ANNs feature only connections from inputs to neurons, but no inter-neuron connections. In the most general form, every input is connected to every neuron as shown in figure 1.2. Should this not be the case, unused connections are set to zero. This topology is called *single-layer* or *direct network*. With m neurons and n inputs, the function realized by this network—or the knowledge coded in it—is described by a weight matrix and a threshold vector:

$$\mathbf{w}(t) = \begin{pmatrix} w_{1,1}(t) & w_{1,2}(t) & \cdots & w_{1,n}(t) \\ w_{2,1}(t) & w_{2,2}(t) & \cdots & w_{2,n}(t) \\ \vdots & \vdots & \ddots & \vdots \\ w_{m,1}(t) & w_{m,2}(t) & \cdots & w_{m,n}(t) \end{pmatrix}$$

$$\vec{\theta}(t) = \begin{pmatrix} \theta_1(t) & \theta_2(t) & \ldots & \theta_m(t) \end{pmatrix}^{\mathrm{T}}$$

Using matrix and vector notation,[3] a single-layer network of McCulloch and Pitts neurons computes the following output vector:

$$\vec{\mathbf{y}}(t,s) = \sigma\left(\vec{\mathbf{p}}(t,s) - \vec{\theta}(t)\right) = \sigma\left(\mathbf{w}(t) \cdot \vec{\mathbf{x}}(s) - \vec{\theta}(t)\right) \tag{1.4}$$

When the threshold vector is constant (i.e., $\vec{\theta}(t) = \vec{\theta}$) with equal components (i.e., $\theta_i = \theta$ for $i = 1, 2, \ldots, m$), it may be considered as part of the activation function (i.e., $\sigma_{\text{eff}}(v) = \sigma(v - \theta)$). However, in several models, the thresholds are learned by the network. When they follow the same learning rule as the synaptic weights, thresholds can be considered as additional weights associated with a constant unity input. The input vector and the weight matrix must then be recast as:

$$\vec{\mathbf{x}}^{\theta}(s) = \begin{pmatrix} x_1(s) & x_2(s) & \ldots & x_n(s) & 1 \end{pmatrix}^{\mathrm{T}} \quad \text{for } s = 1, 2, \ldots, S \tag{1.5}$$

[3] The notation $\sigma(\vec{\mathbf{v}})$ is used to designate the vector resulting from the application of the function σ to each element of the vector $\vec{\mathbf{v}}$.

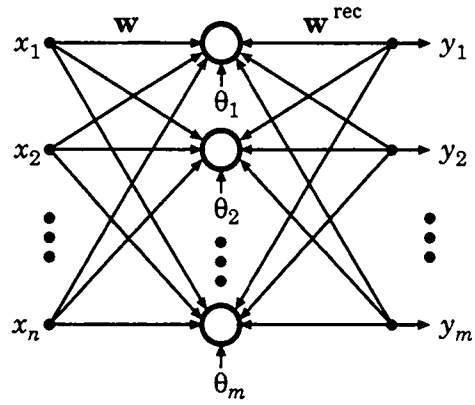


Figure 1.3: *Fully-connected direct and recurrent networks.*

$$\mathbf{w}^{\theta}(t) = \begin{pmatrix} w_{1,1}(t) & w_{1,2}(t) & \cdots & w_{1,n}(t) & -\theta_1(t) \\ w_{2,1}(t) & w_{2,2}(t) & \cdots & w_{2,n}(t) & -\theta_2(t) \\ \vdots & \vdots & \ddots & \vdots & \vdots \\ w_{m,1}(t) & w_{m,2}(t) & \cdots & w_{m,n}(t) & -\theta_m(t) \end{pmatrix} \tag{1.6}$$

Equation (1.4) now becomes:

$$\vec{\mathbf{y}}(t,s) = \sigma\left(\vec{\mathbf{p}}^{\theta}(t,s)\right) = \sigma\left(\mathbf{w}^{\theta}(t) \cdot \vec{\mathbf{x}}^{\theta}(s)\right) \tag{1.7}$$

The reader should keep in mind that this technique is only valid for McCulloch and Pitts neurons. In the remaining of this document, only direct networks without thresholds[4] (i.e., described by equation (1.4) with $\vec{\theta}(t) = \vec{\mathbf{0}}$) or with thresholds treated as weights (i.e., described by equation (1.7)) are considered. When a model offers both possibilities, the symbols $\mathbf{w}^*$, $\vec{\mathbf{x}}^*$, and $\vec{\mathbf{p}}^*$ are used to replace either $\mathbf{w}$, $\vec{\mathbf{x}}$, and $\vec{\mathbf{p}}$ in the first case or $\mathbf{w}^{\theta}$, $\vec{\mathbf{x}}^{\theta}$, and $\vec{\mathbf{p}}^{\theta}$ in the second case. Hence, This notation offers the possibility of analyzing both cases together.

1.4.2 Recurrent networks

A *recurrent network* contains inter-neuron connections with loops. Therefore, the output of such a system depends not only on its inputs but also on its current state. The most general form is the *fully-connected recurrent network*, shown in figure 1.3, where each neuron is connected to all others (including itself). The function of an m-neuron network of this type is described by a weight matrix and a threshold vector:

$$\mathbf{w}^{\text{rec}}(t) = \begin{pmatrix} w^{\text{rec}}_{1,1}(t) & w^{\text{rec}}_{1,2}(t) & \cdots & w^{\text{rec}}_{1,m}(t) \\ w^{\text{rec}}_{2,1}(t) & w^{\text{rec}}_{2,2}(t) & \cdots & w^{\text{rec}}_{2,m}(t) \\ \vdots & \vdots & \ddots & \vdots \\ w^{\text{rec}}_{m,1}(t) & w^{\text{rec}}_{m,2}(t) & \cdots & w^{\text{rec}}_{m,m}(t) \end{pmatrix}$$

$$\vec{\theta}^{\text{rec}}(t) = \begin{pmatrix} \theta^{\text{rec}}_1(t) & \theta^{\text{rec}}_2(t) & \ldots & \theta^{\text{rec}}_m(t) \end{pmatrix}^{\text{T}}$$

[4] Constant thresholds can be considered as part of the activation function.

As shown in figure 1.3, this type of network is always associated with a direct network connecting inputs to neurons. Both networks can interact in two basic ways, described below as *type I* and *type II*. The two schemes do not represent different types of networks, but rather different possibilities of cascading a direct and a recurrent network.

Recurrent networks are often used to reach an asymptotically stable pattern, as in the Hopfield and Kohonen models (see sections 1.9 and 1.10.2). If the network converges, the final output is defined as:

$$\vec{\mathbf{y}}^{\,\text{stable}}(t,s) \;=\; \lim_{\tau\to\infty} \vec{\mathbf{y}}(t,s,\tau) \tag{1.8}$$

It is beyond the scope of this thesis to discuss the convergence conditions, which are heavily model-dependent. In practice, the output $\vec{\mathbf{y}}^{\,\text{stable}}$ may sometimes be approximated as the network's state after a fixed number $\tau_{\max}$ of iterations:

$$\vec{\mathbf{y}}^{\,\text{stable}}(t,s) \;\approx\; \vec{\mathbf{y}}(t,s,\tau_{\max}) \tag{1.9}$$

Type I recurrent networks

In the first interconnection scheme, the direct network is used to generate an initial value for the recurrent network. Thus, it is only active during the first cycle (i.e., $\tau = 0$). A stable pattern is then reached by the recurrent network alone. Although not shown in figure 1.3, the threshold vector $\vec{\theta}$ and the activation function σ of the direct network may be different from those of the recurrent network (i.e., $\vec{\theta}^{\,\text{rec}}$ and σ^{rec}). With McCulloch and Pitts neurons, this system computes:

$$\vec{\mathbf{y}}(t,s,0) \;=\; \sigma\left(\vec{\mathbf{p}}(t,s,0) - \vec{\theta}(t)\right) \;=\; \sigma\left(\mathbf{w}(t)\cdot\vec{\mathbf{x}}(s) - \vec{\theta}(t)\right) \tag{1.10}$$

$$\begin{aligned}\vec{\mathbf{y}}(t,s,\tau) \;&=\; \sigma^{\text{rec}}\left(\vec{\mathbf{p}}(t,s,\tau) - \vec{\theta}^{\,\text{rec}}(t)\right)\\ &=\; \sigma^{\text{rec}}\left(\mathbf{w}^{\text{rec}}(t)\cdot\vec{\mathbf{y}}(t,s,\tau-1) - \vec{\theta}^{\,\text{rec}}(t)\right) \qquad \text{for } \tau = 1,2,3,\ldots\end{aligned} \tag{1.11}$$

The direct network may be as simple as just providing initial values for all neurons (i.e., $\mathbf{w}(t) = \mathbf{I}$, $\vec{\theta}(t) = \vec{\mathbf{0}}$, and $\sigma(v) = v$). The technique described in section 1.4.1 can be used again to merge the threshold vectors $\vec{\theta}$ and $\vec{\theta}^{\,\text{rec}}$ with the weight matrices $\mathbf{w}$ and $\mathbf{w}^{\text{rec}}$, respectively:

$$\mathbf{w}^{\theta}(t) \;=\; \begin{pmatrix} w_{1,1}(t) & w_{1,2}(t) & \cdots & w_{1,n}(t) & -\theta_1(t)\\ w_{2,1}(t) & w_{2,2}(t) & \cdots & w_{2,n}(t) & -\theta_2(t)\\ \vdots & \vdots & \ddots & \vdots & \vdots\\ w_{m,1}(t) & w_{m,2}(t) & \cdots & w_{m,n}(t) & -\theta_m(t)\\ 0 & 0 & \cdots & 0 & \sigma^{-1}(1) \end{pmatrix} \tag{1.12}$$

$$\mathbf{w}^{\text{rec}\,\theta}(t) \;=\; \begin{pmatrix} w^{\text{rec}}_{1,1}(t) & w^{\text{rec}}_{1,2}(t) & \cdots & w^{\text{rec}}_{1,m}(t) & -\theta^{\text{rec}}_1(t)\\ w^{\text{rec}}_{2,1}(t) & w^{\text{rec}}_{2,2}(t) & \cdots & w^{\text{rec}}_{2,m}(t) & -\theta^{\text{rec}}_2(t)\\ \vdots & \vdots & \ddots & \vdots & \vdots\\ w^{\text{rec}}_{m,1}(t) & w^{\text{rec}}_{m,2}(t) & \cdots & w^{\text{rec}}_{m,m}(t) & -\theta^{\text{rec}}_m(t)\\ 0 & 0 & \cdots & 0 & \sigma^{\text{rec}\,-1}(1) \end{pmatrix} \tag{1.13}$$

where the notation σ^{-1} (or $\sigma^{\mathrm{rec}\,-1}$) is used to denote the inverse function of σ (respectively σ^{rec}). The reader should notice that the matrices $\mathbf{w}^{\theta}$ and $\mathbf{w}^{\mathrm{rec}\,\theta}$ have an additional row that can be considered as the weights of a *pseudo-neuron* whose output constantly stays at one. During the learning process, this row must not be updated. This happens to be implicit for some rules, but, in general, the update operation must be explicitly prevented.

The input vector $\vec{\mathbf{x}}^{\theta}$ being defined as in equation (1.5), equations (1.10) and (1.11) become:

$$\begin{aligned} \vec{\mathbf{y}}^{\theta}(t,s,0) &= \sigma\left(\vec{\mathbf{p}}^{\theta}(t,s,0)\right) = \sigma\left(\mathbf{w}^{\theta}(t)\cdot\vec{\mathbf{x}}^{\theta}(s)\right) && (1.14)\\ \vec{\mathbf{y}}^{\theta}(t,s,\tau) &= \sigma^{\mathrm{rec}}\left(\vec{\mathbf{p}}^{\theta}(t,s,\tau)\right) \\ &= \sigma^{\mathrm{rec}}\left(\mathbf{w}^{\mathrm{rec}\,\theta}(t)\cdot\vec{\mathbf{y}}^{\theta}(t,s,\tau-1)\right) \qquad \text{for } \tau = 1,2,3,\ldots && (1.15)\end{aligned}$$

Like for single-layer networks, explicit thresholds are not considered any more, and a star version ($*$-notation) of the symbols is used, when a model is valid both without thresholds or with thresholds treated as weights (θ-notation).

Type II recurrent networks

In the second interconnection type, the direct and recurrent networks are simultaneously active. For McCulloch and Pitts neurons, the potential vector $\vec{\mathbf{p}}$ is the weighted sum of the direct and recurrent connections. Hence, there is a unique threshold vector $\vec{\theta}$ and a unique activation function σ. The output of this system is:

$$\begin{aligned} \vec{\mathbf{y}}(t,s,\tau) &= \sigma\left(\vec{\mathbf{p}}(t,s,\tau) - \vec{\theta}(t)\right) \\ &= \sigma\left(\mathbf{w}(t)\cdot\vec{\mathbf{x}}(s) + \mathbf{w}^{\mathrm{rec}}(t)\cdot\vec{\mathbf{y}}(t,s,\tau-1) - \vec{\theta}(t)\right) \qquad \text{for } \tau = 1,2,3,\ldots && (1.16)\end{aligned}$$

The initial state vector $\vec{\mathbf{y}}(t,s,0)$ is set to a predefined value, for example $\vec{\mathbf{0}}$. To merge the threshold vector $\vec{\theta}$ and the weight matrix $\mathbf{w}$, it is sufficient to use the input vector $\vec{\mathbf{x}}^{\theta}$ and the weight matrix $\mathbf{w}^{\theta}$ defined by equations (1.5) and (1.6). Equation (1.16) then becomes:

$$\begin{aligned} \vec{\mathbf{y}}(t,s,\tau) &= \sigma\left(\vec{\mathbf{p}}^{\theta}(t,s,\tau)\right) \\ &= \sigma\left(\mathbf{w}^{\theta}(t)\cdot\vec{\mathbf{x}}^{\theta}(s) + \mathbf{w}^{\mathrm{rec}}(t)\cdot\vec{\mathbf{y}}(t,s,\tau-1)\right) \qquad \text{for } \tau = 1,2,3,\ldots && (1.17)\end{aligned}$$

1.4.3 Multi-layer feed-forward networks

A *multi-layer feed-forward network* is the juxtaposition of L single-layer networks such that the m_k outputs of a layer k are the inputs of the next one, as illustrated in figure 1.4. The function of this type of network is defined by a set of L weight matrices and L threshold vectors:

$$\mathbf{w}^{[k]}(t) = \begin{pmatrix} w^{[k]}_{1,1}(t) & w^{[k]}_{1,2}(t) & \cdots & w^{[k]}_{1,m_{k-1}}(t) \\ w^{[k]}_{2,1}(t) & w^{[k]}_{2,2}(t) & \cdots & w^{[k]}_{2,m_{k-1}}(t) \\ \vdots & \vdots & \ddots & \vdots \\ w^{[k]}_{m_k,1}(t) & w^{[k]}_{m_k,2}(t) & \cdots & w^{[k]}_{m_k,m_{k-1}}(t) \end{pmatrix} \qquad \text{for } k = 1,2,\ldots,L$$

$$\vec{\theta}^{[k]}(t) = \begin{pmatrix} \theta^{[k]}_1(t) & \theta^{[k]}_2(t) & \ldots & \theta^{[k]}_{m_k}(t) \end{pmatrix}^{\mathrm{T}} \qquad \text{for } k = 1,2,\ldots,L$$

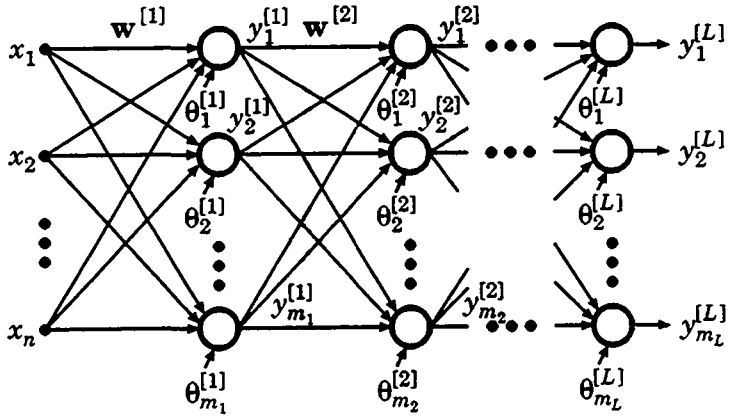


Figure 1.4: *Multi-layer feed-forward network.*

Since the inputs are connected to the first layer, m_0 is defined as the number of inputs (i.e., $m_0 = n$). With McCulloch and Pitts neurons, the output vector of each layer is equal to:

$$\begin{aligned} \vec{\mathbf{y}}^{[k]}(t,s) &= \sigma\left(\vec{\mathbf{p}}^{[k]}(t,s) - \vec{\theta}^{[k]}(t)\right) \\ &= \sigma\left(\mathbf{w}^{[k]}(t) \cdot \vec{\mathbf{y}}^{[k-1]}(t,s) - \vec{\theta}^{[k]}(t)\right) \qquad \text{for } k = 1,2,\ldots,L \end{aligned} \tag{1.18}$$

where $\vec{\mathbf{y}}^{[0]}(t,s) = \vec{\mathbf{x}}(s)$. Like for single-layer and recurrent networks, thresholds can be considered as weights. This is done by adding a pseudo-neuron to each layer but the last one. The weight matrix for the hidden layers can then be rewritten as:

$$\mathbf{w}^{\theta[k]}(t) = \begin{pmatrix} w_{1,1}^{[k]}(t) & w_{1,2}^{[k]}(t) & \cdots & w_{1,m_{k-1}}^{[k]}(t) & -\theta_1^{[k]}(t) \\ w_{2,1}^{[k]}(t) & w_{2,2}^{[k]}(t) & \cdots & w_{2,m_{k-1}}^{[k]}(t) & -\theta_2^{[k]}(t) \\ \vdots & \vdots & \ddots & \vdots & \vdots \\ w_{m_k,1}^{[k]}(t) & w_{m_k,2}^{[k]}(t) & \cdots & w_{m_k,m_{k-1}}^{[k]}(t) & -\theta_{m_k}^{[k]}(t) \\ 0 & 0 & \cdots & 0 & \sigma^{-1}(1) \end{pmatrix} \tag{1.19}$$

$$\text{for } k = 1,2,\ldots,L-1$$

While the weight matrix of the last layer is given by:

$$\mathbf{w}^{\theta[L]}(t) = \begin{pmatrix} w_{1,1}^{[L]}(t) & w_{1,2}^{[L]}(t) & \cdots & w_{1,m_{L-1}}^{[L]}(t) & -\theta_1^{[L]}(t) \\ w_{2,1}^{[L]}(t) & w_{2,2}^{[L]}(t) & \cdots & w_{2,m_{L-1}}^{[L]}(t) & -\theta_2^{[L]}(t) \\ \vdots & \vdots & \ddots & \vdots & \vdots \\ w_{m_L,1}^{[L]}(t) & w_{m_L,2}^{[L]}(t) & \cdots & w_{m_L,m_{L-1}}^{[L]}(t) & -\theta_{m_L}^{[L]}(t) \end{pmatrix} \tag{1.20}$$

Equation (1.18) now becomes:

$$\vec{\mathbf{y}}^{\theta[k]}(t,s) = \sigma\left(\vec{\mathbf{p}}^{\theta[k]}(t,s)\right) = \sigma\left(\mathbf{w}^{\theta[k]}(t) \cdot \vec{\mathbf{y}}^{\theta[k-1]}(t,s)\right) \qquad \text{for } k = 1,2,\ldots,L \tag{1.21}$$

Since there is no pseudo-neuron on the last layer, the corresponding output vector $\vec{\mathbf{y}}^{\theta[L]}$ is equal to $\vec{\mathbf{y}}^{[L]}$. Like for the other topologies the $*$-notation can be used to refer either to networks without

thresholds or to networks with thresholds treated as weights, when both cases can be analyzed in a uniform way.

1.4.4 Mixed topologies

Many other topologies—such as multi-layer networks with intra-layer recurrent connections or multi-layer networks with input connections to all layers, etc.—have also been proposed. Being seldom used, they are not described here. It can be noticed that any topology is embedded in a system formed by a fully-connected direct network and a fully-connected recurrent network. However, in most application, the resulting weight matrices would be much too sparse for efficient implementations.

1.5 The Hebbian learning rule

As stated in section 1.3, ANNs learn by modifying their parameters. The way weights are updated is defined by a *learning rule*. One of the most fundamental rule is derived from the neuro-physiological work of D. Hebb [HEB49], who states:

> *When an axon of cell* A *is near enough to excite a cell* B *and repeatedly or persistently takes part in firing it, some growth process or metabolic change takes place in one or both cells such that* A*'s efficiency, as one of the cells firing* B*, is increased.*

This principle can be generalized by updating every synaptic weight proportionally to the product of the corresponding input and output. For a direct network, this can be expressed as:

$$\Delta \mathbf{w}^* \quad \propto \quad \vec{\mathbf{y}} \cdot \vec{\mathbf{x}}^{*\mathrm{T}} \tag{1.22}$$

Therefore, weights tend to be increased when the corresponding inputs and outputs are analogous (same sign), and to be decreased when they are dissimilar (opposite sign).

When the weights are updated after each presentation, the prototype number s is a function of the time t. This learning paradigm is referred to as *on-line*. In the simplest case, when the prototypes are repeatedly presented, this function is simply: $s(t) = (t \bmod S) + 1$. However, in many applications, it is more complex as, for instance, a pseudo-random series. To simplify the notations, network parameters (weights, inputs, outputs, etc.) are considered as functions like those used in physics, rather than pure mathematical functions. This means that the following shorthands are, for instance, used: $\vec{\mathbf{x}}(t) = \vec{\mathbf{x}}(s(t))$ and $\vec{\mathbf{y}}(t) = \vec{\mathbf{y}}(s(t), t)$. Relation (1.22) can then be rewritten as:

$$\mathbf{w}^*(t+1) \quad = \quad \mathbf{w}^*(t) + \alpha(t) \cdot \vec{\mathbf{y}}(t) \cdot \vec{\mathbf{x}}^{*\mathrm{T}}(t) \tag{1.23}$$

where α is a time-decreasing *learning coefficient*. It is clear from the notations that equation (1.23) is not directly applicable to topologies that introduce a pseudo-neuron to merge the thresholds with the weight matrix.

1.5.1 The Hebbian learning rule with epoch updating

Another learning paradigm is to present a set or *epoch* of prototypes between weight updates. The Hebbian learning rule becomes then:

$$\mathbf{w}^*(t+1) = \mathbf{w}^*(t) + \alpha(t) \cdot \sum_{s \in E(t)} \vec{\mathbf{y}}(t,s) \cdot \vec{\mathbf{x}}^{*\mathrm{T}}(s) \tag{1.24}$$

where all elements of the set E belong to the range of prototypes $[1, S]$. A possible policy is to present them sequentially, by group of e prototypes per epoch, until a full presentation is completed:

$$E(t) = \left[\left(t \bmod \left\lceil \frac{S}{e} \right\rceil\right) \cdot e + 1,\ \min\left(\left(t \bmod \left\lceil \frac{S}{e} \right\rceil\right) \cdot e + e, S\right)\right] \tag{1.25}$$

where the notation $\lceil v \rceil$ represents the ceiling function, defined as the smallest integer that is larger or equal to v. It should be noticed that on-line learning is a special case of epoch updating where the length of every epoch is equal to one. On the other hand, *batch* updating is the special case when each epoch consists of all prototypes (i.e., $E(t) = [1, S]$).

1.6 The Perceptron learning rule

The Perceptron is a type of ANN introduced by F. Rosenblatt in 1958 [ROS58]. This network has multiple layers, but only the last one is subject to learning. The *Perceptron learning rule* presented here, is the weight-update algorithm proposed by H. Block [BLO62] for this layer. Many variations of the Perceptron have then been proposed, as listed by M. Minsky and S. Papert in "Perceptrons" [MIN69]. In this book, the authors analyze this type of device and clearly show the limitations of Perceptrons. This book delivered an almost fatal blow to the already declining interest in Perceptrons. Nowadays, many authors use the term "Perceptron" for any single-layer non-recurrent network (see section 1.4.1) and the term "multi-layer Perceptron" for any multi-layer feed-forward network (see section 1.4.3). To avoid ambiguities, the longer—but less confusing—descriptive names are preferred in this document.

The Perceptron rule is an on-line supervised learning algorithm (i.e., $s = s(t)$) for single-layer networks (see section 1.4.1) of McCulloch and Pitts binary neurons. This type of ANN is described by equation (1.4) or (1.7) using the sign function (i.e., $\sigma(v) = \mathrm{sign}(v)$). For each prototype the error vector is computed:

$$\vec{\varepsilon}(t) = \vec{\mathbf{d}}(t) - \vec{\mathbf{y}}(t) = \vec{\mathbf{d}}(t) - \mathrm{sign}(\mathbf{w}^*(t) \cdot \vec{\mathbf{x}}^*(t)) \tag{1.26}$$

The learning rule aims at minimizing each element of this vector. Since the elements of the output vector $\vec{\mathbf{y}}$ and the desired-output vector $\vec{\mathbf{d}}$ are either $+1$ or -1, the situation can easily be analyzed. Table 1.1 gives the direction of update (i.e., the sign of $\Delta w_{i,j}$) for each possible outcome. Although in the original algorithm [BLO62], the inputs x_j are restricted to 0 and 1, the generalization to $\mathbb{R}$ is easily justified.[5] An analysis of table 1.1 shows that the increment $\Delta w_{i,j}$ has the same sign as

[5] Since the philosophy of this rule is to update the weights with fixed increments, the inputs x_j are usually restricted to two values, either 0 and 1 or $+1$ and -1 (or more generally 0 and a or $+b$ and $-b$).

	$d_i(t) = +1$		$d_i(t) = -1$	
	$\varepsilon_i(t) = 0$	$\varepsilon_i(t) = +2$	$\varepsilon_i(t) = 0$	$\varepsilon_i(t) = -2$
$x_j(t) < 0$	$\Delta w_{ij}(t) = 0$	$\Delta w_{ij}(t) < 0$	$\Delta w_{ij}(t) = 0$	$\Delta w_{ij}(t) > 0$
$x_j(t) = 0$	$\Delta w_{ij}(t) = 0$	$\Delta w_{ij}(t) = 0$	$\Delta w_{ij}(t) = 0$	$\Delta w_{ij}(t) = 0$
$x_j(t) > 0$	$\Delta w_{ij}(t) = 0$	$\Delta w_{ij}(t) > 0$	$\Delta w_{ij}(t) = 0$	$\Delta w_{ij}(t) < 0$

Table 1.1: *The Perceptron learning rule.*

the product of ε_i and x_j, giving the learning rule:

$$\mathbf{w}^*(t+1) = \mathbf{w}^*(t) + \alpha(t) \cdot \left(\vec{\mathbf{d}}(t) - \vec{\mathbf{y}}(t)\right) \cdot \vec{\mathbf{x}}^{*\mathrm{T}}(t) \tag{1.27}$$

1.6.1 The Perceptron learning rule with epoch updating

The Perceptron rule given by equation (1.27) can easily be adapted to epoch updating:

$$\mathbf{w}^*(t+1) = \mathbf{w}^*(t) + \alpha(t) \cdot \sum_{s \in E(t)} \left(\vec{\mathbf{d}}(s) - \vec{\mathbf{y}}(t,s)\right) \cdot \vec{\mathbf{x}}^{*\mathrm{T}}(s) \tag{1.28}$$

This algorithm tends to minimize the components of the total error vector $\sum_{s \in E(t)} \vec{\varepsilon}(t,s)$ over each epoch.

1.7 The delta rule

The *delta rule* is a supervised learning algorithm for single-layer networks (see section 1.4.1) of McCulloch and Pitts neurons, whose activation function is differentiable (i.e., $\sigma'(v) = \frac{\mathrm{d}}{\mathrm{d}v}\sigma(v)$). Equation (1.4) or (1.7) describes the behavior of these networks. Using a stochastic gradient descent, the delta rule minimizes the quadratic error of each neuron i, defined as:

$$\xi_i(t) = \frac{1}{2} \cdot \varepsilon_i^2(t) = \frac{1}{2} \cdot (d_i(t) - y_i(t))^2 = \frac{1}{2} \cdot (d_i(t) - \sigma(\mathbf{w}_i^*(t) \cdot \vec{\mathbf{x}}^*(t)))^2 \tag{1.29}$$

The weight updating rule, derived in appendix A.2, is equal to:

$$\begin{aligned} \mathbf{w}^*(t+1) &= \mathbf{w}^*(t) + \alpha(t) \cdot \vec{\delta}(t) \cdot \vec{\mathbf{x}}^{*\mathrm{T}}(t) \\ &= \mathbf{w}^*(t) + \alpha(t) \cdot \left(\left(\vec{\mathbf{d}}(t) - \vec{\mathbf{y}}(t)\right) \circ \sigma'(\vec{\mathbf{p}}^*(t))\right) \cdot \vec{\mathbf{x}}^{*\mathrm{T}}(t) \end{aligned} \tag{1.30}$$

where the symbol $\circ$ represents the *Hadamard product*, that is, the term-to-term multiplication of two matrices or vectors of identical size. The delta rule has been named after the error signal vector $\vec{\delta}(t) = (\vec{\mathbf{d}}(t) - \vec{\mathbf{y}}(t)) \circ \sigma'(\vec{\mathbf{p}}^*(t))$.

This method has been first proposed by B. Widrow and M. Hoff [WID60], for a neuron model called *Adaline* (contraction of *ada*ptive *line*ar). Though the activation function of Adalines is the sign function and hence is non-differentiable, the learning process is based on the potential p_i^* instead of the output y_i. Therefore, as far as learning is concerned, this rule is equivalent to equation (1.30) with the identity as activation function (i.e., $\sigma(v) = v$ and thus $\sigma'(v) = 1$). This

system has been extended to multiple neurons and the learning rule improved in the *Madaline I, II,* and *III* models (contraction of *m*ultiple *Adalines*). These algorithms, together with many others, are reviewed by B. Widrow and M. Lehr [WID90].

The delta rule is also often called *least mean square (LMS) procedure.* Though the term "mean" is better suited for the epoch or batch versions, it is used for the on-line method as well. B. Widrow and M. Hoff [WID60] explain this as follows:

> *The square of the error for a single pattern (the mean square error for a sample size of one) is given* [...]

1.7.1 The delta rule with epoch updating

The extension of the delta rule to epoch updating corresponds to the minimization, using the gradient descent, of the total quadratic error:

$$\xi_i(t) = \frac{1}{2} \cdot \sum_{s \in E(t)} \varepsilon_i^2(t,s) = \frac{1}{2} \cdot \sum_{s \in E(t)} \left(d_i(s) - \sigma\left(\mathbf{w}_i^*(t) \cdot \vec{\mathbf{x}}^*(s)\right)\right)^2 \tag{1.31}$$

Thanks to the linearity of the differentiation, the sum can be extracted from the gradient in the proof of appendix A.2. This leads to the rule:

$$\begin{aligned} \mathbf{w}^*(t+1) &= \mathbf{w}^*(t) + \alpha(t) \cdot \sum_{s \in E(t)} \vec{\delta}(t,s) \cdot \vec{\mathbf{x}}^{*\mathrm{T}}(s) \\ &= \mathbf{w}^*(t) + \alpha(t) \cdot \sum_{s \in E(t)} \left(\left(\vec{\mathbf{d}}(s) - \vec{\mathbf{y}}(t,s)\right) \circ \sigma'\left(\vec{\mathbf{p}}^*(t,s)\right)\right) \cdot \vec{\mathbf{x}}^{*\mathrm{T}}(s) \end{aligned} \tag{1.32}$$

The rule in batch mode is given by equation (1.32) with $E(t) = [1,\ S]$.

1.8 The back-propagation rule

The *back-propagation rule* is undoubtedly one of the most popular models. Several variations have been developed simultaneously by different researchers [RUM86, LEC85, PAR85]. This algorithm is essentially a generalization of the delta rule presented in section 1.7 to multi-layer feed-forward networks (see section 1.4.3). In evaluation mode, the network's behavior is described by equation (1.18) or (1.21). On the last layer, the error signal vector $\vec{\delta}^{[L]}$ is computed identically to that of the delta rule:

$$\vec{\delta}^{[L]}(t) = \left(\vec{\mathbf{d}}(t) - \vec{\mathbf{y}}^{[L]}(t)\right) \circ \sigma'\left(\vec{\mathbf{p}}^{*[L]}(t)\right) \tag{1.33}$$

This vector is then back-propagated through the layers to compute the error signal for all neurons. Unlike the delta rule, it is not possible to describe the case of networks without thresholds and that with thresholds treated as weights in a uniform fashion (using the $*$-notation). For the sake of simplicity, the algorithm is presented in this section without thresholds. Two ways of dealing with thresholds are proposed in appendix A.3. The error signal vector is computed as:

$$\vec{\delta}^{[k]}(t) = \left(\mathbf{w}^{[k+1]^{\mathrm{T}}}(t) \cdot \vec{\delta}^{[k+1]}(t)\right) \circ \sigma'\left(\vec{\mathbf{p}}^{[k]}(t)\right) \qquad \text{for } k = 1,2,\ldots,L-1 \tag{1.34}$$

By defining $\vec{\mathbf{y}}^{[0]}(t) = \vec{\mathbf{x}}(t)$, the weights can then be updated as for the delta rule:

$$\mathbf{w}^{[k]}(t+1) = \mathbf{w}^{[k]}(t) + \alpha(t) \cdot \vec{\delta}^{[k]}(t) \cdot \vec{\mathbf{y}}^{[k-1]^{\mathrm{T}}}(t) \qquad \text{for } k = 1,2,\ldots,L \tag{1.35}$$

1.8.1 The back-propagation rule with epoch updating

The back-propagation rule can be extended to epoch updating in a similar way as the delta rule. Equations (1.33), (1.34), and (1.35) then become:

$$\vec{\delta}^{[L]}(t,s) = \left(\vec{\mathbf{d}}(s) - \vec{\mathbf{y}}^{[L]}(t,s)\right) \circ \sigma'\left(\vec{\mathbf{p}}^{[L]}(t,s)\right) \tag{1.36}$$

$$\vec{\delta}^{[k]}(t,s) = \left(\mathbf{w}^{[k+1]^{\mathrm{T}}}(t) \cdot \vec{\delta}^{[k+1]}(t,s)\right) \circ \sigma'\left(\vec{\mathbf{p}}^{[k]}(t,s)\right) \qquad \text{for } k = 1,2,\ldots,L-1 \tag{1.37}$$

$$\mathbf{w}^{[k]}(t+1) = \mathbf{w}^{[k]}(t) + \alpha(t) \cdot \sum_{s \in E(t)} \vec{\delta}^{[k]}(t,s) \cdot \vec{\mathbf{y}}^{[k-1]^{\mathrm{T}}}(t,s) \qquad \text{for } k = 1,2,\ldots,L \tag{1.38}$$

with $\vec{\mathbf{y}}^{[0]}(t,s) = \vec{\mathbf{x}}(s)$.

1.9 The Hopfield model

The *Hopfield model* [HOP82] is based on a type I fully-connected recurrent network, described in section 1.4.2. There are as many inputs as neurons (i.e., $n = m$), and the input network is a direct correspondence between them (i.e., $\mathbf{w}(t) = \mathbf{I}$, $\vec{\theta}(t) = \vec{\mathbf{0}}$, and $\sigma(v) = v$). Therefore, the behavior of this network is described by equation (1.11) with $\vec{\theta}^{\,\mathrm{rec}}(t) = \vec{\mathbf{0}}$ and $\vec{\mathbf{y}}(t,s,0) = \vec{\mathbf{x}}(s)$. Though this model has been first proposed for binary neurons—that is, whose activation function σ^{rec} is a hard limiter—it has then been extended to neurons with continuous outputs [HOP84].

This network is intended for pattern recognition and to be used as an associative memory. Once a pattern is presented as input, the network is expected to converge to the nearest stable pattern (if there is no limit cycle). The aim of the learning process is to train the network so that the prototypes $\vec{\mathbf{x}}$ become stable patterns. It has been proved [HOP82] that this is only possible if the weight matrix $\mathbf{w}^{\mathrm{rec}}$ is symmetric with non-negative diagonal elements. The learning rule proposed by J. Hopfield is the following:

$$w_{i,i}^{\mathrm{rec}} = 0 \tag{1.39}$$

$$w_{i,j}^{\mathrm{rec}} = \sum_{s=1}^{S} (2x_i(s) - 1) \cdot (2x_j(s) - 1) \qquad \text{for } i \neq j \tag{1.40}$$

It can be noticed that this rule is not iterative, unlike those that have already been studied for other models. Several other rules have been proposed, for instance, the Hebbian rule:

$$\mathbf{w}^{\mathrm{rec}} = \frac{1}{m} \cdot \sum_{s=1}^{S} \vec{\mathbf{x}}(s) \cdot \vec{\mathbf{x}}^{\mathrm{T}}(s) \tag{1.41}$$

Computer simulations show that the number of prototypes S that can be reliably learned is 10–20 % of the number of neurons m [DEN86].

1.10 Self-organizing feature maps

A *self-organizing feature map* results from a method proposed by T. Kohonen [KOH82, KOH89] to automatically establish a "map" of a large-dimensionality input space onto an output space usually of much smaller dimensionality. There are many variations of this model, forming two classes. The first one is of biological inspiration and uses a fixed recurrent network (i.e., $\mathbf{w}^{\text{rec}}(t) = \mathbf{w}^{\text{rec}}$) to find the most active neurons. The second class has been proposed to reduce the amount of computation (especially on serial machines). The recurrent network has been replaced by a search for the smallest (largest) element of a set. Though it is the first class that inspired the second one, they are presented in reverse order, because Kohonen himself [KOH89] presents the recurrent networks only as a biological justification for the minimum search. Other authors [LEH93, MII91] have studied the properties of models based on recurrent networks.

1.10.1 The Kohonen model with minimum

This model is based on a single-layer network of distance-based neurons. Each neuron is associated with a weight-matrix row, whose transpose $\mathbf{w}_i^{\mathrm{T}}$ —sometimes called *weight vector*—belongs to the input space. When an input vector or *stimulus* is presented, its distance to the weight vector of each neuron is computed. The Euclidean and Manhattan distances of equations (1.2) and (1.3) are commonly used. The outputs of neurons are then equal to:

$$y_i(t) \;=\; \sigma(p_i(t)) \;=\; \sigma\left(\Delta\left(\mathbf{w}_i^{\mathrm{T}}(t), \bar{\mathbf{x}}(t)\right)\right) \tag{1.42}$$

Though there is usually no activation function (i.e., $\sigma(v) = v$), one has been introduced in equation (1.42) because the finite precision of most implementations imposes some kind of saturation. The closest neuron to the input or *winner* is then searched:

$$y_{i_{\text{win}}(t)}(t) \;=\; \min_i(y_i(t)) \tag{1.43}$$

In the case of the Euclidean distance, an optimized implementation would search for the minimum of the squared distance, and thus would save the time and/or hardware to extract a useless square root.

An arbitrary topological relation is defined between the neurons. Frequently used topologies are linear chains and bi-dimensional meshes, shown in figures 1.5(a) and (b). The aim of the learning process is to arrange or map the neurons in the input space—where they are represented by their weight vectors—so that neighbor neurons are activated by neighbor stimuli. In some applications the feature-map space is wrapped around (i.e., ring or torus). Though meshes can easily be generalized to n dimensions, these topologies are seldom used, because of the difficulty of representation and interpretation. Sometimes, exotic topologies, like the hexagonal network of figure 1.5(c), are also used.

During the learning phase, the weights of the winner and its neighbors are updated in the direction of the stimulus that has activated them:

$$\mathbf{w}_i(t+1) \;=\; \mathbf{w}_i(t) + \alpha(t) \cdot \lambda_{i,i_{\text{win}}(t)}(t) \cdot \left(\bar{\mathbf{x}}^{\mathrm{T}}(t) - \mathbf{w}_i(t)\right) \tag{1.44}$$

where α is the time-decreasing learning coefficient ($0 < \alpha(t) \leq 1$), while λ denote the *neighborhood coefficient* ($0 \leq \lambda_{i,i_{\text{win}}(t)}(t) \leq 1$), which is equal to unity for the winner (i.e., $\lambda_{i_{\text{win}}(t),i_{\text{win}}(t)}(t) = 1$)

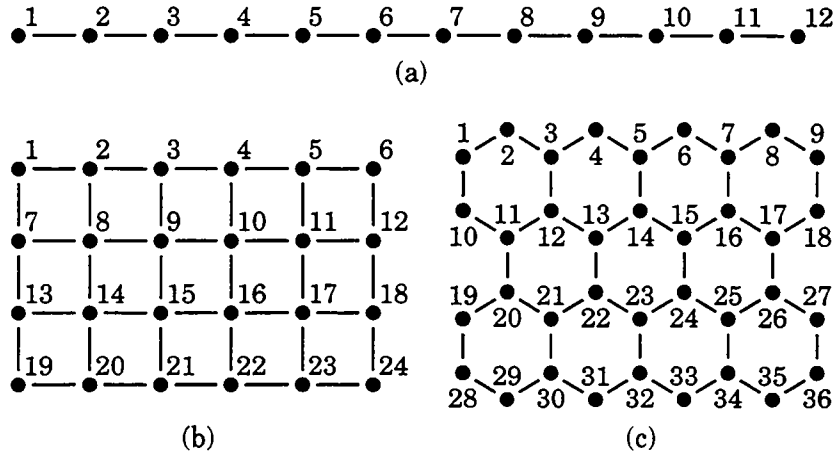


Figure 1.5: *Network topologies for the Kohonen model.* (a) *Linear chain.* (b) *Bi-dimensional mesh.* (c) *Hexagonal network.*

and decreases the further away neurons are in the feature-map space. In the simplest case, this coefficient is equal to 1 for a predefined set of neighbors—for instance, the four or eight closest ones—and to 0 elsewhere. More generally, this coefficient decreases as a function of the distance between a neuron and the winner. This distance, which may be called *output distance*, should not be confused with the *input distance* of equation (1.42) used to select the winner. Again, the output distance can be freely chosen (Euclidean, Manhattan, etc.), and the relation between the distance and the neighborhood coefficient arbitrarily defined. In the framework of this thesis, this coefficient is simply defined by the matrix Λ of all possible values.

Although, in most applications, the learning coefficient α and the neighborhood coefficient λ are merged in a single factor, they appear separately in equation (1.44) to emphasize that both the amplitude of update and the size of the neighborhood decrease with the time.

The Kohonen model with minimum and epoch updating

Similarly to what has been done for the Perceptron, delta, and back-propagation rules, equations (1.42), (1.43) and (1.44) describing the Kohonen model can be extended to epoch updating:

$$y_i(t,s) = \sigma(p_i(t,s)) = \sigma\left(\Delta\left(\mathbf{w}_i^{\mathrm{T}}(t), \vec{\mathbf{x}}(s)\right)\right) \tag{1.45}$$

$$y_{i_{\mathrm{win}}(t,s)}(t,s) = \min_i\left(y_i(t,s)\right) \tag{1.46}$$

$$\mathbf{w}_i(t+1) = \mathbf{w}_i(t) + \alpha(t) \cdot \sum_{s \in E(t)} \lambda_{i,i_{\mathrm{win}}(t,s)}(t) \cdot \left(\vec{\mathbf{x}}^{\mathrm{T}}(s) - \mathbf{w}_i(t)\right) \tag{1.47}$$

The convergence of this algorithm is significantly worse than the on-line version. This can intuitively be understood, because, in the on-line version, neurons are always moved toward the stimuli (in the weight space), while, with epoch updating, they can be moved away.

An alternate approach is to search for the winners by group of prototypes, but to update the weights in a serial way. This mixture of the epoch updating and on-line versions may be referred to as *semi epoch updating*. The winner for each prototype s of the epoch E is still selected according

to equations (1.45) and (1.46), but the weights are updated in an on-line process:

$$\mathbf{w}_i(t,\tau) = \mathbf{w}_i(t,\tau-1) + \alpha(t) \cdot \lambda_{i,i_{win}(t,s(\tau))}(t) \cdot \left(\vec{\mathbf{x}}^T(s(\tau)) - \mathbf{w}_i(t,\tau-1)\right) \quad (1.48)$$

for $\tau = 1,2,\ldots,e$, where e is the epoch length. The initial value of the weight matrix $\mathbf{w}(t,0)$ is defined as $\mathbf{w}(t)$, while the last one $\mathbf{w}(t,e)$ gives the weight matrix for the next epoch $\mathbf{w}(t+1)$.

Like the on-line algorithm, this version guarantees that weights are always moved toward the stimuli. It converges significantly faster than the true epoch updating version, and is much less sensitive to the tuning of the learning and neighborhood coefficient α and λ [VAS94, IEN94]. However, if the number of prototypes per epoch is small compared to the number of neurons, there should be little overlap between the winner neighborhoods of different prototypes, and both versions are expected to have a small influence on the convergence.

The Kohonen model with McCulloch and Pitts neurons

A particular distance that can be used for self-organizing maps is the angle between two vectors in a space of normalized vectors. In this case, only the direction of vectors is considered. It can easily be proved that any vector space can be transformed into such a space by the addition of an extra dimension. Moreover, the transformation is not unique.

If both the stimuli $\vec{\mathbf{x}}$ and the weights $\mathbf{w}_i$ are normalized, their scalar or dot product is equal to the cosine of their angle. Looking for the smallest angle is then equivalent to searching for the largest scalar product. Equations (1.42) and (1.43) can then be replaced by:

$$y_i(t) = \sigma(p_i(t)) = \sigma(\mathbf{w}_i(t) \cdot \vec{\mathbf{x}}(t)) \quad (1.49)$$

$$y_{i_{win}(t)}(t) = \max_i(y_i(t)) \quad (1.50)$$

The interesting property of this method is that the neurons become of the McCulloch and Pitts type, defined in section 1.2.1. The weights must then be updated as:

$$\mathbf{w}_i(t+1) = \frac{\mathbf{w}_i(t) + \alpha(t) \cdot \lambda_{i,i_{win}(t)}(t) \cdot \left(\vec{\mathbf{x}}^T(t) - \mathbf{w}_i(t)\right)}{\left|\mathbf{w}_i(t) + \alpha(t) \cdot \lambda_{i,i_{win}(t)}(t) \cdot \left(\vec{\mathbf{x}}^T(t) - \mathbf{w}_i(t)\right)\right|} \quad (1.51)$$

With normalized weights, the input vectors do not require any normalization, because all cosines are multiplied by the same constant (i.e., the input vector's norm), what does not affect their order relationship.

The main disadvantage of this algorithm, concerning parallel implementations, is that the normalization operation is not local. To overcome this problem T. Kohonen [KOH89] and other authors [DEM92] have suggested to normalize only the input vectors, what can be done only once during pre-processing. In this way, weights tend to get auto-normalized, because they are always updated towards the normalized stimuli. The learning rule is then again described by equation (1.44). The generalization of this algorithm to epoch updating presents no problems.

1.10.2 The Kohonen model with recurrent network

This model, inspired by biological neurons, has been used by T. Kohonen as a starting point to develop the model with minimum described in section 1.10.1. It is composed of McCulloch and Pitts neurons connected to a direct network of plastic weights $\mathbf{w}$ and to a type II recurrent network $\mathbf{w}^{rec}$.

The learning process takes place on the first network, while the second is used to select an *activity bubble* of neurons to be updated. This bubble plays the role of the neighborhood in the model with minimum.

The behavior of these networks is given by equation (1.16) with a constant recurrent network (i.e., $\mathbf{w}^{\text{rec}}(t) = \mathbf{w}^{\text{rec}}$), and should converge to a stable output as defined by equation (1.8). The Kohonen learning rule is expressed by replacing the neighborhood coefficient λ by the outputs y_i^{stable} of the recurrent network:

$$\mathbf{w}_i(t+1) = \frac{\mathbf{w}_i(t) + \alpha(t) \cdot y_i^{\text{stable}}(t) \cdot \left(\vec{\mathbf{x}}^{\text{T}}(t) - \mathbf{w}_i(t)\right)}{\left|\mathbf{w}_i(t) + \alpha(t) \cdot y_i^{\text{stable}}(t) \cdot \left(\vec{\mathbf{x}}^{\text{T}}(t) - \mathbf{w}_i(t)\right)\right|} \tag{1.52}$$

Since this model makes use of McCulloch and Pitts neurons, the discussion of section 1.10.1 on the normalization is also relevant here. Hence, provided that appropriate steps are taken, the denominator of equation (1.52) can be removed.

Type I recurrent networks are sometimes also used. In this case the behavior of the direct network is described by equation (1.10) and that of the recurrent network by equation (1.11).

The Kohonen model with recurrent network and epoch updating

The use of a recurrent network for the Kohonen model can also be extended to epoch or semi epoch updating. In the former case, equation (1.52) becomes:

$$\mathbf{w}_i(t+1) = \frac{\mathbf{w}_i(t) + \alpha(t) \cdot \sum\limits_{s \in E(t)} y_i^{\text{stable}}(t,s) \cdot \left(\vec{\mathbf{x}}^{\text{T}}(s) - \mathbf{w}_i(t)\right)}{\left|\mathbf{w}_i(t) + \alpha(t) \cdot \sum\limits_{s \in E(t)} y_i^{\text{stable}}(t,s) \cdot \left(\vec{\mathbf{x}}^{\text{T}}(s) - \mathbf{w}_i(t)\right)\right|} \tag{1.53}$$

On the other hand, the iterative learning process of the semi epoch updating version is given by:

$$\mathbf{w}_i(t,\tau) = \frac{\mathbf{w}_i(t,\tau-1) + \alpha(t) \cdot y_i^{\text{stable}}(t,s(\tau)) \cdot \left(\vec{\mathbf{x}}^{\text{T}}(s(\tau)) - \mathbf{w}_i(t,\tau-1)\right)}{\left|\mathbf{w}_i(t,\tau-1) + \alpha(t) \cdot y_i^{\text{stable}}(t,s(\tau)) \cdot \left(\vec{\mathbf{x}}^{\text{T}}(s(\tau)) - \mathbf{w}_i(t,\tau-1)\right)\right|} \tag{1.54}$$

1.11 Motivations for using ANNs

In this chapter, ANN models have been presented without providing motivations to use them. Also, no examples of applications have been given. This has been purposely done for conciseness reasons and because the knowledge of ANN algorithms is required to understand this architectural thesis, while knowledge of applications is not. Nevertheless, applications are of capital importance, since they are the main motivations for research in the ANN accelerator domain.

ANNs are used when a process should or can not be modeled by a "conventional" method. Several reasons can be found to adopt such an ANN model instead of another solution. The most obvious one is when the current state of science does not provide any model or when the resulting model would be computationally untractable. ANNs are sometimes also used although "conventional" models are known and daily used, for instance, when such a model is too slow to be implemented in a real-time system. In this case, an ANN can be trained off-line with the "conventional" model and then incorporated in the real-time system. Another example is when

establishing the "conventional" model — which requires some engineering work — is too slow or too expensive. For instance, if an ANN can be trained to recognize bank notes, porting a recognition system based on such an ANN to a new set of bank notes can be done automatically, while a "conventional" method would have to be adapted by hand.

From the description of ANN models presented in this chapter, it can be seen that all these algorithms contain a huge intrinsic parallelism, which make them well suited for dedicated accelerators. While designing the MANTRA I machine proposed in this thesis, a close interaction has been settled with two application teams dealing with power-system security assessment [NIE92] and meteorology [CAT94].

References

[AND88] J. A. Anderson and E. Rosenfeld (eds.). *Neurocomputing: Foundations of Research*. MIT Press, Cambridge, MA (USA), 1988.

[BLO62] H. D. Block. *The Perceptron: A Model for Brain Functioning. I. Reviews of Modern Physics*, 34(1):123–135, January 1962. Reprinted in [AND88].

[CAT94] D. Cattani, D. Engfer, J. Ambühl, E. Amaldi, F. Aviolat, E. Mayoraz, and V. Robert. *AMETIS II–MANTRA B Working Progress Report 93*. MANTRA internal report no. 94/1, EPFL, Lausanne (CH), May 1994.

[DEM92] P. Demartines and F. Blayo. *Kohonen Self-Organizing Maps: Is the Normalization Necessary? Complex Systems*, 6:105–123, 1992.

[DEN86] J. S. Denker (ed.). *Neural Networks for Computing*, AIP Conference Proceedings 151. American Institute of Physics, New York, NY (USA), 1986.

[HEB49] D. O. Hebb. *The Organization of Behavior*. John Wiley & Sons, New York, NY (USA), 1949. Introduction and chapter 4 reprinted in [AND88].

[HER91] J. Hertz, A. Krogh, and R. G. Palmer. *Introduction to the Theory of Neural Computation*, vol. 1 of *Santa Fe Institute Studies in Sciences of Complexity*. Addison-Wesley, Redwood City, CA (USA), 1991.

[HOP82] J. J. Hopfield. *Neural Networks and Physical Systems with Emergent Collective Computational Abilities. Proceedings of the National Academy of Sciences*, 79:2554–2558, April 1982. Reprinted in [AND88].

[HOP84] J. J. Hopfield. *Neurons with Graded Response Have Collective Computational Properties like Those of Two-State Neurons. Proceedings of the National Academy of Sciences*, 81:3088–3092, 1984. Reprinted in [AND88].

[IEN94] P. Ienne. *Experimental Comparison of Batch Kohonen Algorithms*. Internal report, LAMI, EPFL, Lausanne (CH), 1994. To appear.

[KOH82] T. Kohonen. *Self-Organized Formation of Topologically Correct Feature Maps. Biological Cybernetics*, 43:59–69, 1982. Reprinted in [AND88].

[KOH89] T. Kohonen. *Self-Organization and Associative Memory*, vol. 8 of *Springer Series in Information Sciences*. Springer-Verlag, Berlin Heidelberg (D), 3rd edition, 1989.

[LEC85] Y. Le Cun. *A Learning Scheme for Asymmetric Threshold Network.* In *Proceedings of Cognitiva 85*, Paris (F), June 1985.

[LEH93] C. Lehmann. *Réseaux de neurones compétitifs de grandes dimensions pour l'auto-organisation: analyse, synthèse et implantation sur circuits systoliques.* Ph.D. thesis no. 1129, EPFL, Lausanne (CH), 1993.

[MCC43] W. S. McCulloch and W. Pitts. *A Logical Calculus of the Ideas Immanent in Nervous Activity. Bulletin of Mathematical Biophysics*, 5:115–133, 1943. Reprinted in [AND88].

[MII91] R. Miikkulainen. *Self-Organizing Process Based on Lateral Inhibition and Synaptic Resource Redistribution.* In T. Kohonen, K. Mäkisara, O. Simula, and J. Kangas (eds.), *Artificial Neural Networks*, vol. 1, pp. 415–420. North-Holland, Amsterdam (NL), June 1991. Proceedings of ICANN-91.

[MIN69] M. Minsky and S. Papert. *Perceptrons.* MIT Press, Cambridge, MA (USA), 1969. Introduction reprinted in [AND88].

[NIE92] D. Niebur and A. J. Germond. *Unsupervised Neural Net Classification of Power System Static Security States. International Journal on Electrical Power and Energy Systems*, 114(2 & 3):233–242, April & June 1992.

[PAR85] D. Parker. *Learning Logic.* Technical report TR-87, Center for Computational Research in Economics and Management Science, MIT, Cambridge, MA (USA), 1985.

[ROS58] F. Rosenblatt. *The Perceptron: A Probabilistic Model for Information Storage and Organization in the Brain. Psychological Review*, 65:386–408, 1958. Reprinted in [AND88].

[RUM86] D. E. Rumelhart, G. E. Hinton, and R. J. Williams. *Learning Internal Representations by Error Propagation.* In D. E. Rumelhart and J. L. McClelland (eds.), *Parallel Distributed Processing: Explorations in the Microstructures of Cognition,* vol. 1: *Foundations*, chapter 8, pp. 318–362. MIT Press, Cambridge, MA (USA), 1986. Reprinted in [AND88].

[VAS94] N. Vassilas, P. Thiran, and P. Ienne. *A Study of Two Variants of Kohonen's Self-Organizing Feature Maps for Continuous Input and Weight Spaces.* Internal report, National Research Center "Demokritos" & EPFL, Agia Paraskevi (GR) & Lausanne (CH), 1994. To appear.

[WEB91] Merriam-Webster, Springfield, MA (USA). *Webster's Ninth New Collegiate Dictionary*, 1991.

[WID60] B. Widrow and M. E. Hoff. *Adaptive Switching Circuits.* In *IRE WESCON Convention Record*, pp. 96–104, New York, NY (USA), 1960. Reprinted in [AND88].

[WID90] B. Widrow and M. A. Lehr. *30 Years of Adaptive Neural Networks: Perceptron, Madaline, and Backpropagation. Proceedings of the IEEE*, 78(9):1415–1442, September 1990. Invited paper.

Death, taxes, and parallelism: nobody's in favor of any of them, but they're inevitable facts of life.
Paul Schneck, *IEEE Spectrum*, Sep. 1992

2

Architectures for Neural Computation

This chapter presents a brief survey of dedicated systems for ANNs. Since this thesis aims at a digital implementation, no analog solutions are presented. The "digital versus analog" debate has been discussed in the ANN community for a long time. However, one of the two technologies does not have to take over the other. Both can co-exist, since some applications require a digital processing while others can clearly benefit from an analog implementation. Designing digital and analog hardware requires very different skills. ANN users expect digital designers to develop digital tools and machines, and analog engineers to provide analog solutions. In this way, each user should be able to choose the best technology for a given application.

There are several types of dedicated systems for ANNs:

- *Application-specific systems* implement a given model, topology, and set of weights. These circuits are tailored to a unique application. They are often analog.
- *Problem-specific systems* implement a given model and topology. They can be customized by loading a set of weights. The learning process is often executed by an external machine, but some of these systems feature a learning mechanism.
- *Algorithm-specific systems* are dedicated to a given model. Usually, the topology can be freely chosen, and the learning phase is implemented by the system.
- *ANN* or *neural computers*, also referred to as *multi-model accelerators*, are suited to the implementation of a wide class of algorithms.

Since the goal of this thesis is to design an accelerator for the models presented in chapter 1, the emphasis is on the last category. Some algorithm-specific systems are also studied in this chapter.

Due to the huge intrinsic parallelism of ANN algorithms, almost all dedicated systems are parallel. A short introduction on parallel architectures is given in section 2.1, followed by the presentation of a selection of ANN machines in section 2.2.

2.1 Parallel architectures

Since the first prototypes built in the 60's, research on *parallel computers* or *multi-processor architectures* has become a very important field of computer architecture. For the sake of conciseness, a taxonomy is first defined, giving a glimpse of the whole field. Only the few architectures relevant to this thesis are then presented in more detail.

2.1.1 Taxonomy

Many taxonomies of parallel computers [DUC90] have been proposed. Most of them are structured as trees. Since these trees grow often very large, it is more convenient to define a few "orthogonal" criteria, and to classify a given architecture by a set of characteristics (one per criterion). Since the goal of this taxonomy is to provide a short overview of parallel systems, only four of the most important criteria have been selected:

Flynn's taxonomy
This taxonomy [FLY66] is one of the oldest classifications that have proposed for parallel machines. M. J. Flynn divides computers into four categories according to the number of instruction and data streams:

- *Single instruction stream–single data stream (SISD)* architectures.
- *Multiple instruction stream–single data stream (MISD)* architectures.
- *Single instruction stream–multiple data stream (SIMD)* architectures.
- *Multiple instruction stream–multiple data stream (MIMD)* architectures.

Although it is nowadays generally recognized that this taxonomy is too coarse, this division is so important that it has been included in almost all taxonomies proposed since then. Though some authors restrict these classes to their most common model, these definitions are taken here in their most general sense. In particular, for many authors, all processors of an SIMD computer execute simultaneously the same instructions. In this document, Flynn's original definition is adopted, that is, SIMD implies only that the same instruction stream is executed by all processors, but different processor may execute a given instruction at different times.

Parallelism grain
This criterion corresponds to the number of processors. Since the exact number is often of little use, the following classes are usually sufficient:

- *Coarse-grain parallelism*: 2–16 processors.
- *Medium-grain parallelism*: 8–128 processors.
- *Fine-grain parallelism*: 64–2048 processors.
- *Massive parallelism*: $\geq$ 1024 processors.

These classes overlap to reflect that this division is arbitrary and approximative.

Memory Type
This characteristic divides parallel computers into *shared-memory* and *distributed-memory* machines. Some systems feature a mixture of both [PFI85]. Shared-memory computers are

further divided according to the "distance" between processors and memories: *uniform memory access (UMA)* and *non-uniform memory access (NUMA)*. Distributed-memory computers, on the other hand, are sub-divided on whether they have a *global address space* or not. This property means that each of the local memories is mapped on a different address range. Under this paradigm a software layer, which makes a distributed-memory machine appear like shared-memory, may be implemented.

Topology of the interconnection network

Any parallel computer requires an interconnection network [SIE79, SIE86], either to connect the processor to the memories (for shared-memory architectures) or to interconnect the processors (usually for distributed-memory architectures, but sometimes also for shared-memory ones). The simplest topology is the bus. Other common topologies are linear arrays, rings, meshes, tori, and hyper-cubes, that can be generalized as k-ary n-cubes [DAL90]. Finally, trees (fat trees), cross-bars, X-nets, multi-stage networks (e.g., the omega network), and hierarchical structures are also used in many systems. Some machines make use of multiple networks, either hierarchically or in parallel.

Many other criteria could be added, for instance, the communication mode of the interconnection network [KUN89] or the cache coherence mechanism (for shared-memory computers with caches) [STE90].

2.1.2 Systolic arrays

The idea of *systolic arrays* has been introduced in the late 70's by H. T. Kung and C. E. Leiserson [KUN79].[1] These arrays provide a large computing power for matrix and/or vector applications, by means of parallel systems well suited to VLSI. H. T. Kung published a dozen papers on the subject. Among them, a good didactic introduction can be found in "Computer" [KUN82A].

A systolic array is a network of *cells* or *processing elements (PEs)* with connections restricted to direct neighbors. A continuous stream of data flows through the array. The term "systolic" has been chosen by analogy with the circulatory blood system. H. T. Kung [KUN82A] explains it as follows:

> *In a systolic system, data flows from the computer memory in a rhythmic fashion, passing through many processing elements before it returns to memory, much as blood circulates to and from the heart.*

As it is often the case in computer science, there is no commonly accepted definition of systolic arrays. In a book on the subject [MOR92A], J. H. Moreno and T. Lang write:

> *In spite of their popularity, there is no single formal definition of systolic arrays that is widely accepted; moreover, a variety of features (not necessarily compatible) have been considered key aspects in defining this type of structure.*

Although H. T. Kung omitted to give this formal definition, the following is a summary of important characteristics of systolic arrays, as found in his work:

[1] A slightly modified version of this paper has been included by C. A. Mead and L. A. Conway in their book [KUN80]. Strangely, the term "systolic array" has been removed.

Modularity and regularity
The array is made of identical PEs, interconnected according to a regular topology.

Local communications
Each PE is only connected to its nearest neighbors. When data are transmitted between distant PEs, they travel through all intermediate PEs. There is also no broadcast mechanism.

Equal computation and communication rates
Each PE contains only the necessary registers to store the operands (or the result) of the current computation. A transfer takes place on each communication port for each operation. Therefore, the computation and communication rates are equal.

Synchronism
Computation and communication are synchronized on a global clock. Therefore, very efficient and well-mastered design techniques for sequential systems can be used. However, the distribution of a global high-speed clock is one of the toughest problem faced by designers.

Input/output on the array's edges
Read and write accesses to memory take place at the extremities of linear arrays or on the edges of bi-dimensional arrays.

This definition is nearly equivalent to that of S. Y. Kung (not to be confused with H. T. Kung) in his book on processor arrays [KUN88B].

J. H. Moreno and T. Lang [MOR92A] introduce the concept of *systolic-type arrays*, and define systolic arrays as a sub-class. Their definition is somewhat more restrictive than the one above, since the following characteristics have been added:

1. Structures are restricted to linear or bi-dimensional topologies.
2. Connections are limited to at most 4 neighbors.
3. Connections are uni-directional.

In the author's opinion, these conditions are too restrictive to be adopted. Though point 1 is verified in practice — because connections must also be local at the implementation level — some provision should be made for tri-dimensional structures, even if the technology is not yet ready. On the other hand, points 2 and 3 should be rejected because hexagonal array (i.e., 6 neighbors) and bi-directional data streams have already been proposed in the first paper [KUN79]. The GENES IV circuit presented in chapter 4 has bi-directional links, but the counter-flow path is computationally passive. However, this is not always the case [CHU92].

Referring to the taxonomy of section 2.1.1, systolic array are shared-memory systems, with a parallelism ranging from fine-grain to massively parallel. The most common network topologies are: linear arrays, meshes, and hexagonal arrays. In the original idea, each PE executes a unique hard-wired operation. Hence, there are no instructions, and these systems can hardly be classified under Flynn's taxonomy. It is even debatable whether this type of dedicated hardware should be considered as a computer. These systems being very restrictive, arrays with PEs capable of executing several operations have been proposed, with either a single or multiple instruction streams. These models are studied in the next section.

2.1.3 Programmable systolic arrays

The classical systolic arrays presented in section 2.1.2 are hard-wired systems dedicated to a single application. In many cases, some additional flexibility is required. Different types of *programmable systolic arrays* have been proposed. In some of these systems, the interconnection network is programmable, but in most architectures, the programmability is restricted to the PEs. A review of this type of machine can be found in "Computer" [JOH93].

In the simplest programmable systolic arrays, all PEs execute simultaneously the same operation in an SIMD fashion. In this model, the same operation is usually executed for a "long period" and is then reconfigured to perform another function on some other data. These arrays may be considered equivalent to several classical systolic arrays implemented on the same PEs, and are referred to as *configurable systolic arrays*. The GENES HH8 and VM16 systems, presented in sections 3.3 and 3.4, belong to this category.

A further improvement is to broadcast a different instruction on each column and a different selector — that is, a binary value used to enable or disable the execution of instructions — on each row. These arrays are called *instruction broadcasting arrays (IBAs)* by M. Kunde *et al.* [KUN88A]. There is more than one instruction stream, and these machines are classified as MIMD. Since each column forms an SIMD system, it is better to speak about MIMD sets of SIMD clusters.

H.-W. Lang proposed the idea of *instruction systolic arrays (ISAs)* [LAN86, LAN90]. These arrays are identical to IBAs, except that instructions and selectors flow systolically through the columns and rows instead of being broadcast. This model avoids non-local communication (i.e., the instruction and selector broadcast), and, thanks to its pipelined nature, may have a better throughput than IBAs.

The GENES IV array, presented in chapter 4, implements a mixture of configurable systolic arrays and ISAs. As the former category, it is an SIMD system since there is a single instruction stream, but instructions flow systolically through the array instead of being broadcast.

Systolic arrays should not be confused with parallel computers using *systolic communication*, sometimes named by the confusing term of *systolic computers*. These are MIMD machines with asynchronous processors communicating through point-to-point channels. Processors send and receive raw-data words (i.e., words are not grouped into packets and no message header is added) by writing or reading them through dedicated registers called *gates*. The links consist of blocking FIFOs (or in the simplest case of registers, i.e., FIFOs of depth 1). When a processor tries to write into a full FIFO or read from an empty one, it blocks and thus achieves implicit synchronization. This paradigm has been introduced with the Warp [ANN87] and iWarp [BOR88, BOR90] machines in an attempt to exploit the advantages of both systolic arrays and MIMD computers.

2.1.4 Wavefront array processors

As stated in section 2.1.2, the main drawback of systolic arrays is their synchronism. Distributing a global high-speed clock on a chip or a board may prove very difficult, and clock skews may severely limit the maximum operating frequency. To overcome this problem, S.-Y. Kung *et al.* have proposed the *wavefront array processors (WAPs)* [KUN82B]. These machines are similar to systolic arrays, except that PEs are asynchronous and data transmission between neighbor PEs is enabled by a hand-shake protocol. Although originally proposed for MIMD machines, any type of systolic array without broadcasting (i.e., classical systolic arrays, ISAs, SIMD arrays with systolic instruction flow, etc.) has a wavefront counterpart.

There are no restrictions on the type of hand-shake protocol that can be used for communication, and the PE technology may also be freely chosen. Each PE can have its own clock when synchronous logic is to be used, but it is also possible to use asynchronous logic, which can be either *self-timed* [SEI80, SUT89] or *delay-insensitive* [DEG93].

The frontier between WAPs and multi-processor systems using systolic communication, briefly mentioned in section 2.1.3, is not very clear. These models have been developed by different teams, and no formal classification has been defined. The difference is more quantitative than qualitative. Parallel computers using systolic communication are usually coarse or medium-grain parallel machines with powerful processors, while WAPs are fine-grain or massively parallel systems with simple PEs. As a consequence the synchronization grain is finer in WAPs, and communication is usually based on registers, while systolic communication is often implemented by FIFOs.

2.2 ANN architectures

The traditional approach to the design of ANN accelerators is to decompose the target algorithm(s) into basic operations, and to tailor the hardware to these operations [RAM91A, RAM91B, LEH93]. There are many ways to perform this decomposition, reflecting the variety of commercial and research computers. Not surprisingly, all these architectures optimize matrix and vector computation, in particular matrix-vector or matrix-matrix products. At the element level, these products are composed of a large number of multiply-accumulate operations with relatively few dependencies. This huge parallelism explains why all ANN machines implement this part of the computation by parallel systems and/or pipelines.

The rest of the computation (in particular the activation function) is performed in several ways. In some systems, this task is performed by the parallel neural architecture, other systems provide some dedicated hardware for this purpose (look-up tables, shifters, etc.), and finally the remaining machines leave this computation to a conventional SISD processor.

Another constraint is given by Amdahl's law [AMD67]. This law sets an upper bound on the speed-up that can be achieved by the parallelization (or performance improvement by any other method) of an application. If f_{par} is the fraction of a program that should be parallelized (expressed in execution time before parallelization) and $S_{\text{up, par}}$ is the speed-up achieved over this part of code, than the overall speed-up is equal to:

$$S_{\text{up}} = \frac{1}{(1 - f_{\text{par}}) + \frac{f_{\text{par}}}{S_{\text{up, par}}}} \tag{2.1}$$

This quantity has the following upper bound:

$$S_{\text{up, max}} = \frac{1}{1 - f_{\text{par}}} \tag{2.2}$$

Since almost all ANN applications require data pre- and post-processing, and since the purpose of a neural computer is to reduce the execution time of the neural part of an application, there is a limit on the global achievable speed-up, as stated by J. Wawrzynek *et al.* [WAW93]:

> *Successful neural system design must be sensitive to what could be considered a "neural" corollary to Amdahl's Law: A connectionist accelerator can be best speed up an*

application [sic] *by a factor of 1/(fraction of nonconnectionist computation).*

This implies that an efficient ANN machine should either be able to run the non-neural part of the application, or be tightly connected to a system capable of speeding up this code.

The rest of this section is devoted to the presentation of several examples of neural computers. A good review of this class of machines can be found in a technical report by P. Ienne [IEN93].

2.2.1 Microprocessor-based general-purpose accelerators

This section presents four accelerators that have been developed either to run ANN algorithms or with ANNs as a possible application. They are all based on commercial microprocessors or ALUs, and therefore can efficiently run a wide class of algorithms having the same compute-intensive characteristics as neural networks, including scientific computation and signal or image processing. All these systems use floating-point numbers.

The SPRINT computer

The *SPRINT* machine[2] [DEG87], developed at Lawrence Livermore National Laboratory, is a parallel computer composed of 64 IMS T800 Transputers. This machine can run either in MIMD or in SIMD mode. The processors are connected by a reconfigurable network that can emulate square and triangular meshes, binary trees, shuffle-exchange networks, etc. Although not specifically designed to run ANN algorithms, this architecture has been found well suited for this purpose [DEG89].

The RAP computer

The *Ring Array Processor (RAP)* [MOR90, MOR92B] is a multi-processor machine developed at the International Computer Science Institute (ICSI) to run ANN simulations. Each PE is based on the TMS320C30 *digital signal processor (DSP)* from Texas Instruments, featuring a peak performance of 32 MFLOPS. Each DSP is connected to a 256-Kbyte static RAM and a dynamic RAM of 4 or 16 Mbyte, as well as to a custom ring controller implemented in PGAs. The ring is composed of up to 64 PEs, four of them being housed on a VMEbus board. Each node can be accessed by the host (SUN workstation) through a VMEbus interface.

The MUSIC computer

The *MUSIC* machine[3] [MUL92] is a DSP ring, built at the Swiss Federal Institute of Technology at Zurich (ETH). In many respects this system is very similar to the RAP computer. It is based on the MC96002 microprocessor from Motorola. Three PEs, with a 1-Mbyte static RAM and an 8-Mbyte Video RAM each, are implemented on a single board, together with the communication logic. Each board also contains an IMS T800 Transputer used to communicate with the host system. This additional microprocessor has the same functional purpose as the VMEbus interface of the RAP machine.

[2] SPRINT is the acronym of *Systolic Processor with a Reconfigurable Interconnection Network of Transputers.*

[3] MUSIC is the acronym of *Multiprocessor System with Intelligent Communication.*

The SMART computer

The *SMART* machine[4] [LAW93, FIL93] is a pipelined accelerator dedicated to the processing of sparse matrices, under development at the *Institut National Polytechnique de Grenoble* (INPG). Each pipeline stage or PE is based on a commercial *floating-point unit (FPU)* with its local RAM and dedicated communication logic. The controller—an SISD system based on a SPARC microprocessor—dispatches instructions to the first PE, which propagates them to the second, and so on. The PEs are interconnected both by a ring—which is parallel to the instruction stream, except for the wrap-around link—and a global bus, which is also connected to the controller. The machine is connected to the external world through a VMEbus interface.

2.2.2 ASIC-based general-purpose accelerators

This section presents two accelerators that have similar characteristics as the systems presented in section 2.2.1, except that they are based on custom PEs. The first machine processes only integers, while the second features floating-point arithmetic.

The CNAPS computer

The *Connected Network of Adaptive Processors* or *CNAPS* machine [HAM91, ADA93] is a commercial computation server developed at Adaptive Solution. It is dedicated to neural networks, pattern recognition, and signal processing. Its SIMD architecture is composed of 64, 128, 256, or 512 PEs connected by three global busses (8-bit input bus, 8-bit output bus, and 31-bit instruction bus) and by point-to-point links (bi-directional, 2 bits per direction) in a linear array. Each PE is a kind of 16-bit integer DSP with independent shifter/logic unit, adder, and multiplier. Some intermediate results are represented on 24 or 32 bits. Each PE has 32 registers and a 4-Kbyte internal RAM.

A set of 80 PEs is integrated on the very large *CNAPS-1064* chip[5] (11 million transistors, $26.2 \times 27.5\,\mathrm{mm}^2$), which is reconfigured into a 64-PE array to avoid faulty units [GRI91, INO91]. Four of these chips are packaged on a VMEbus board together with the SIMD controller, made of a sequencer and a 16-Mbyte RAM. This accelerator is accessed by Ethernet through an MC68030-based system.

This architecture makes the machine rather flexible. Moreover, it is easy to program, thanks to a dialect of the C language, featuring primitives for parallel operations and fixed-point arithmetic, and a library of parallel sub-routines. The CNAPS computer is very efficient as long as the communication is restricted to direct-neighbor and global-broadcast messages. However, the performance drops dramatically if non-neighbor communication must be implemented. In particular, the machine becomes virtually useless if the problem is too large to fit in the 4-Mbyte/PE internal RAMs and PEs have to swap data with the sequencer memory.

The SNAP computer

The *SIMD Neurocomputer Array Processor (SNAP)* [MEA91, MEA93] is a medium-grain parallel system built by HNC. Though called "neurocomputer," this machine is a ring of processors that can also be used for many other applications like signal and image processing, pattern recognition,

[4] SMART is the acronym of *Sparse Matrix and Recursive Transform*.

[5] Also known as Inova *N64000*.

etc. Each PE consists of a 32-bit floating-point multiplier and a 32-bit integer and floating-point ALU. Its peak performance is 40 MFLOPS. It is connected to an external 512-Kbyte RAM. Four PEs are integrated on a custom VLSI chip, and four of these chips are connected on a SNAP board. Since the machine consists of one to four boards, it has from 16 to 64 PEs. The processors are interconnected by a ring and a global bus, making inter-PE communication and accesses to a global memory possible. All data paths (i.e., the local-memory busses, the global bus, and the inter-PE ring) are 32 bits wide.

The SIMD sequencer follows the *very long instruction word (VLIW)* philosophy with 96-bit instruction words. It has a memory of 8 or 32 Kword. The SNAP machine is accessed through a *Balboa 860* system, an accelerator board also built by HNC and based on the i860 microprocessor from Intel. This system can in turn be connected to the external world through a VMEbus interface.

2.2.3 SIMD neural accelerators

This section presents a few SIMD systems that are dedicated to one or several specific ANN models. They are not suited to run non-neural computation.

The TNP machine

The *Toroidal Neural Processor* or *TNP* machine [JON91A, JON91B] is an ANN computer developed at the University of Nottingham. Its architecture is an SIMD ring of custom PEs. The communication is performed through dedicated input and output registers connected by 16-bit channels. Each PE is also attached to its own weight memory. Beside the communication logic, each PE is made of an adder, a multiplier, a comparator, a random generator, and 18 general-purpose registers. Arithmetic operations are performed on 16-bit fixed-point numbers. There are two modes (only relevant for the multiplier), with 7 or 11 bits after the binary point. The PEs have also a tag register, and execute a given operation only if the instruction's tag matches their own. This is a generalization of the enable bit found on most SIMD machines.

The TNP controller is another custom processor [HUN91]. Simulations have been used to design the best instruction set for this unit. A delay-insensitive version of the TNP machine is being designed in collaboration with the Technical University of Denmark [NIE91].

Hitachi machines

The company Hitachi is very active in the domain of neural computers, and had several projects in the past few years. A first direction of research was on *wafer scale integration (WSI)* [FUJ93]. On the first prototype [YAS90], 576 PEs have been integrated on a 5-inch wafer. These PEs are very simple, since this system can only compute weighted sums (no learning capabilities). The PEs are connected by three global busses (9-bit input bus, 9-bit output bus, and 10-bit address bus). The 8-bit weights are stored in the 64 registers of each PE, while the 9-bit data are broadcast on the bus. Each weight register contains also a 10-bit address field. This makes it possible to keep only the 64 largest weights out of set of 1024 possible ones, the others being neglected.

A second system has then been designed [YAS91] to support learning according to the back-propagation rule (see section 1.8). In this implementation, weight matrices are stored twice in the array, once row-wise for the feed-forward phase and once column-wise for the back-propagation phase (where the transpose matrices are used). Both copies are updated. The PE architecture has

been modified accordingly, and the weight size has been extended to 16 bits. As a consequence, only 144 PEs could be integrated on a wafer. Eight wafers have been used to build a 1152-PE machine. Another version of the wafer with 288 PEs has then been proposed [FUJ93].

The originality of these projects lies much more in the WSI technology and the reconfiguration techniques than in the rather conventional architecture. However, it seems that this technology is not mature for commercial applications, since the computer [SAT93], to which these projects finally led, is built of conventional VLSI chips.

Another direction of research of Hitachi is on systems making a heavy use of dynamic RAM circuits [WAT93A, WAT93B]. This project is based on the observation that most ANN computers use internal memories or static RAMs, offering a huge computing power as long as the application fits into this storage, but are often limited to relatively small problems. In this sense, their approach is similar to the SYNAPSE-1 machine, described in section 2.2.4.

2.2.4 Neural systolic arrays

This section presents four systolic arrays dedicated to some specific ANN models. The first is a linear array, the second is a collection of linear arrays, and the last two are bi-dimensional arrays.

Weinfeld's systolic array

M. Weinfeld has proposed a linear systolic array dedicated to the Hopfield network (see section 1.9) [WEI89], which has been integrated in a joint project of the *École Polytechnique* and the *École Supérieure de Physique et de Chimie Industrielles de la Ville de Paris* [JOH92]. Each PE implements a neuron and can store up to 64 weights in a circular buffer. Since the Hopfield model is a recurrent network of binary neurons, both the inputs and outputs are represented on 1 bit. Therefore, the weighted sum is simply a conditional accumulation of the 9-bit weights. The accumulator stores the partial sum on 12 bits, saturating this value in case of overflow. A mechanism has also been implemented to check the convergence, by comparing the new network state with the previous one. Thanks to the very simples PEs, 64 of them could be implemented on a custom integrated circuit. This chip is used to accelerate an ANN simulator [GAS92A, GAS92B].

The SYNAPSE-1 systolic array

The *SYNAPSE-1* machine[6] [RAM91C, RAM92] is a neural computer designed at Siemens, and based on the custom *MA16* systolic circuit [BEI91A, BEI91B]. A preliminary study aimed at unifying different ANN models using the mathematical concept of Hamiltonian [RAM91A, RAM91B]. This analysis helped the hardware designers to define the computational requirements and hence to design the machine.

The PEs, called *elementary chains*, are designed to process sub-matrices of 4×4 elements. Each of them consists of a pipeline of three arithmetic units. The first stage is a matrix unit, which can multiply two sub-matrices in 16 clock cycles. It is composed of 4 multipliers and 4 adders. Other operations, like addition, subtraction, and bypass of sub-matrices are also available. This unit is at the origin of the machine's very high performance. The second stage is a

[6] SYNAPSE is the acronym of *Synthesis of Neural Algorithms on a Parallel Systolic Engine.*

scalar unit that successively executes, in 16 clock cycles, a given operation on all elements of a sub-matrix. Possible operations are the absolute value, the square, and a row or column-wise constant scaling. Finally, the third stage is also a scalar unit, which implements reduction operations (i.e., addition, subtraction, minimum, and maximum). These operations are either performed between the incoming matrix and an "accumulation" matrix, or across rows of the incoming matrix. Since this unit is not orthogonal, not all combinations are possible. Shift and saturation operations are also available.

This circuit accepts 16-bit operands and computes 48-bit results. A linear array of 4 elementary chains, connected by 48-bit channels, is integrated in an MA16 chip. Thanks to interconnection ports of the same width, these circuits can be cascaded to produce arbitrary long arrays.

The SYNAPSE-1 system consists of two rings of four MA16 circuits each (i.e., 2×16 elementary chains). Though this is a bi-dimensional topology, it is more appropriate to speak of a collection of rings, because there are no systolic interconnections between MA16 chips of the same columns. However, these circuits are connected to the same weight memories. A very powerful feature of the machine is that the PEs are directly interfaced to dynamic RAMs. This strategy has been chosen because many ANN computers are limited by their small storage capacity (as, for instance, the CNAPS machine). This makes it possible to connect the machine to a huge amount of memory. The weight memory consists of four banks of 4 to 128 Mbyte plus parity. The inputs and outputs are stored in two banks of 8 or 32 Mbyte, and each of the eight MA16 circuits has a 4-Mbyte RAM for temporary results.

Besides the MA16 chips, there is a fifth node on each ring, which is an arithmetic unit. This system, whose purpose is to compute the activation function and other auxiliary tasks, is based on the MC68040 microprocessor from Motorola. This processor is mainly used to control a barrel shifter, an adder, a multiplier, and a look-up table placed on the data path. It can also perform the computation itself by disconnecting the data stream from these units. The SIMD controller is made of a similar MC68040-based system and a sequencer. All MC68040 microprocessors have a dynamic RAM of 8 or 32 Mbyte and are connected to VMEbus interfaces. A memory of 256 Kbyte or 2 Mbyte is attached to the sequencer to store the 96-bit micro-instructions.

The SYNAPSE-1 machine is probably one of the most interesting ANN computer, for its novel architecture, as well as its impressive computation power (peak performance of 5.1 GCPS) and storage capacity. However, its complex programmer's model may confine it to a very restricted niche of users.

The Aplysie systolic array

The *Aplysie* system [BLA89A, BLA89B] is a square systolic array dedicated to the Hopfield model (see section 1.9). This system is a direct ancestor of the GENES IV array presented in chapter 4 and evaluated in this thesis. The GENES architecture (see chapter 3) proposed by F. Blayo is a generalization of the work he did with P. Hurat on this system. This array is very close to the GENES HN8 circuit (see section 3.2), except that the inputs are 1 bit wide. In particular this system has no on-chip learning capabilities. A custom VLSI chip containing a matrix of 16×16 cells has been built.

Chung's systolic array

J.-H. Chung *et al.* [CHU92] proposed a systolic array dedicated to the back-propagation rule. In many respects this system is similar to the GENES IV array (see chapter 4), because it is also inspired from the early work of F. Blayo and C. Lehmann on the GENES architecture. However, while multi-layer networks in the GENES IV array are implemented by feeding the output of a layer back in the array as the input of the next one, in this implementation a different array is required for each layer. Another fundamental difference is that the GENES IV circuit can execute multiple instructions. Hence, different models or different phases of an algorithm can be implemented on the same array, while J.-H. Chung *et al.* designed two different systolic arrays for the feed-forward and back-propagation phases of the target algorithm, that have be integrated in the same cells. As a consequence, the second part of the computation takes place on the counter-flow path through the same arrays.

This system is only suitable for algorithm-specific machines, because not only the algorithm (i.e., the back-propagation rule) is hard-wired, but the number of layers and the number of neurons per layer are also fixed at design time. Therefore, with a comparable technology, this solution would offer more computing power than the GENES IV array, at the price of a larger amount of hardware and lack of versatility.

2.2.5 Neural processors

This section presents a coprocessor and a processor dedicated to ANNs.

The Lneuro circuit

The *Lneuro 1.0* (contraction of *l*earning *neuro*) VLSI circuit [DUR90, MAU92] is a neural coprocessor, designed at the *Laboratoire d'Électronique Philips* (LEP). It is composed of an array of bit-slice multipliers, an adder tree, and an ALU/accumulator capable of additions, subtractions, and shift operations. The multiplication can be implemented as an iterative process, thanks to a combination of bit-slice multipliers (i.e., a set of AND gates) and an adder tree. The weights are retrieved from a 1024-byte internal memory, while the inputs come from two 16-byte buffers. The weights and inputs are usually 8 bits wide, but a reconfiguration technique makes it possible to use smaller weights and hence to increase the number of implementable neurons. Similarly, the weight size can also be doubled during the learning phase.

In the machine built at LEP, each node consists of four Lneuro 1.0 chips connected to the bus of an IMS T800 Transputer (with its local RAM). Four such nodes are then connected using the Transputer's built-in communication channels. Two additional Transputers are used for synchronization, routing, and input/output. The simulation of ANNs takes place on the Transputer network, with the Transputers dispatching compute-intensive part of the code (mainly additions or multiplications of matrices and vectors) to the Lneuro 1.0 circuits, much like a microprocessor dispatches floating-point operations to a coprocessor. An improved version, referred to as *Lneuro 2.0*, is about to be completed.

The SPERT circuit

The *SPERT* circuit[7] [ASA92, WAW93] is an ANN microprocessor developed at ICSI. It combines a RISC-like processing unit and an SIMD array. The processing unit—used for general computation—is composed of an instruction unit, a 32-bit integer ALU (i.e., adder, logic unit, comparator, and shifter), and two independent address units. The SIMD array—designed for neural computation—consists of eight PEs, each of them containing a multiplier, an adder, a shifter, and a limiter. These devices process 32-bit integers, except the multiplier that accepts two operands of 8 and 24 bits producing a 32-bit output. The memory interface features a 128-bit data bus, and addresses up to 16 Mbyte. The instruction format is a 128-bit VLIW, with separate fields for the processing unit, the SIMD array, the memory interface, and the host synchronization unit. An internal instruction cache is available.

2.3 Motivations for designing neural accelerators

Although the list of machines presented in section 2.2 is not exhaustive, it sketches a good overview of the different classes of neural accelerators and gives a description of the most important and most interesting computers. From this list, it can be seen that most systems either offer a good generality but are based on commercial microprocessors and on a conventional architecture (i.e., SPRINT, RAP, MUSIC, and SMART), or are based on custom hardware but are limited to a single or a very restricted number of models (i.e., TNP, Hitachi machines, Weinfeld's array, Aplysie, Chung's array, and Lneuro). Only the CNAPS, SNAP, SYNAPSE-1, and SPERT machines are multi-model ANN accelerators based on dedicated hardware. Moreover, although based on a very advanced technology, the first two computers (i.e., CNAPS and SNAP) have a rather conventional architecture.

As stated in section 1.11, ANN models contain a huge intrinsic parallelism, what make them well suited for massively parallel systems. However, none of the above-cited machines can be considered as massively parallel (except the array proposed by J.-H. Chung *et al.*, which is inspired by the GENES architecture, but has never been built). This clearly shows a field that has been left unexplored. The GENES arrays and the MANTRA I machine presented in the next three chapters aim at studying the used of massively parallel multi-model neural accelerators.

References

[ADA93] Adaptive Solutions, Beaverton, OR (USA). *CNAPS®, vol. 5: Hardware Series*, August 1993.

[AMD67] G. M. Amdahl. *Validity of the Single Processor Approach to Achieving Large Scale Computing Capabilities.* In *Proceedings of AFIPS Spring Joint Computer Conference*, vol. 30, pp. 483–485, Atlantic City, NJ (USA), April 1967. Academic Press.

[ANN87] M. Annaratone, E. Arnould, T. Gross, H. T. Kung, M. Lam, O. Menzilcioglu, and J. A. Webb. *The Warp Computer: Architecture, Implementation, and Performance. IEEE Transactions on Computers*, C-36(12):1523–1538, December 1987.

[7] SPERT is the acronym of *Synthetic Perceptron Testbed*.

[ASA92] K. Asanović, J. Beck, B. E. D. Kingsbury, P. Kohn, N. Morgan, and J. Wawrzynek. *SPERT: A Neuro-Microprocessor.* In J. G. Delgado-Frias and W. R. Moore (eds.), *Proceedings of the 3rd International Workshop on VLSI for Neural Networks and Artificial Intelligence*, Oxford (GB), September 1992.

[BEI91A] J. Beichter, U. Ramacher, and H. Klar. *VLSI Design of a Neural Signal Processor.* In M. Sami and J. Calzadilla-Daguerre (eds.), *Silicon Architectures for Neural Nets*, pp. 245–260. North-Holland, Amsterdam (NL), 1991. Proceedings of the IFIP workshop on silicon architectures for NNs.

[BEI91B] J. Beichter, N. Brüls, E. Meister, U. Ramacher, and H. Klar. *Design of a General-Purpose Neural Signal Processor.* In U. Ramacher, U. Rückert, and J. A. Nossek (eds.), *Proceedings of the 2nd International Conference on Microelectronics for Neural Networks*, pp. 311–315, Munich (D), October 1991. Kyrill and Method Verlag.

[BLA89A] F. Blayo and P. Hurat. *A VLSI Systolic Array Dedicated to Hopfield Neural Network.* In J. G. Delgado-Frias and W. R. Moore (eds.), *VLSI for Artificial Intelligence*, chapter 8.2, pp. 255–264. Kluwer Academic Publishers, Boston, MA (USA), 1989. Proceedings of the 1st workshop on VLSI for AI.

[BLA89B] F. Blayo and P. Hurat. *A Reconfigurable W.S.I. Neural Network.* In E. Swartzlander and J. Brewer (eds.), *Proceedings of the International Conference on Wafer Scale Integration*, pp. 141–150, San Francisco, CA (USA), January 1989. IEEE Computer Society, IEEE Computer Society Press.

[BOR88] S. Borkar, R. Cohn, G. Cox, S. Gleason, T. Gross, H. T. Kung, M. Lam, B. Moore, C. Peterson, J. Pieper, L. Rankin, P. S. Tseng, J. Sutton, J. Urbanski, and J. Webb. *iWarp: An Integrated Solution to High-Speed Parallel Computing.* In *Proceedings of SUPERCOMPUTING '88*, pp. 330–339, Orlando, FL (USA), November 1988.

[BOR90] S. Borkar, R. Cohn, G. Cox, T. Gross, H. T. Kung, M. Lam, M. Levine, B. Moore, W. Moore, C. Peterson, J. Susman, J. Sutton, J. Urbanski, and J. Webb. *Supporting Systolic and Memory Communication in iWarp.* In *Proceedings of the 17th Annual International Symposium on Computer Architecture*, pp. 70–81, Seattle, WA (USA), May 1990. IEEE Computer Society, ACM SIGARCH, IEEE Computer Society Press.

[CHU92] J.-H. Chung, H. Yoon, and S. R. Maeng. *A Systolic Array Exploiting the Inherent Parallelisms of Artificial Neural Networks. Microprocessing and Microprogramming*, 33(3):145–159, May 1992.

[DAL90] W. J. Dally. *Performance Analysis of k-ary n-cube Interconnection Networks. IEEE Transactions on Computers*, 39(6):775–785, June 1990.

[DEG87] A. J. De Groot, E. M. Johansson, J. P. Fitch, C. W. Grant, and S. R. Parker. *SPRINT—The Systolic Processor with a Reconfigurable Interconnection Network of Transputers. IEEE Transactions on Nuclear Science*, NS-34(4):873–877, August 1987.

[DEG89] A. J. De Groot and S. R. Parker. *Systolic Implementation of Neural Networks.* In K. Bromley (ed.), *High Speed Computing II*, vol. 1058, pp. 182–190, Los Angeles, CA (USA), January 1989. SPIE—The International Society for Optical Engineering.

[DEG93] A. De Gloria, P. Faraboschi, and M. Olivieri. *Delay Insensitive Micro-Pipelined Combinational Logic. Microprocessing and Microprogramming*, 36(5):225–241, October 1993.

[DUC90] R. Ducan. *A Survey of Parallel Computer Architectures. Computer*, 23(2):5–16, February 1990. IEEE Computer Society.

[DUR90] M. Duranton and J. A. Sirat. *Learning on VLSI: A General-Purpose Digital Neurochip. Philips Journal of Research*, 45(1):1–16, 1990.

[FIL93] E. Filippi and J. C. Lawson. *A Parallel Implementation of Kohonen's Self-Organizing Maps on the SMART Neurocomputer.* In J. Mira, J. Cabestany, and A. Prieto (eds.), *New Trends in Neural Computation*, vol. 686 of *Lecture Notes in Computer Science*, pp. 388–393. Springer-Verlag, Berlin Heidelberg (D), June 1993. Proceedings of IWANN'93.

[FLY66] M. J. Flynn. *Very High-Speed Computing Systems. Proceedings of the IEEE*, 54(12):1901–1909, December 1966.

[FUJ93] M. Fujita, Y. Kobayashi, K. Shiozawa, T. Takahashi, F. Mizuno, H. Hayakawa, M. Kato, S. Mori, T. Kase, and M. Yamada. *Development and Fabrication of Digital Neural Network WSIs. IEICE Transactions on Electronics*, E76-C(7):1182–1190, July 1993.

[GAS92A] J.-D. Gascuel, E. Delaunay, L. Montoliu, B. Moobed, and M. Weinfeld. *A Software Reconfigurable Multi-Networks Simulator Using a Custom Associative Chip.* In *Proceedings of the International Joint Conference on Neural Networks*, vol. II, pp. 13–18, Baltimore, MD (USA), June 1992. IEEE, INNS.

[GAS92B] J.-D. Gascuel, E. Delaunay, L. Montoliu, B. Moobed, and M. Weinfeld. *A Custom Associative Chip Used as a Building Block for a Software Reconfigurable Multi-Networks Simulator.* In J. G. Delgado-Frias and W. R. Moore (eds.), *Proceedings of the 3rd International Workshop on VLSI for Neural Networks and Artificial Intelligence*, Oxford (GB), September 1992.

[GRI91] M. Griffin, G. Tahara, K. Knorpp, R. Pinkham, and B. Riley. *An 11-Million Transistor Neural Network Execution Engine.* In *IEEE International Solid-State Circuits Conference*, pp. 180–181, 1991.

[HAM91] D. Hammerstrom. *A Highly Parallel Digital Architecture for Neural Network Emulation.* In J. G. Delgado-Frias and W. R. Moore (eds.), *VLSI for Artificial Intelligence and Neural Networks*, chapter 5.1, pp. 357–366. Plenum Press, New York, NY (USA), 1991. Proceedings of the 2nd workshop on VLSI for AI and NNs.

[HUN91] J. Hunter and S. Jones. *The Design of Controllers for Linear Arrays.* In U. Ramacher, U. Rückert, and J. A. Nossek (eds.), *Proceedings of the 2nd International Conference on Microelectronics for Neural Networks*, pp. 197–205, Munich (D), October 1991. Kyrill and Method Verlag.

[IEN93] P. Ienne. *Architectures for Neuro-Computers: Review and Performance Evaluation.* Technical report no. 93/21, DI, EPFL, Lausanne (CH), January 1993.

[INO91] Inova, Santa Clara, CA (USA). *N64000 Digital Neural Network Processor*, February 1991. Preliminary data.

[JOH92] A. Johannet, L. Personnaz, G. Dreyfus, J.-D. Gascuel, and M. Weinfeld. *Specification and Implementation of a Digital Hopfield-Type Associative Memory with On-Chip Training. IEEE Transactions on Neural Networks*, 3(4):529–539, July 1992.

[JOH93] K. T. Johnson, A. R. Hurson, and B. Shirazi. *General-Purpose Systolic Arrays. Computer*, 26(11):20–31, November 1993. IEEE Computer Society.

[JON91A] S. Jones, K. Sammut, C. Nielsen, and J. Staunstrup. *Toroidal Neural Network: Architecture and Processor Granularity Issues.* In U. Ramacher and U. Rückert (eds.), *VLSI Design of Neural Networks*, pp. 229–254. Kluwer Academic Publishers, Boston, MA (USA), 1991. Proceedings of the workshop on microelectronics for NNs.

[JON91B] S. Jones, K. Sammut, and J. Hunter. *Toroidal Neural Network Processor: Architecture, Operation, Performance.* In U. Ramacher, U. Rückert, and J. A. Nossek (eds.), *Proceedings of the 2nd International Conference on Microelectronics for Neural Networks*, pp. 163–169, Munich (D), October 1991. Kyrill and Method Verlag.

[KUN79] H. T. Kung and C. E. Leiserson. *Systolic Arrays (for VLSI).* In I. S. Duff and G. W. Stewart (eds.), *Sparse Matrix Proceedings 1978*, pp. 256–282. Society for Industrial and Applied Mathematics, Philadelphia, PA (USA), 1979.

[KUN80] H. T. Kung and C. E. Leiserson. *Algorithms for VLSI Processor Arrays.* In C. Mead and L. Conway, *Introduction to VLSI Systems*, Addison-Wesley Series in Computer Science, section 8.3, pp. 271–292. Addison-Wesley, Reading, MA (USA), 1980.

[KUN82A] H. T. Kung. *Why Systolic Architectures? Computer*, 15(1):37–46, January 1982. IEEE Computer Society.

[KUN82B] S.-Y. Kung, K. S. Arun, R. J. Gal-Ezer, and D. V. Bhaskar Rao. *Wavefront Array Processors: Language, Architecture, and Applications. IEEE Transactions on Computers*, C-31(11):1054–1066, November 1982.

[KUN88A] M. Kunde, H.-W. Lang, M. Schimmler, H. Schmeck, and H. Schröder. *The Instruction Systolic Array and its Relation to Other Models of Parallel Computers. Parallel Computing*, 7:25–39, 1988.

[KUN88B] S. Y. Kung. *VLSI Array Processors.* Prentice Hall, Englewood Cliffs, NJ (USA), 1988.

[KUN89] H. T. Kung. *Network-Based Multicomputers: Redefining High Performance Computing in the 1990s.* In C. L. Seitz (ed.), *Proceedings of the Decennial Caltech Conference on VLSI*, pp. 49–66. MIT Press, March 1989. Invited paper.

[LAN86] H.-W. Lang. *The Instruction Systolic Array—A Parallel Architecture for VLSI. Integration, the VLSI Journal*, 4(1):65–74, March 1986.

[LAN90] H.-W. Lang. *Das befehlssystolische Prozessorfeld—Architektur und Programmierung.* Ph.D. thesis, Christian-Albrechts-Universität, Kiel (D), April 1990.

[LAW93] J. C. Lawson, N. Maria, and J. Hérault. *SMART: A Neurocomputer Using Sparse Matrices.* In *Proceedings of the EUROMICRO Workshop on Parallel and Distributed Processing*, Gran Canaria (E), January 1993.

[LEH93] C. Lehmann, M. Viredaz, and F. Blayo. *A Generic Systolic Array Building Block for Neural Networks with On-Chip Learning. IEEE Transactions on Neural Networks*, 4(3):400–407, May 1993. Special issue on neural network hardware.

[MAU92] N. Mauduit, M. Duranton, J. Gobert, and J.-A. Sirat. *Lneuro 1.0: A Piece of Hardware LEGO for Building Neural Network Systems. IEEE Transactions on Neural Networks*, 3(3):414–422, May 1992.

[MEA91] R. W. Means and L. Lisenbee. *Extensible Linear Floating Point SIMD Neurocomputer Array Processor.* In *Proceedings of the International Joint Conference on Neural Networks*, vol. I, pp. 587–592, Seattle, WA (USA), July 1991. IEEE, INNS.

[MEA93] R. W. Means and L. Lisenbee. *Floating-Point SIMD Neurocomputer Array Processor.* Internal report, HNC, San Diego, CA (USA), 1993.

[MOR90] N. Morgan, J. Beck, P. Kohn, J. Bilmes, E. Allman, and J. Beer. *The RAP: A Ring Array Processor for Layered Network Calculations.* In S.-Y. Kung, E. E. Swartzlander, J. A. B. Fortes, and K. W. Przytula (eds.), *Proceedings of the International Conference on Application Specific Array Processors*, pp. 296–308, Princeton, NJ (USA), September 1990. IEEE, IEEE Computer Society Press.

[MOR92A] J. H. Moreno and T. Lang. *Matrix Computations on Systolic-Type Arrays.* Kluwer Academic Publishers, Boston, MA (USA), 1992.

[MOR92B] N. Morgan, J. Beck, P. Kohn, J. Bilmes, E. Allman, and J. Beer. *The Ring Array Processor: A Multiprocessing Peripheral for Connectionist Applications. Journal of Parallel and Distributed Computing*, 14(3):248–259, March 1992. Special issue on neural computing on massively parallel processing.

[MUL92] U. A. Müller, B. Bäumle, P. Kohler, A. Gunzinger, and W. Guggenbühl. *Achieving Supercomputer Performance for Neural Net Simulation with an Array of Digital Signal Processors. IEEE Micro*, 12(5):55–65, October 1992.

[NIE91] C. D. Nielsen, J. Staunstrup, and J. R. Jones. *A Delay-Insensitive Neural Network Engine.* In J. G. Delgado-Frias and W. R. Moore (eds.), *VLSI for Artificial Intelligence and Neural Networks*, chapter 5.2, pp. 367–376. Plenum Press, New York, NY (USA), 1991. Proceedings of the 2nd workshop on VLSI for AI and NNs.

[PFI85] G. F. Pfister, W. C. Brantley, D. A. George, S. L. Harvey, W. J. Kleinfelder, K. P. McAuliffe, E. A. Melton, V. A. Norton, and J. Weiss. *The IBM Research Parallel Processor Prototype (RP3): Introduction and Architecture.* In *Proceedings of the 1985 International Conference on Parallel Processing*, pp. 764–771. IEEE Computer Society, ACM, August 1985.

[RAM91A] U. Ramacher and B. Schürmann. *Unified Description of Neural Algorithms for Time-Independent Pattern Recognition.* In U. Ramacher and U. Rückert (eds.), *VLSI Design of*

Neural Networks, pp. 255–270. Kluwer Academic Publishers, Boston, MA (USA), 1991. Proceedings of the workshop on microelectronics for NNs.

[RAM91B] U. Ramacher and P. Nachbar. *Hamiltonian Dynamics of Neural Networks.* In U. Ramacher, U. Rückert, and J. A. Nossek (eds.), *Proceedings of the 2nd International Conference on Microelectronics for Neural Networks*, pp. 95–102, Munich (D), October 1991. Kyrill and Method Verlag.

[RAM91C] U. Ramacher, W. Raab, J. Anlauf, U. Hachmann, and M. Wesseling. *SYNAPSE-X: A General-Purpose Neurocomputer.* In U. Ramacher, U. Rückert, and J. A. Nossek (eds.), *Proceedings of the 2nd International Conference on Microelectronics for Neural Networks*, pp. 401–409, Munich (D), October 1991. Kyrill and Method Verlag.

[RAM92] U. Ramacher. *SYNAPSE—A Neurocomputer That Synthesizes Neural Algorithms on a Parallel Systolic Engine. Journal of Parallel and Distributed Computing*, 14(3):306–318, March 1992. Special issue on neural computing on massively parallel processing.

[SAT93] Y. Sato, K. Shibata, M. Asai, M. Ohki, M. Sugie, T. Sakaguchi, M. Hashimoto, and Y. Kuwabara. *Development of a High-Performance General Purpose Neuro-Computer Composed of 512 Digital Neurons.* In *Proceedings of the International Joint Conference on Neural Networks*, vol. 2, pp. 1967–1970, Nagoya (J), October 1993. IEEE, INNS.

[SEI80] C. L. Seitz. *System Timing.* In C. Mead and L. Conway, *Introduction to VLSI Systems*, Addison-Wesley Series in Computer Science, chapter 7, pp. 218–262. Addison-Wesley, Reading, MA (USA), 1980.

[SIE79] H. J. Siegel. *Interconnection Networks for SIMD Machines. Computer*, 12(6):57–65, June 1979. IEEE Computer Society.

[SIE86] H. J. Siegel, W. Tsun-yuk Hsu, and M. Jeng. *Interconnection Networks: The Multistage Cube, Extra-Stage Cube, and Dynamic Redundancy Networks.* In *New Frontiers in Computer Architecture Conference Proceedings*, March 1986.

[STE90] P. Stenström. *A Survey of Cache Coherence Schemes for Multiprocessors. Computer*, 23(6):12–24, June 1990. IEEE Computer Society.

[SUT89] I. E. Sutherland. *Micropipelines. Communications of the ACM*, 32(6):720–738, June 1989.

[WAT93A] T. Watanabe, K. Kimura, M. Aoki, T. Sakata, and K. Ito. *A Single 1.5-V Digital Chip for a 10^6 Synapse Neural Network. IEEE Transactions on Neural Networks*, 4(3):387–393, May 1993. Special issue on neural network hardware.

[WAT93B] T. Watanabe, M. Aoki, K. Kimura, T. Sakata, and K. Itoh. *The Advantages of a DRAM-Based Digital Architecture for Low-Power, Large-Scale Neuro-Chips. IEICE Transactions on Electronics*, E76-C(7):1206–1214, July 1993.

[WAW93] J. Wawrzynek, K. Asanović, and N. Morgan. *The Design of a Neuro-Microprocessor. IEEE Transactions on Neural Networks*, 4(3):394–399, May 1993. Special issue on neural network hardware.

[WEI89] M. Weinfeld. *A Fully Digital Integrated CMOS Hopfield Network Including the Learning Algorithm.* In J. G. Delgado-Frias and W. R. Moore (eds.), *VLSI for Artificial Intelligence*, chapter 6.2, pp. 169–178. Kluwer Academic Publishers, Boston, MA (USA), 1989. Proceedings of the 1st workshop on VLSI for AI.

[YAS90] M. Yasunaga, N. Masuda, M. Yagyu, M. Asai, M. Yamada, and A. Masaki. *Design, Fabrication and Evaluation of a 5-Inch Wafer Scale Neural Network LSI Composed of 576 Digital Neurons.* In *Proceedings of the International Joint Conference on Neural Networks*, vol. II, pp. 527–535, San Diego, CA (USA), June 1990. IEEE, INNS.

[YAS91] M. Yasunaga, N. Masuda, M. Yagyu, M. Asai, K. Shibata, M. Ooyama, M. Yamada, T. Sakaguchi, and M. Hashimoto. *A Self-Learning Neural Network Composed of 1152 Digital Neurons in Wafer-Scale LSIs.* In *Proceedings of the International Joint Conference on Neural Networks*, vol. 2, pp. 1844–1849, Singapore (SGP), November 1991. IEEE, INNS.

Es soll nicht genügen, dass man Schritte tue, die einst zum Ziele führen, sondern jeder Schritt soll Ziel sein und als Schritt gelten.[1]

Johann Wolfgang von Goethe

3

The GENES Systolic Arrays

In his Ph.D. thesis [BLA90], F. Blayo proposed a systolic array, called *GENES*,[2] dedicated to the implementations of ANNs. He observed that a few basic operations are sufficient to implement the most widely used neural models, and showed that these operations can be efficiently implemented on a systolic array. Different types of linear and bi-dimensional arrays where analyzed, before committing to the architecture presented in this chapter.

Three successive implementations of GENES chips have been integrated by C. Lehmann [LEH93A, LEH93B]. Although, only the first one is used on an accelerator board, all three architectures are presented here. Besides analyzing the advantages and weaknesses of these three versions, this chapter constitutes a didactical presentation of the concepts underlying the GENES architecture, and the start-point for the design of the GENES IV array presented in chapter 4.

3.1 Notations

To emphasize the difference between the mathematical relations defined in chapter 1 and the description of implemented operations, different notations are used. While most mathematical quantities are represented by lower-case letters, capitals are used for implementation variables. To avoid confusion, the corresponding upper-case letter is used when appropriate. For instance, the symbol $\mathbf{W}$ corresponds either to the weight matrix $\mathbf{w}$ or to one of its sub-matrices.

These variables should further be distinguished from the hardware registers holding them. The name of these registers is printed with sans-serif characters. For instance, the W register holds an element W_{ij} of the weight matrix $\mathbf{W}$. Other registers, which do not always hold the same type of data, have names with no correspondence in the ANN models, such as PS or D.

The number of bits of a register is denoted by the symbol N and the register's name as subscript, for instance, N_{W}, N_{PS} or N_{D}.

[1] *It is not enough to take steps which may some day lead to a goal; each step must be itself a goal and a step likewise.*

[2] GENES is the acronym of *Generic Element for Neuro-Emulator Systolic Arrays*.

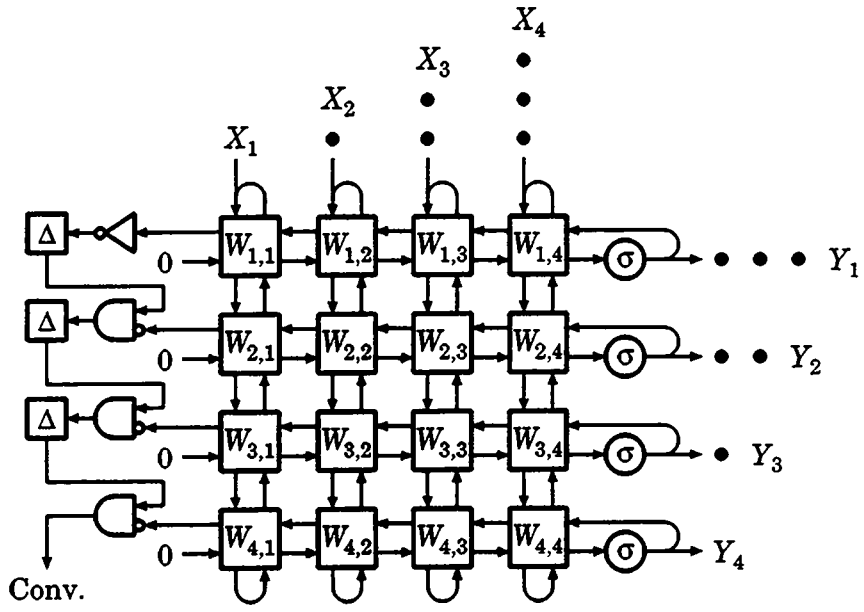


Figure 3.1: *Architecture of the GENES HN8 systolic array (4 × 4 cells).*

3.2 The GENES HN8 circuit

The first VLSI building block for GENES arrays has been integrated by C. Lehmann [LEH91, LEH93A]. It is called *GENES HN8*, which stands for *Hopfield Network, 8 Bits*. As indicated by its name, it was primarily intended for the Hopfield model in recall phase (see section 1.9), no special mechanism for learning being provided. Since this array implements a fully-connected recurrent network of McCulloch and Pitts neurons (see section 1.4.2), it can also be used for other purposes. For instance, C. Lehmann used it as recurrent network for the Kohonen model (see section 1.10.2). It is shown below that the Adaline learning rule in batch mode can also be implemented.

3.2.1 Architecture

Figure 3.1 shows a systolic array based on GENES HN8 circuits. Each cell implements a synaptic connection, storing the value of the corresponding weight $W_{i,j}$. Since the array is bi-dimensional, each cell is connected to its four direct neighbors. Connections are bit-serial in both directions. The input vector $\vec{\mathbf{X}}$ is injected "diagonally" on the north edge. A matrix-vector product is then performed, using a classical systolic scheme yielding the potential on the east edge. The activation function σ is then computed externally to the array (by a look-up table), and the resulting output vector $\vec{\mathbf{Y}}$ re-injected. This vector travels west until its reaches the north-west to south-east diagonal—from now on simply referred to as *diagonal*—where it bounces in the north direction. Thanks to this mechanism, the output vector $\vec{\mathbf{Y}}$ can be re-injected as input, to implement the recurrent algorithm of the Hopfield model.

At the same time, the input vector $\vec{\mathbf{X}}$ is reflected on the south edge, and flows north until reaching the diagonal, where it is compared to the corresponding output vector $\vec{\mathbf{Y}}$. Each diagonal cell issues an active-low comparison signal that propagates west. On the west edge, these signals are combined through a special circuit, finally yielding the convergence status. If the computation

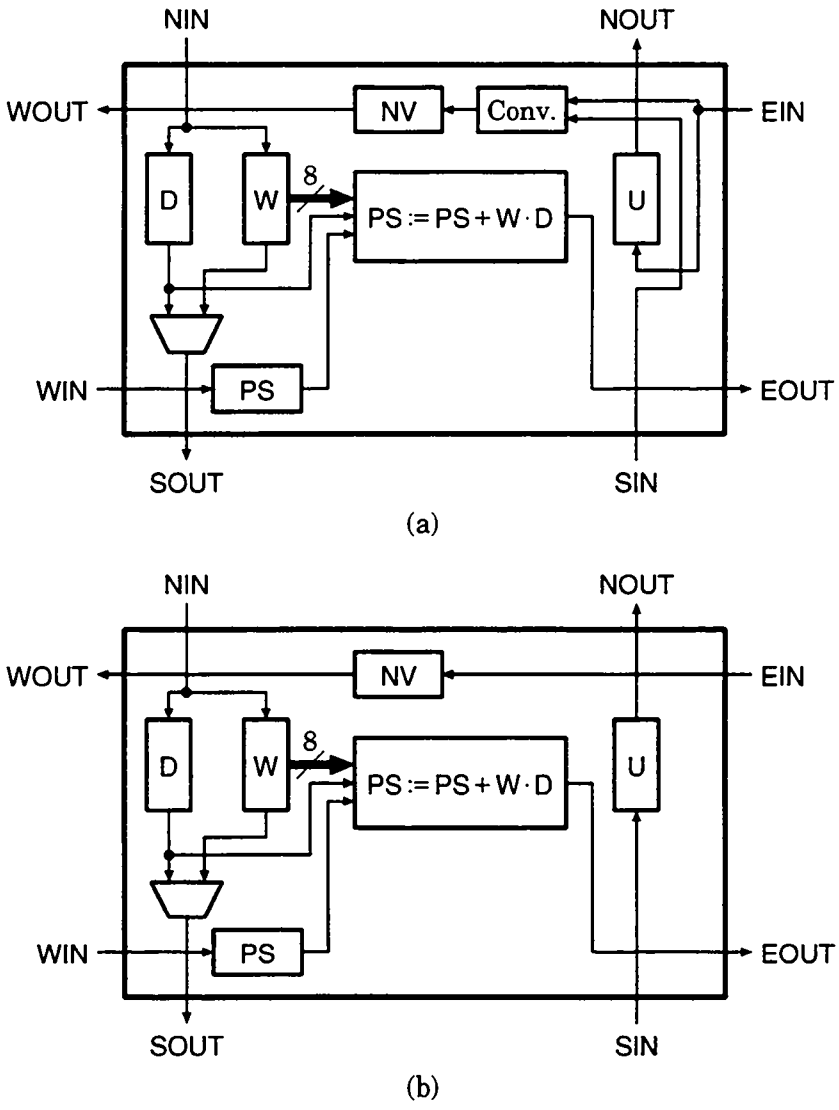


Figure 3.2: *Architecture of the GENES HN8 circuit.* (a) *Diagonal cell.* (b) *Non-diagonal cell.*

of the activation function σ is not combinatorial and requires some clock cycles, a corresponding number of delay elements should be added to the south wrap-around connections.

Figure 3.2(a) shows a GENES HN8 diagonal cell, and figure 3.2(b) a non-diagonal one. Each cell contains the following registers:

Synaptic weight register: W

This register contains the synaptic weight associated with the cell. During initialization, it is loaded through the north-south path. ($N_W = 8$ bits.)

Data register: D

This register is the local storage on the north-south path. It buffers the input corresponding to the synaptic weight stored in the cell. ($N_D = 8$ bits.)

Partial sum register: PS

This register is the local storage on the west-east path. It holds the partial sum while

computing the potentials. ($N_{\mathsf{PS}} = 24$ bits.)

Up register: U
This register is the local storage on the south-north path. On the south of the diagonal, it is used to propagate inputs, while, on the north of the diagonal, it carries the outputs of the neurons. ($N_{\mathsf{U}} = 8$ bits.)

New value register: NV
This register is the local storage on the east-west path. On the east of the diagonal, it is used to propagate the outputs of the neurons, while, on the west of the diagonal, it carries the convergence signals. ($N_{\mathsf{NV}} = 8$ bits.)

All registers are considered as signed two's complement numbers. The communication and computation being bit-serial, it takes a period of $N_{\mathsf{PS}} = 24$ clock cycles—called a *macro-cycle*—for each cell to execute the operation:

$$\mathsf{PS}_{i,j+1} := \mathsf{PS}_{i,j} + \mathsf{W}_{i,j} \cdot \mathsf{D}_{i,j} \qquad \text{for } i,j = 1,2,\ldots,N \tag{3.1}$$

where the symbol := is used to indicate the assignment of a register, in terms of the values held by registers during the previous macro-cycle.[3] During computation, the weight register W is preserved (i.e., $\mathsf{W}_{i,j} := \mathsf{W}_{i,j}$), while the D register is shifted south (i.e., $\mathsf{D}_{i+1,j} := \mathsf{D}_{i,j}$). Boundary conditions are:

$$\mathsf{D}_{1,j} := \mathsf{U}_{1,j} \qquad \text{for } j = 1,2,\ldots,N \tag{3.2}$$
$$\mathsf{PS}_{i,1} := 0 \qquad \text{for } i = 1,2,\ldots,N \tag{3.3}$$
$$\mathsf{U}_{N,j} := \mathsf{D}_{N,j} \qquad \text{for } j = 1,2,\ldots,N \tag{3.4}$$
$$\mathsf{NV}_{i,N} := \sigma\left(\mathsf{PS}_{i,N} + \mathsf{W}_{i,N} \cdot \mathsf{D}_{i,N}\right) \qquad \text{for } i = 1,2,\ldots,N \tag{3.5}$$

On non-diagonal cells, the content of the NV and U registers is simply forwarded unchanged (i.e., $\mathsf{NV}_{i,j-1} := \mathsf{NV}_{i,j}$ and $\mathsf{U}_{i-1,j} := \mathsf{U}_{i,j}$). On diagonal cells, the value coming from the east is propagated north (i.e., $\mathsf{U}_{i,i} := \mathsf{NV}_{i,i+1}$), while the south and east streams are compared to check for the convergence:

$$\mathsf{NV}_{i,i,k} := \sum_{l=0}^{k} \mathsf{NV}_{i,i+1,l} \oplus \mathsf{U}_{i+1,i,l} \qquad \begin{array}{l}\text{for } i = 1,2,\ldots,N \\ \text{for } k = 0,1,\ldots,N_{\mathsf{NV}} - 1\end{array} \tag{3.6}$$

where the sum corresponds to the logical OR operation, and the symbol $\oplus$ represents the XOR operation. The subscripts k and l refer to the bit number. The global convergence is then computed using the circuit shown in figure 3.1. The delay elements have a latency of one macro-cycle, and the convergence signal is only valid after the $(N_{\mathsf{NV}})^{\text{th}}$ clock cycle of each macro-cycle.

The GENES HN8 circuit has been built as a full-custom chip using a 2 μm CMOS technology. It contains an array of 2×2 cells. This device is fully cascadable, since diagonal cells can be configured as non-diagonal. Its implementation is described in an internal report [LEH90].

[3] The assignment $f := g$ is a shorthand for the relation $f(t_{\text{mc}}) = g(t_{\text{mc}} - 1)$. This notation helps avoiding confusion between the discretized macro-cycle time t_{mc} and the discretized update time t introduced in chapter 1.

3.2.2 Overflow handling

A very simple mechanism is used to detect overflows: if the two *most-significant bits (MSBs)* of the PS register differ, the computation of equations (3.1) and (3.5) is inhibited and the PS register is transmitted unchanged (i.e., $\mathsf{PS}_{i,j+1} := \mathsf{PS}_{i,j}$ and $\mathsf{NV}_{i,N} := \sigma\,(\mathsf{PS}_{i,N})$). This means that the partial sum is frozen whenever it leaves the range $[-2^{N_{\mathsf{PS}}-2},\ 2^{N_{\mathsf{PS}}-2}-1]$. Therefore, the effective size of the PS register is equal to $N_{\mathsf{PS}}^{\mathrm{eff}} = N_{\mathsf{PS}} - 1 = 23$ bits.

It is then possible to calculate the maximum number of inputs $n_{\text{V-lim}}$ that will never overflow. The values of the W, D, and PS registers belong to the following ranges:

$$\mathsf{W} \in \left[-2^{N_{\mathsf{W}}^{\mathrm{eff}}-1},\ 2^{N_{\mathsf{W}}^{\mathrm{eff}}-1}-1\right] \tag{3.7}$$

$$\mathsf{D} \in \left[-2^{N_{\mathsf{D}}^{\mathrm{eff}}-1},\ 2^{N_{\mathsf{D}}^{\mathrm{eff}}-1}-1\right] \tag{3.8}$$

$$\mathsf{PS} \in \left[-2^{N_{\mathsf{PS}}^{\mathrm{eff}}-1},\ 2^{N_{\mathsf{PS}}^{\mathrm{eff}}-1}-1\right] \tag{3.9}$$

where $N_{\mathsf{W}}^{\mathrm{eff}} = N_{\mathsf{W}} = 8$ bits and $N_{\mathsf{D}}^{\mathrm{eff}} = N_{\mathsf{D}} = 8$ bits. The weight-input product satisfies then:

$$\mathsf{W}\cdot\mathsf{D} \in \left[\min\left(-2^{N_{\mathsf{W}}^{\mathrm{eff}}+N_{\mathsf{D}}^{\mathrm{eff}}-2}+2^{N_{\mathsf{W}}^{\mathrm{eff}}-1}, -2^{N_{\mathsf{W}}^{\mathrm{eff}}+N_{\mathsf{D}}^{\mathrm{eff}}-2}+2^{N_{\mathsf{D}}^{\mathrm{eff}}-1}\right),\ 2^{N_{\mathsf{W}}^{\mathrm{eff}}+N_{\mathsf{D}}^{\mathrm{eff}}-2}\right] \tag{3.10}$$

Hence, the maximum number of inputs that will never overflow is equal to:

$$\begin{aligned} n_{\text{V-lim}} &= \min\left(\left\lfloor \frac{-2^{N_{\mathsf{PS}}^{\mathrm{eff}}-1}}{\min\left(-2^{N_{\mathsf{W}}^{\mathrm{eff}}+N_{\mathsf{D}}^{\mathrm{eff}}-2}+2^{N_{\mathsf{W}}^{\mathrm{eff}}-1}, -2^{N_{\mathsf{W}}^{\mathrm{eff}}+N_{\mathsf{D}}^{\mathrm{eff}}-2}+2^{N_{\mathsf{D}}^{\mathrm{eff}}-1}\right)}\right\rfloor, \left\lfloor \frac{2^{N_{\mathsf{PS}}^{\mathrm{eff}}-1}-1}{2^{N_{\mathsf{W}}^{\mathrm{eff}}+N_{\mathsf{D}}^{\mathrm{eff}}-2}}\right\rfloor\right) \\ &= \min\left(\left\lfloor \frac{2^{N_{\mathsf{PS}}^{\mathrm{eff}}-1}}{\max\left(2^{N_{\mathsf{W}}^{\mathrm{eff}}+N_{\mathsf{D}}^{\mathrm{eff}}-2}-2^{N_{\mathsf{W}}^{\mathrm{eff}}-1}, 2^{N_{\mathsf{W}}^{\mathrm{eff}}+N_{\mathsf{D}}^{\mathrm{eff}}-2}-2^{N_{\mathsf{D}}^{\mathrm{eff}}-1}\right)}\right\rfloor, \left\lfloor \frac{2^{N_{\mathsf{PS}}^{\mathrm{eff}}-1}-1}{2^{N_{\mathsf{W}}^{\mathrm{eff}}+N_{\mathsf{D}}^{\mathrm{eff}}-2}}\right\rfloor\right) \\ &= \min\left(\left\lfloor \frac{2^{N_{\mathsf{PS}}^{\mathrm{eff}}-1}}{2^{N_{\mathsf{W}}^{\mathrm{eff}}+N_{\mathsf{D}}^{\mathrm{eff}}-2}-2^{N_{\mathsf{W}}^{\mathrm{eff}}-1}}\right\rfloor, \left\lfloor \frac{2^{N_{\mathsf{PS}}^{\mathrm{eff}}-1}}{2^{N_{\mathsf{W}}^{\mathrm{eff}}+N_{\mathsf{D}}^{\mathrm{eff}}-2}-2^{N_{\mathsf{D}}^{\mathrm{eff}}-1}}\right\rfloor, \left\lfloor \frac{2^{N_{\mathsf{PS}}^{\mathrm{eff}}-1}-1}{2^{N_{\mathsf{W}}^{\mathrm{eff}}+N_{\mathsf{D}}^{\mathrm{eff}}-2}}\right\rfloor\right) \end{aligned} \tag{3.11}$$

where the notation $\lfloor v \rfloor$ represents the floor function, defined as the largest integer that is smaller or equal to v. Since the following inequalities are always true:

$$\frac{2^{N_{\mathsf{PS}}^{\mathrm{eff}}-1}-1}{2^{N_{\mathsf{W}}^{\mathrm{eff}}+N_{\mathsf{D}}^{\mathrm{eff}}-2}} < \frac{2^{N_{\mathsf{PS}}^{\mathrm{eff}}-1}}{2^{N_{\mathsf{W}}^{\mathrm{eff}}+N_{\mathsf{D}}^{\mathrm{eff}}-2}-2^{N_{\mathsf{W}}^{\mathrm{eff}}-1}} \tag{3.12}$$

$$\frac{2^{N_{\mathsf{PS}}^{\mathrm{eff}}-1}-1}{2^{N_{\mathsf{W}}^{\mathrm{eff}}+N_{\mathsf{D}}^{\mathrm{eff}}-2}} < \frac{2^{N_{\mathsf{PS}}^{\mathrm{eff}}-1}}{2^{N_{\mathsf{W}}^{\mathrm{eff}}+N_{\mathsf{D}}^{\mathrm{eff}}-2}-2^{N_{\mathsf{D}}^{\mathrm{eff}}-1}} \tag{3.13}$$

equation (3.11) becomes:

$$n_{\text{V-lim}} = \left\lfloor \frac{2^{N_{\mathsf{PS}}^{\mathrm{eff}}-1}-1}{2^{N_{\mathsf{W}}^{\mathrm{eff}}+N_{\mathsf{D}}^{\mathrm{eff}}-2}}\right\rfloor = 2^{N_{\mathsf{PS}}^{\mathrm{eff}}-N_{\mathsf{W}}^{\mathrm{eff}}-N_{\mathsf{D}}^{\mathrm{eff}}+1}-1 \tag{3.14}$$

The last term of equation (3.14) is correct, for $N_{\mathsf{PS}}^{\mathrm{eff}}, N_{\mathsf{W}}^{\mathrm{eff}}, N_{\mathsf{D}}^{\mathrm{eff}} \in \mathbb{N}^*$, only when the condition $N_{\mathsf{PS}}^{\mathrm{eff}} - N_{\mathsf{W}}^{\mathrm{eff}} - N_{\mathsf{D}}^{\mathrm{eff}} + 1 \geq 0$ is satisfied. This relation is always true, because $N_{\mathsf{PS}}^{\mathrm{eff}} > N_{\mathsf{W}}^{\mathrm{eff}} + N_{\mathsf{D}}^{\mathrm{eff}}$ to

ensure that $n_{\text{V-lim}} > 1$.

For the GENES HN8 circuit, equation (3.14) gives $n_{\text{V-lim}} = 255$. Due to the way overflows are handled, the potential is still correct if a single overflow occurs. Therefore, it is possible to implement up to $n = 256$ inputs without errors due to overflows.

If larger networks are implemented, this overflow-detection mechanism has the advantage, besides its simplicity, of indicating the overflow direction. It is then possible to implement the activation function so that any overflowed value is saturated to the largest or smallest representable number.

3.2.3 Performance

In evaluation phase, the performance of ANN implementations is often measured in *connections per second (CPS)*, defined in section 7.1.3. Since a new vector (either a new prototype or the result of a previous iteration) can be processed, by the GENES HN8 array, each macro-cycle, this metric is given by the following formula:

$$P^{\text{eval}} = \frac{N^2 \cdot f}{N_{\text{PS}}} \cdot U \tag{3.15}$$

where N^2 is the number of cells in the array, $f = 10\,\text{MHz}$ is the clock frequency, $N_{\text{PS}} = 24$ is the number of clock cycles per operation, and U is the utilization rate. For instance, the peak performance ($U = 100\,\%$) of a system with 16×16 cells is 107 MCPS.

In the system built by C. Lehmann [LEH93A], the performance is much lower, with a utilization rate of $U \approx 0.75\,\%$. The main reason for this performance drop is that the array is controlled by a SUN workstation. Therefore, all accesses (commands and data) are performed through the SBus interface and a custom 8-bit input/output bus (Mubus). This clearly shows the need for a dedicated system (microprocessor, microcontroller, or sequencer) to control a GENES array.

3.2.4 The Adaline rule in batch mode

Although the GENES HN8 circuit has been designed for Hopfield networks, it can also be used for the Adaline model. This model, described in section 1.7, is a delta rule without activation function (at least during the learning process). The implementation of the recall phase presents no difficulties, since it consists only of a matrix-vector product. On the other hand, the learning phase can be implemented in batch mode, by defining the auxiliary matrices:

$$\mathbf{R} = \sum_{s=1}^{S} \vec{\mathbf{x}}^*(s) \cdot \vec{\mathbf{x}}^{*\mathrm{T}}(s) \tag{3.16}$$

$$\mathbf{Q} = \sum_{s=1}^{S} \vec{\mathbf{d}}(s) \cdot \vec{\mathbf{x}}^{*\mathrm{T}}(s) \tag{3.17}$$

and using the learning rule derived in appendix A.2.1:

$$\mathbf{w}^*(t+1) = \mathbf{w}^*(t) + \alpha(t) \cdot (\mathbf{Q} - \mathbf{w}^*(t) \cdot \mathbf{R}) \tag{3.18}$$

This equation can be rewritten as:

$$\mathbf{w}^{*\mathrm{T}}(t+1) = \alpha(t) \cdot \mathbf{Q}^{\mathrm{T}} + \left(\mathbf{I} - \alpha(t) \cdot \mathbf{R}^{\mathrm{T}}\right) \cdot \mathbf{w}^{*\mathrm{T}}(t) \tag{3.19}$$

It is now possible to load the array with the matrix $\mathbf{W} = \mathbf{I} - \alpha(t) \cdot \mathbf{R}^{\mathrm{T}}$, and to compute equation (3.19) by setting the input vectors to $\vec{\mathbf{X}} = \mathbf{w}_i^{*\mathrm{T}}(t)$ and injecting the vectors $\alpha(t) \cdot \mathbf{Q}_i^{\mathrm{T}}$ instead of zeroes on the west edge of the array. Using this scheme, the output vectors—corresponding to $\vec{\mathbf{Y}} = \mathbf{w}_i^{*\mathrm{T}}(t+1)$—can be re-injected in the array to be used as input for the next iteration, as for the Hopfield model.

This scheme may be useful, because the external computation (i.e., the matrix $\mathbf{I} - \alpha \cdot \mathbf{R}^{\mathrm{T}}$ and the vectors $\alpha \cdot \mathbf{Q}_i^{\mathrm{T}}$) can be made negligible, if the learning coefficient α is not modified at each iteration but takes a few discrete values. If the number of prototypes S is smaller than the number of cells N per array's edge, it can be shown that this computation may also be performed on the array. However, this is of limited use since the number of prototypes S is often large.

3.3 The GENES HH8 circuit

The second implementation, called *GENES HH8* for *Hopfield Network with Hebbian Learning Rule, 8 Bits*, has also been designed by C. Lehmann. This circuit extends the functionality of GENES arrays by implementing the Hebbian learning rule. Hence, besides Hopfield networks, it can also implement single-layer networks with the Perceptron or delta rules.[4]

3.3.1 Architecture

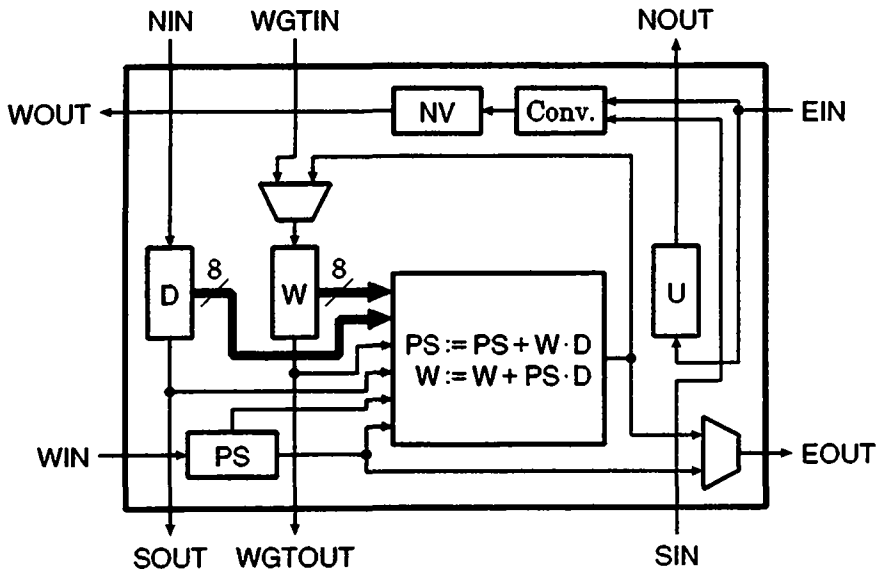
Figure 3.3(a) shows a GENES HH8 diagonal cell and figure 3.3(b) a non-diagonal one. The cell's architecture is very similar to that of the GENES HN8 circuit, in particular, the same registers have been implemented. Their sizes are identical ($N_{\mathsf{W}} = 8$ bits, $N_{\mathsf{D}} = 8$ bits, $N_{\mathsf{PS}} = 24$ bits, $N_{\mathsf{U}} = 8$ bits, and $N_{\mathsf{NV}} = 8$ bits), and the computation is also executed with two's complement integers. A noticeable difference, between the two implementations, is the addition of a dedicated path to load the weights.

A GENES HH8 array implements two different functions: the *matrix-vector product* and the *Hebbian learning rule*. The first mode is identical to the GENES HN8 circuit, the cells computing equation (3.1). Diagonal cells check also for the convergence as described by equation (3.6). In the Hebbian learning mode, each cell executes the following operation:

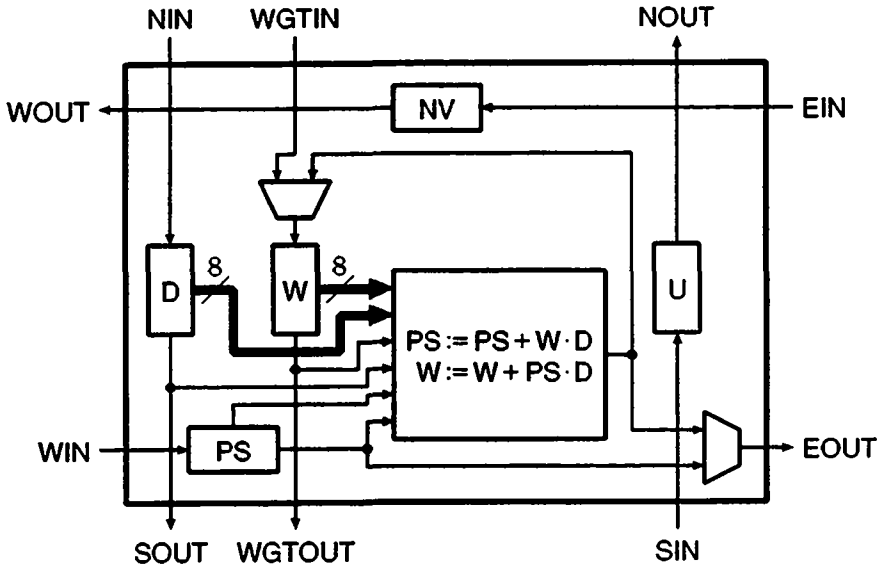
$$\mathsf{W}_{i,j} := \mathsf{W}_{i,j} + \left\lfloor \frac{\mathsf{PS}_{i,j-1,7..0} \cdot \mathsf{D}_{i-1,j}}{2^c} \right\rfloor \tag{3.20}$$

In this mode, only the $N_{\mathsf{PS}}^{\mathrm{lrn}} = 8$ first bits of the PS register are used. The division by 2^c—where $c \in [0, 8]$ is a global constant—and the floor operation are implemented by a right shift of c bits. At the same time, the D and PS registers are transmitted unchanged (i.e., $\mathsf{D}_{i+1,j} := \mathsf{D}_{i,j}$ and $\mathsf{PS}_{i,j+1} := \mathsf{PS}_{i,j}$). The behavior of the NV and U registers in both modes is identical to that of the

[4] The mapping of the delta rule on a GENES array requires some external logic to compute the error signal vector $\vec{\delta}$. An example of such a circuit is presented in section 5.1.1.



(a)



(b)

Figure 3.3: *Architecture of the GENES HH8 circuit.* (a) *Diagonal cell.* (b) *Non-diagonal cell.*

GENES HN8 circuit (see section 3.2.1). Since the communication and computation are bit-serial, a macro-cycle is also equal to $N_{PS} = 24$ clock cycles.

The GENES HH8 circuit, has been built as a full-custom chip using a 2 μm CMOS technology, and contains an array of 4×4 cells. However, this circuit is not functional.

3.3.2 Overflow handling

In the GENES HH8 circuit, the overflow handling mechanism of the GENES HN8 implementation for the matrix-vector product has been retained. This mechanism has also been adapted to the W register in learning mode. This means that weights are frozen whenever they leave the range

$[-2^{N_W-2}, 2^{N_W-2}-1]$. This is one of the main drawbacks of this scheme. If a weight overflows during the learning process, it will never converge to its ideal value, even if this quantity belongs to the range of representable values. This is especially bad for this implementation, because weight overflows may occur after the first update. Although the GENES HH8 chip is not working, this would have prevented its use for any real applications. From what precedes, it becomes clear that an overflow detection mechanism should not freeze the weight register. Such a scheme is implemented in the GENES IV circuit (see section 4.2.7).

3.3.3 Performance

During the evaluation phase the performance (measured in CPS) is given by equation (3.15). In the learning phase, the performance is measured in *connection updates per second (CUPS)*, defined in section 7.1.3. For the GENES HN8 array this metric is equal to:

$$P^{\text{lrn}} = \frac{N^2 \cdot f}{2N_{\text{PS}}} \cdot U \tag{3.21}$$

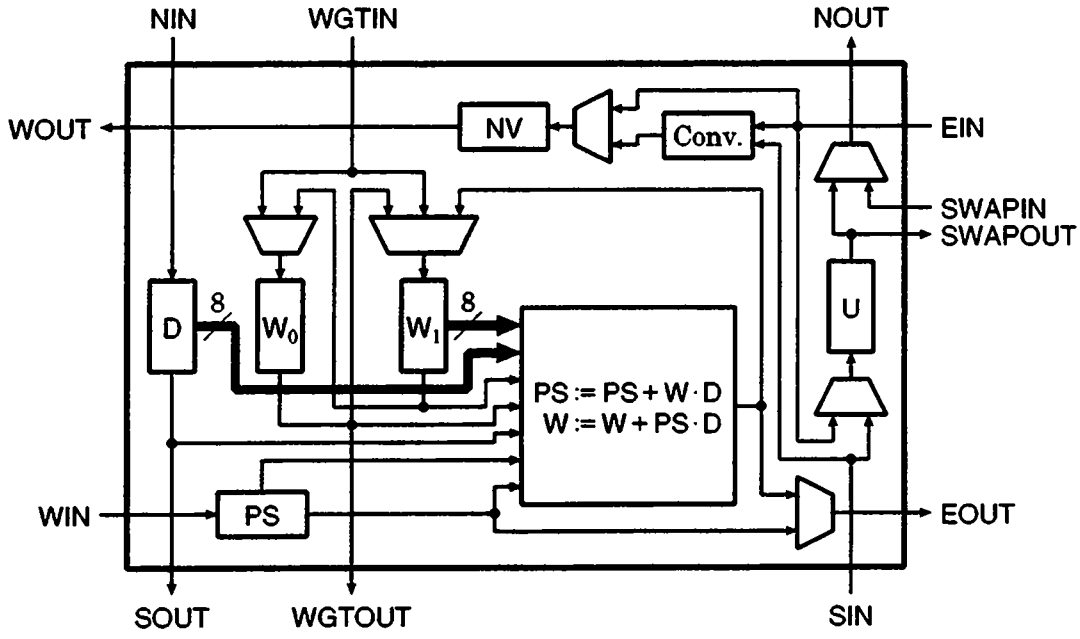
where the factor 2 accounts for the fact that a matrix-vector product and a Hebbian updating operation are required for each weight-matrix update. The strength of the GENES architecture, like any systolic array, comes from a large number of cells computing in parallel or, more precisely, in a pipelined fashion. Therefore, the utilization rate U and the performance P^{lrn} may decrease if the pipeline is not kept full. During learning, a matrix-vector product is performed for every input vector of an epoch, then the weights are updated by executing the same number of Hebbian learning operations. However, the array must be emptied before the operation change and re-filled afterwards. Since the epoch length should be kept low to help convergence, this may significantly reduce the utilization rate. For an epoch length of $e = 2N$ the maximum achievable utilization rate drops to $U_{\max} = \frac{e}{2N+e-2} \approx 50\,\%$. On the GENES IV circuit, this problem is solved by the use of a systolic instruction flow (see section 4.1.3).

3.4 The GENES VM16 circuit

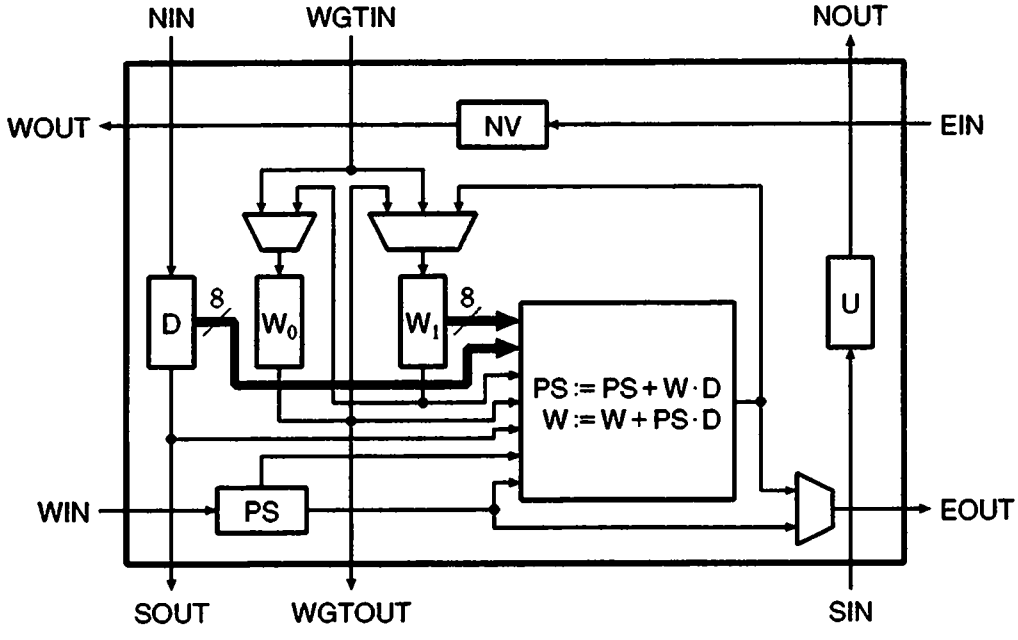
After the first two prototypes, C. Lehmann implemented a third circuit, called *GENES VM16* for *Virtual Matrix, 16 Synapses*. With this system, weight matrices larger than the physical array can be efficiently used. These matrices are labeled *virtual* by analogy with virtual memory: the virtual matrix being paged on the physical one. The matrix-swapping mechanism can also be used to implement multi-layer feed-forward networks in the evaluation phase. However, due to the lack of transposition mechanism, the implementation of the back-propagation rule is impossible. An unsuccessful attempt has also been made to provide primitives for Kohonen model. These operations being inappropriate for the algorithm, they are not presented here.

Figure 3.4(a) and (b) show the GENES VM16 diagonal and non-diagonal cells. Each of them is essentially a GENES HH8 cell augmented with a second 8-bit weight register. The other registers have been left unchanged. In particular they have identical sizes and are interpreted as two's complement values. The W_1 register is connected to the arithmetic unit and used for computation, while the W_0 register is only connected to the weight path.

The U and NV paths on diagonal cells have also been modified. First, the cell can be configured so that data flow from south to north and from east to west, like on non-diagonal cells. Second, an



(a)



(b)

Figure 3.4: *Architecture of the GENES VM16 circuit.* (a) *Diagonal cell.* (b) *Non-diagonal cell.*

additional pair of input and output ports, called SWAPIN and SWAPOUT, are implemented to enter and retrieve data into/from the south-north path. Thanks to these ports, the input vector $\vec{\mathbf{X}}$ does not need to be entered "diagonally," as shown in figure 3.1, but can be entered on the diagonal, all elements at the same time. It flows then north and is wrapped around on the north edge before entering the D registers. A similar scheme makes it possible to retrieve the output vector $\vec{\mathbf{Y}}$ on the diagonal, all elements at the same time.

Matrices larger than the array can then be used by decomposing them into sub-matrices that fit on the hardware:

$$\mathbf{w}(t) = \begin{pmatrix} \mathbf{w}_{[1,1]}(t) & \mathbf{w}_{[1,2]}(t) & \cdots & \mathbf{w}_{[1,r]}(t) \\ \mathbf{w}_{[2,1]}(t) & \mathbf{w}_{[2,2]}(t) & \cdots & \mathbf{w}_{[2,r]}(t) \\ \vdots & \vdots & \ddots & \vdots \\ \mathbf{w}_{[q,1]}(t) & \mathbf{w}_{[q,2]}(t) & \cdots & \mathbf{w}_{[q,r]}(t) \end{pmatrix} \tag{3.22}$$

At the start, the sub-matrix $\mathbf{w}_{[1,1]}$ is loaded into the W_1 registers and the computation begins. Meanwhile, the sub-matrix $\mathbf{w}_{[1,2]}$ is loaded in background. When it is resident in the W_0 registers, the two weight registers are swapped and the computation can proceed. For the matrix-vector product, the result of the first iteration is injected in the PS registers instead of zeroes. During the second iteration, the sub-matrix $\mathbf{w}_{[1,1]}$ can be stored in memory (if it has been updated) while the sub-matrix $\mathbf{w}_{[1,3]}$ is loaded. This procedure is then repeated for all sub-matrices of the row. While the last sub-matrix $\mathbf{w}_{[1,r]}$ is used, the first sub-matrix of the next row $\mathbf{w}_{[2,1]}$ is loaded. After the swap, the computation re-starts with zeroes injected on the west edge. All rows of sub-matrices are computed the same way.

The weight register swap is performed by sending a vector of special patterns[5] in the D registers. When such a pattern is present in a cell's D register, the current macro-cycle is used to shift the content of a weight register into the other and vice-versa. In this way, the weight register swapping does not occur globally, but rather takes place on each cell just after the last data associated with a sub-matrix has been processed. This avoids the need of emptying and re-filling the array at each sub-matrix change. This method is generalized and optimized on the GENES IV array by the implementation of a systolic instruction flow (see section 4.1.3).

The use of the return path formed by the NV registers to re-inject the result of an iteration into the PS registers imposes an epoch length of $e = 2N - 1$, where N is the number of cells per array's edge.[6] A careful analysis of the data flow shows that there is only one macro-cycle per epoch where no cell is swapped (see section 4.2.4). No new sub-matrix can be loaded, since this operation takes N macro-cycles. A possible solution to this problem is to add a buffer or fixed-size FIFO on the east or west wrap-around connections and to increase the epoch length to $e \geq 3N - 2$. On the GENES IV chip, this problem has been solved by controlling the shift of weight registers on a column basis. The weight-matrix loading starts with the first column, then, after a macro-cycle, it continues with the second one, and so on. With this scheme, epoch lengths as short as $e = 2N$ are implementable (see section 4.2.4).

This mechanism can also be used to efficiently implement multi-layer feed-forward networks

[5] Such a pattern is recognized when the two MSBs of a word are different, that is, when it belongs to one of the ranges $[-2^{N_D-1}, -2^{N_D-2} - 1]$ and $[2^{N_D-2}, 2^{N_D-1} - 1]$. As a consequence, the effective size of the D register is reduced to $N_D^{\text{eff}} = N_D - 1 = 7$ bits.

[6] The term -1 accounts for the macro-cycle used to swap the registers.

in the evaluation phase (see section 1.4.3).

When virtual matrices are not required, it is also possible to cascade both weight registers. In this way, the W_0 register can be considered as 8 fractional bits placed after the binary point of the W_1 register. The matrix-vector product is performed identically, which means that only the most-significant half of the weight (i.e., the W_1 register) is used. On the other hand, the Hebbian learning rule is computed as:

$$W_{1,i,j}.W_{0,i,j} := W_{1,i,j}.W_{0,i,j} + \frac{PS_{i,j-1,7..0} \cdot D_{i-1,j}}{2^{N_{W_0}}} \tag{3.23}$$

where $W_1.W_0$ is a shorthand to represent the concatenation of the W_0 register after the binary point of the W_1 register. With this technique, small updates can be accumulated and eventually make a visible change on the weight used in forward mode (i.e., the W_1 register). This prevents the convergence to be prematurely stopped because all updates are rounded to zero. Since the W_1 register is also connected to the weight path, 16-bit weight matrices can be loaded and retrieved.

The GENES VM16 circuit, has been built as a full-custom chip using a 2 μm CMOS technology, and contains an array of 4×4 cells. However, this circuit is not functional.

3.5 Motivations for designing a new GENES array

The analysis presented in this chapter has clearly shown that the GENES architecture may offer a large computing power. However, the three proposed implementations hide some specific flaws that prevent them to be efficiently used. These problems are a poor utilization rate (because the array must be frequently emptied and re-filled) and a cumbersome programming model (vectors of special patterns to swap sub-matrices, etc.). Moreover, the chosen overflow mechanism is not appropriate to learning.

The first three versions of the GENES array are specific systems that can be used only for a very restricted number of ANN models. Therefore, beside correcting the above-mentioned flaws, a new implementation should be made more general, and hence suited to a multi-model neural computer. The fact that the last two circuits have not been found fully functional, suggests that standard cells, rather than full-custom should be used for a university prototype.

References

[BLA90] F. Blayo. *Une implantation systolique des algorithmes connexionnistes.* Ph.D. thesis no. 904, EPFL, Lausanne (CH), 1990.

[LEH90] C. Lehmann. *Conception VLSI: de la parole au geste.* Rapport interne, LAMI, EPFL, Lausanne (CH), August 1990.

[LEH91] C. Lehmann and F. Blayo. *A VLSI Implementation of a Generic Systolic Synaptic Building Block for Neural Networks.* In J. G. Delgado-Frias and W. R. Moore (eds.), *VLSI for Artificial Intelligence and Neural Networks*, chapter 4.8, pp. 325–334. Plenum Press, New York, NY (USA), 1991. Proceedings of the 2nd workshop on VLSI for AI and NNs.

[LEH93A] C. Lehmann, M. Viredaz, and F. Blayo. *A Generic Systolic Array Building Block for Neural Networks with On-Chip Learning. IEEE Transactions on Neural Networks*, 4(3):400–407, May 1993. Special issue on neural network hardware.

[LEH93B] C. Lehmann. *Réseaux de neurones compétitifs de grandes dimensions pour l'auto-organisation: analyse, synthèse et implantation sur circuits systoliques.* Ph.D. thesis no. 1129, EPFL, Lausanne (CH), 1993.

Sichè si conclude, per le assegniate ragione, che 'l moto d'ogni peso è più facile per linia equidiaciente che per linia obbliqua.[1]

Leonardo da Vinci, *Cod. Madrid I*, f.43r

4

The GENES IV Systolic Array

In this chapter a novel design of the GENES array, called *GENES IV* [IEN93], is presented. This evolutionary architecture keeps the features of the previous versions, while avoiding as well as possible their restrictions. In particular, the bi-dimensional nature of the array and the bi-directional data streams have been conserved. The architecture has also been extended to new ANN models.

The GENES HN8 array is restricted to the Hopfield model (see section 1.9). The GENES HH8 implementation can implement the Perceptron and the delta rules (see sections 1.6 and 1.7) as well. Finally, multi-layer feed-forward networks in the evaluation phase (see section 1.4.3) can also be implemented on the GENES VM16 circuit. In addition to these algorithms, the GENES IV array is extended to support the back-propagation rule (see section 1.8) and the Kohonen models with minimum or recurrent network (see section 1.10).

The GENES IV architecture includes substantial improvements in data input/output, matrix swapping, overflow handling, and learning precision. The arithmetic precision of the registers is doubled. Finally the performance is increased and the programming model simplified by introducing a systolic flow of instructions.

4.1 Functional principles

In his Ph.D. thesis [BLA90], F. Blayo decomposed a set of ANN algorithms into four different operations. A similar analysis [LEH93], extended to a larger number of models, led to nine operations. For a given set of algorithms, the exact number of operations depends on the type of decomposition that is done, which in turn is biased by the target architecture and technology. These operations can be divided in two categories depending on the required amount of computation. In the first

[1] *Therefore, we may conclude that, because of the mentioned reasons, the motion of a weight along the horizontal line is easier than along the oblique line* (translation by Paolo Ienne). On the GENES IV array, the horizontal and vertical paths are arbitrarily distinguished.

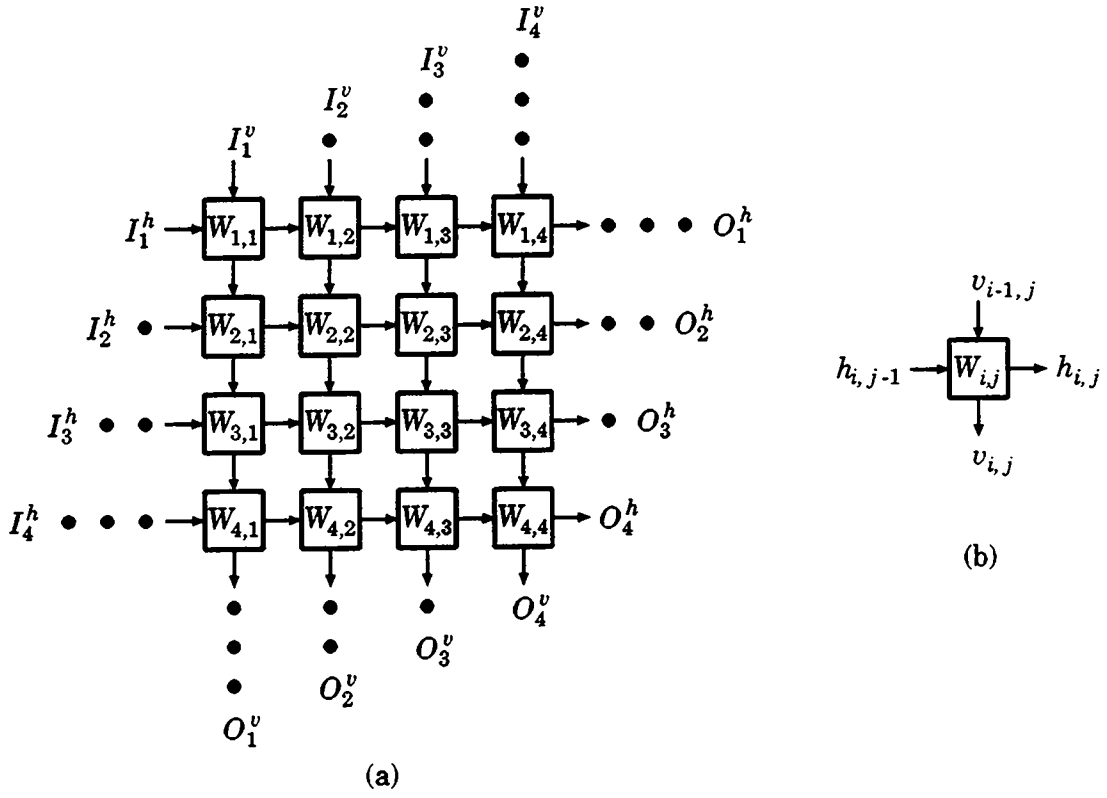


Figure 4.1: *Functional structure of the GENES IV systolic array.* (a) *Active data streams (4 × 4 PEs).* (b) *Definition of the input/output ports of a PE.*

category fall all operations whose computation is proportional to number of synapses, that is, to the number of elements of the weight matrix. They are also said to be of order $m \cdot n$ or to be $O(m \cdot n)$, using the O-notation[2] [KNU68]. The computation of operations in the second class increases with the number of neurons, that is, the number of rows of the weight matrix: $O(m)$. The bi-dimensional GENES IV systolic array targets the former category (matrix operations), while the latter (vector computation, activation function, etc.) is performed outside the mesh.

Six GENES IV operations have been found necessary to implements the models presented in chapter 1. Among these operations, the two that search the smallest or largest element of a vector are rather peculiar. On a serial machine they could easily be programmed as $O(m)$. However, because of the GENES IV architecture, implementing these operations as an $O(m^2)$ process, sharing the same array and executed in parallel, has been found more elegant than adding some dedicated units outside the array.

The bit-serial *processing elements (PEs)* are connected by serial lines to their four direct neighbors to form a square mesh. Figure 4.1(a) shows the active data streams in the array. Additional streams are implemented as described in section 4.2. These streams undergo no computation, and hence can be ignored when analyzing the functional principles. Each of the $N \times N$ PEs contains one element $W_{i,j}$ of the matrix **W**. This matrix can either be the weight matrix **w**, one of its sub-matrices, or an auxiliary matrix (e.g., the neighborhood matrix λ).

[2] More precisely, the O-notation means that the order of the operation is lower or equal to the corresponding expression.

Each operation takes up to three operands: the stored matrix and the two input vectors $\vec{\mathbf{I}}^h$ and $\vec{\mathbf{I}}^v$. Its result is either one of the two output vectors $\vec{\mathbf{O}}^h$ and $\vec{\mathbf{O}}^v$, or an update of the matrix $\mathbf{W}$. Since no operation produces two vectors, of the two output streams, only one carries the result. The other consists of the corresponding input vectors delayed by N macro-cycles. The same remark applies to the weight update operations. The input vectors flow unchanged through the array and appear as output vectors with N macro-cycles of latency.

Figure 4.1(b) defines the input and output ports $h_{i,j-1}$, $h_{i,j}$, $v_{i-1,j}$, and $v_{i,j}$ of each PE. The boundary conditions are the following:

$$h_{i,0} = I_i^h \quad \text{for } i = 1,2,\ldots,N \tag{4.1}$$
$$v_{0,j} = I_j^v \quad \text{for } j = 1,2,\ldots,N \tag{4.2}$$
$$O_i^h = h_{i,N} \quad \text{for } i = 1,2,\ldots,N \tag{4.3}$$
$$O_j^v = v_{N,j} \quad \text{for } j = 1,2,\ldots,N \tag{4.4}$$

4.1.1 Operations

Beside the *no operation (NOP)* command, the GENES IV instruction set is composed of six operations grouped in three classes of two. Each of these operations can be executed with one of the two weight registers: W_0 and W_1. In addition, they are two modes: *normal* and *transpose*, which control the input and output data streams. These modes may be chosen independently from the operation, although the transpose mode is useless for most of them. The operations are described here in normal mode, the transpose mode being presented in section 4.1.2.

Evaluation operations

The first operation is the *matrix-vector product*, which is implemented using the same scheme as in the previous versions (see chapter 3). The weights are local and constant (i.e., $W_{i,j} := W_{i,j}$). Vertical vectors flow unchanged (i.e., $v_{i,j} := v_{i-1,j}$), while the actual computation takes place on the horizontal path. Each PE computes a multiply-accumulate operation:

$$h_{i,j} := h_{i,j-1} + W_{i,j} \cdot v_{i-1,j} \quad \text{for } i,j = 1,2,\ldots,N \tag{4.5}$$

Therefore, the global result on the whole array is:

$$\vec{\mathbf{O}}^h := \vec{\mathbf{I}}^h + \mathbf{W} \cdot \vec{\mathbf{I}}^v \tag{4.6}$$

In equation (4.6), as in all global relations, the symbol := refers to an assignment after $2N$ macro-cycles, while it represents a single macro-cycle of latency in the local formulae. The additive term $\vec{\mathbf{I}}^h$ is required for the implementation of virtual matrices. When the weight matrix $\mathbf{w}$ is larger than the array, it is decomposed into sub-matrices as shown in equation (3.22). When computing the first sub-matrix of each line, this vector is set to zero (i.e., $\vec{\mathbf{I}}^h = \vec{\mathbf{0}}$). The sub-matrices are then successively processed, while the partial sums resulting from the previous iterations are re-injected into the array. The matrix-vector product can also be viewed as the *scalar* or *dot product* between the input vector $\vec{\mathbf{I}}^v$ and each row of the weight matrix $\mathbf{W}$.

The second operation is the computation of the *squared Euclidean distance* between the input vector $\vec{\mathbf{I}}^v$ and each rows of the matrix $\mathbf{W}$. Since this operation is used in the Kohonen algorithm

to find the smallest distance (see section 1.10.1), there is no need to compute the square root. The data streams are identical to those of the previous operation (i.e., $W_{i,j} := W_{i,j}$ and $v_{i,j} := v_{i-1,j}$), and each PE executes:

$$h_{i,j} := h_{i,j-1} + (v_{i-1,j} - W_{i,j})^2 \quad \text{for } i,j = 1,2,\ldots,N \tag{4.7}$$

Globally, the array computes the following expression:

$$O_i^h := I_i^h + \sum_{j=1}^{N} (I_j^v - W_{i,j})^2 = I_i^h + \left|\vec{\mathbf{I}}^v - \mathbf{W}_i^{\mathrm{T}}\right|^2 \quad \text{for } i = 1,2,\ldots,N \tag{4.8}$$

Here again, the additive term I_i^h is used to deal with virtual matrices.

Since these two operations are mainly used for the evaluation phase of ANN models, they are referred to as *evaluation*. The Manhattan distance has not been implemented, in this category, because it is usually adopted for its low computational requirement rather than for its intrinsic properties. Therefore, it is of little use once the Euclidean distance is available.

Minimum/maximum operations

The next class of operations contains the search for the *minimum* and the *maximum element* of a vector. The weights are ignored and the vertical vector flows unchanged (i.e., $v_{i,j} := v_{i-1,j}$). In the case of the minimum, a value can propagate horizontally though a PE only if it is smaller than the vertical input. If it is not, it is saturated to the maximum representable value $N_{\max}$, that is:

$$h_{i,j} := \begin{cases} h_{i,j-1} & \text{if } h_{i,j-1} \leq v_{i-1,j} \\ N_{\max} & \text{otherwise} \end{cases} \quad \text{for } i,j = 1,2,\ldots,N \tag{4.9}$$

At the array level, this computation yields the following result:

$$O_i^h := \begin{cases} I_i^h & \text{if } I_i^h \leq \min_j (I_j^v) \\ N_{\max} & \text{otherwise} \end{cases} \quad \text{for } i,j = 1,2,\ldots,N \tag{4.10}$$

When the same vector is injected horizontally and vertically: $\vec{\mathbf{I}}^h = \vec{\mathbf{I}}^v = \vec{\mathbf{I}}$, this result becomes:

$$O_i^h := \begin{cases} I_i & \text{if } I_i = \min_j (I_j) \\ N_{\max} & \text{otherwise} \end{cases} \quad \text{for } i,j = 1,2,\ldots,N \tag{4.11}$$

This vector—which contains everywhere the largest representable number $N_{\max}$ except at the position(s) corresponding to the smallest element(s) $\min_j (I_j)$ of the original vector $\vec{\mathbf{I}}$—gives the value of the smallest element(s), as well as its (their) position(s). Virtual matrices can be used with the minimum operation described by equation (4.10). The idea is simple: an input I_i^h must be smaller or equal to all inputs I_j^v of all sub-matrices on a row to propagate through all iterations.

The maximum element of a vector is searched in a similar way. For the same horizontal and vertical input (i.e., $\vec{\mathbf{I}}^h = \vec{\mathbf{I}}^v = \vec{\mathbf{I}}$), the following result is output:

$$O_i^h := \begin{cases} I_i & \text{if } I_i = \max_j (I_j) \\ N_{\min} & \text{otherwise} \end{cases} \quad \text{for } i,j = 1,2,\ldots,N \tag{4.12}$$

Instruction	PE operation	Array global result
NOP	$h_{ij} := h_{ij-1}$	$\vec{\mathbf{O}}^h := \vec{\mathbf{I}}^h$
Matrix-vector product	$h_{ij} := h_{ij-1} + W_{ij} \cdot v_{i-1j}$	$\vec{\mathbf{O}}^h := \vec{\mathbf{I}}^h + \mathbf{W} \cdot \vec{\mathbf{I}}^v$
Square Euclidean distance	$h_{ij} := h_{ij-1} + (v_{i-1j} - W_{ij})^2$	$O_i^h := I_i^h + \lvert\vec{\mathbf{I}}^v - \mathbf{W}_i^{\mathrm{T}}\rvert^2$
Minimum element	$h_{ij} := \begin{cases} h_{ij-1} & \text{if } h_{ij-1} \leq v_{i-1j} \\ N_{\max} & \text{otherwise} \end{cases}$	$O_i^h := \begin{cases} I_i^h & \text{if } I_i^h \leq \min_j (I_j^v) \\ N_{\max} & \text{otherwise} \end{cases}$
Maximum element	$h_{ij} := \begin{cases} h_{ij-1} & \text{if } h_{ij-1} \geq v_{i-1j} \\ N_{\min} & \text{otherwise} \end{cases}$	$O_i^h := \begin{cases} I_i^h & \text{if } I_i^h \geq \max_j (I_j^v) \\ N_{\min} & \text{otherwise} \end{cases}$
Hebbian learning rule	$W_{ij} := W_{ij} + h_{ij-1} \cdot v_{i-1j}$	$\mathbf{W} := \mathbf{W} + \vec{\mathbf{I}}^h \cdot \vec{\mathbf{I}}^{v\,\mathrm{T}}$
Kohonen learning rule	$W_{ij} := W_{ij} + h_{ij-1} \cdot (v_{i-1j} - W_{ij})$	$\mathbf{W}_i := \mathbf{W}_i + I_i^h \cdot (\vec{\mathbf{I}}^{v\,\mathrm{T}} - \mathbf{W}_i)$

Table 4.1: *Operations of the GENES IV array in normal mode.*

where $N_{\min}$ is the smallest representable number.

Learning operations

Finally, the last class of operations are the weight update according to the *Hebbian* and *Kohonen learning rules*. For these operations, both input vectors flow unmodified through the array (i.e., $h_{ij} := h_{ij-1}$ and $v_{ij} := v_{i-1j}$). Each PE executes equation (4.13) for the Hebbian rule and equation (4.14) for the Kohonen rule:

$$W_{ij} := W_{ij} + h_{ij-1} \cdot v_{i-1j} \qquad \text{for } i,j = 1,2,\ldots,N \tag{4.13}$$

$$W_{ij} := W_{ij} + h_{ij-1} \cdot (v_{i-1j} - W_{ij}) \qquad \text{for } i,j = 1,2,\ldots,N \tag{4.14}$$

The global update rules are respectively:

$$\mathbf{W} := \mathbf{W} + \vec{\mathbf{I}}^h \cdot \vec{\mathbf{I}}^{v\,\mathrm{T}} \tag{4.15}$$

$$\mathbf{W}_i := \mathbf{W}_i + I_i^h \cdot \left(\vec{\mathbf{I}}^{v\,\mathrm{T}} - \mathbf{W}_i\right) \qquad \text{for } i = 1,2,\ldots,N \tag{4.16}$$

It should be noticed that the Hebbian rule is general enough to implement the Perceptron, delta, and back-propagation rules (see sections 1.6, 1.7, and 1.8). Since the learning operations are local, the implementation of virtual matrices offers no problem. Table 4.1 summarizes the operations of the GENES IV array's operations in normal mode.

4.1.2 Transpose mode

In the back-propagation rule (see section 1.8), the computation of the error signal $\vec{\delta}^{[k]}$, given by equation (1.37), requires the multiplication of the transpose weight matrix $\mathbf{w}^{[k+1]\,\mathrm{T}}$ by the vector $\vec{\delta}^{[k+1]}$. Instead of implementing a transpose operation, it is much more efficient to make use either of the stored matrix or of its transpose. This can easily be achieved by exchanging the

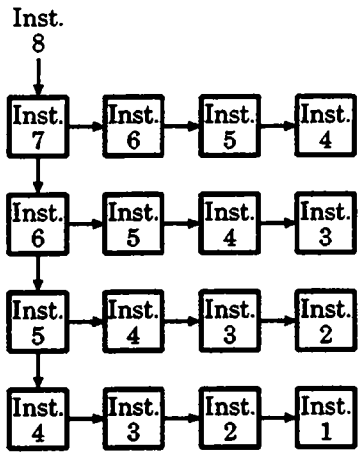


Figure 4.2: *Systolic instruction flow in the GENES IV array (4 × 4 PEs).*

roles of rows and columns. The horizontal vector $\vec{\mathbf{I}}^h$ flows now unchanged (i.e., $h_{i,j} := h_{i,j-1}$), and equation (4.5) is replaced by:

$$v_{i,j} := v_{i-1,j} + W_{i,j} \cdot h_{i,j-1} \qquad \text{for } i,j = 1,2,\ldots,N \tag{4.17}$$

At the array level, equation (4.6) becomes:

$$\vec{\mathbf{O}}^v := \vec{\mathbf{I}}^v + \mathbf{W}^{\mathrm{T}} \cdot \vec{\mathbf{I}}^h \tag{4.18}$$

Although any of the six operations can be executed in transpose mode, this is of little use[3] The behavior of the GENES IV array in transpose mode can nonetheless be derived from table 4.1 by exchanging the symbols h and v in the second column (with $h_{i,j-1}$ being replaced by $v_{i-1,j}$ and vice-versa) and recalculating the third one. The search for the minimum or maximum element and the Hebbian learning rule being symmetric, their result in transpose mode is equivalent to that in normal mode (though a different control sequence is required).

4.1.3 Systolic instruction flow

As described in section 3.3.3 an important drawback of previous GENES arrays is their low utilization rate, which is due to their "conventional" SIMD nature. This paradigm—where all PEs execute the same operation simultaneously—implies that the array must be emptied before each operation change, and re-filled afterwards. To avoid this inefficiency, GENES IV PEs should be locally controlled by instructions following the appropriate data, much like in a pipelined microprocessor. This is implemented by the *systolic instruction flow* shown in figure 4.2. Instructions are dispatched to the north-west PE and then systolically propagated through the array. This scheme should still be considered as SIMD in Flynn's taxonomy (see section 2.1.1), but the common instruction stream flows systolically through the PEs. This mechanism presents some similarities with ISAs (see section 2.1.3).

[3] In the MANTRA I machine (see chapter 5), the external $\mathrm{UIN}_{39..0}$ and $\mathrm{LIN}_{39..0}$ paths are not symmetric and the transpose mode is sometimes used to take advantage of their respective characteristics.

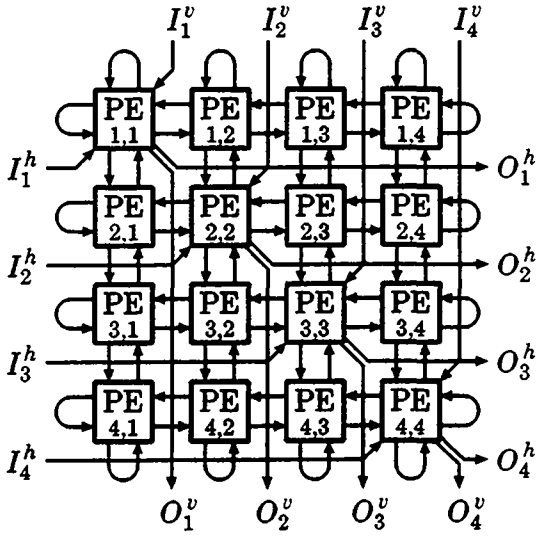


Figure 4.3: *Architecture of the GENES IV systolic array (4 × 4 PEs).*

Being associated with the operation, the control of the normal or transpose mode should also be part of the instruction word. The input and output ports should be independently controlled, because the former always switch mode one macro-cycle earlier than the latter.

The instruction word is also used to select the weight register to be used. This avoids the cumbersome use of a vector of special patterns, as on the GENES VM16 circuit (see section 3.4).

4.2 Implementation

In many ways, the implementation of the GENES IV array differs from the functional principles presented in section 4.1. The actual implementation is discussed in this section, as well as design choices.

4.2.1 Array architecture

The architecture of the GENES IV array is illustrated in figure 4.3. This implementation presents a few differences with the functional structure shown in figure 4.1(a). Like on the GENES VM16 array, presented in section 3.4, the input and output ports are located on the north-west to south-east diagonal. The advantage is that no external hardware is required to "diagonalize" the inputs or "undiagonalize" the outputs, as shown in figure 4.1(a). All the components of a vector are entered at the same time and flow in the reverse direction, before being wrapped around on an array's edge. Similarly, an output vector is reflected on an edge and then flows backwards, before being extracted, all components simultaneously. The disadvantage is an additional latency of N macro-cycles.

The inter-PE communication is bit-serial *least-significant bit (LSB)* first. The bit-serial protocol is imposed by the silicon area of a parallel multiplier—which would have prevented multiple PEs to be integrated on a single die—and by the number of pads required for the parallel communication (about 600 for a single-PE chip). Serial solutions with multiple bits have also been discarded because of their increased hardware complexity.

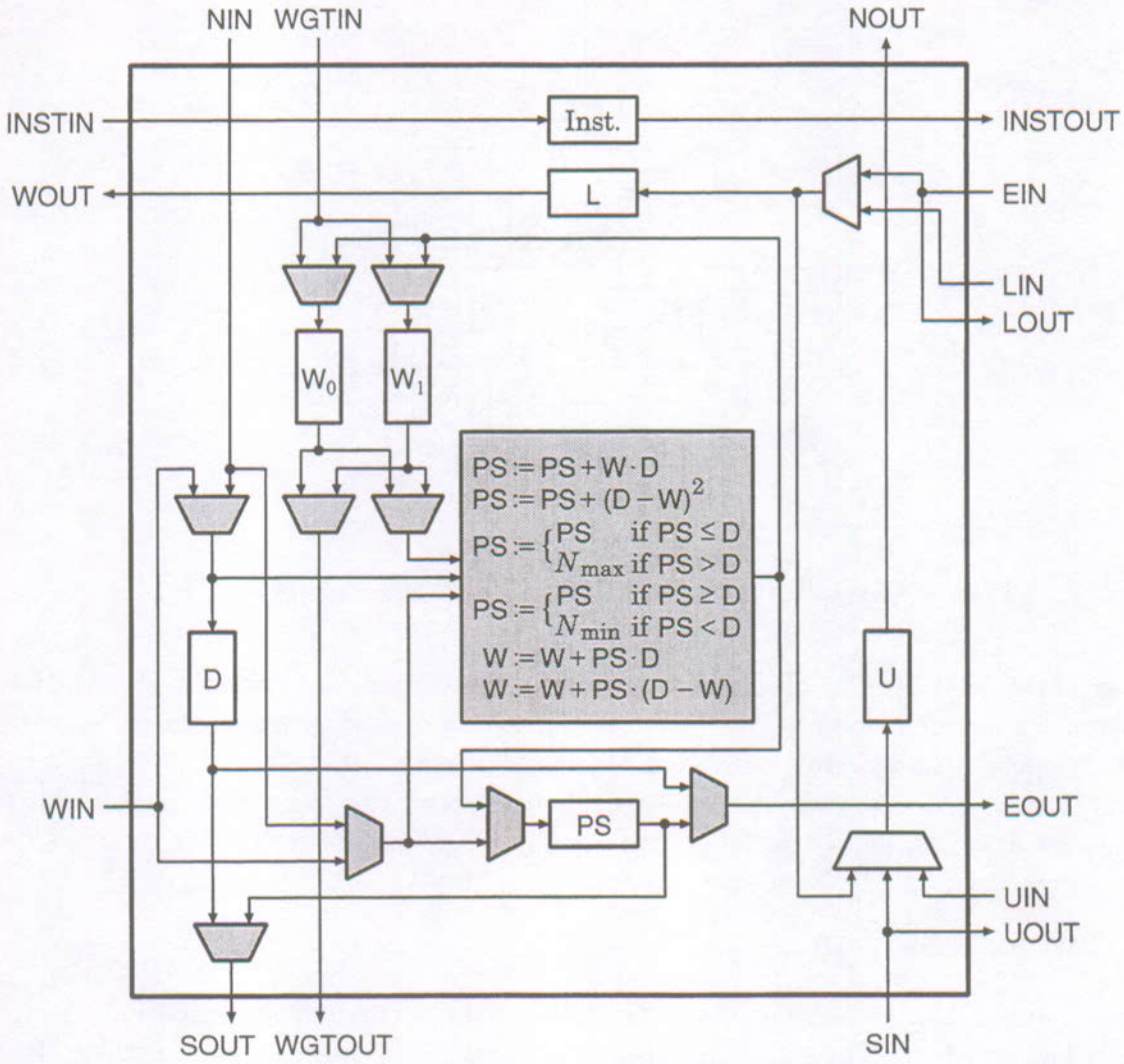


Figure 4.4: *Architecture of the GENES IV diagonal processing element (PE). Units shown shaded are controlled by the instruction stream, while others are globally controlled by external signals.*

4.2.2 Processing-element architecture

Figure 4.4 shows the architecture of a GENES IV diagonal PE. For modularity reasons, non-diagonal PEs are identical, but the input/output ports UIN, LIN, UOUT, and LOUT are left unconnected, and the control of the corresponding multiplexers is hard-wired, as shown in figure 4.5. Each of these PEs contains the following registers:

Instruction register

This register contains the current instruction and controls the units shown shaded in figures 4.4 and 4.5. It is described in section 4.2.8.

Synaptic weight registers: W_0 and W_1

These registers are used to hold the synaptic weights. One of them is selected by the instruction for the current operation. The other is connected to the dedicated path WGTIN–

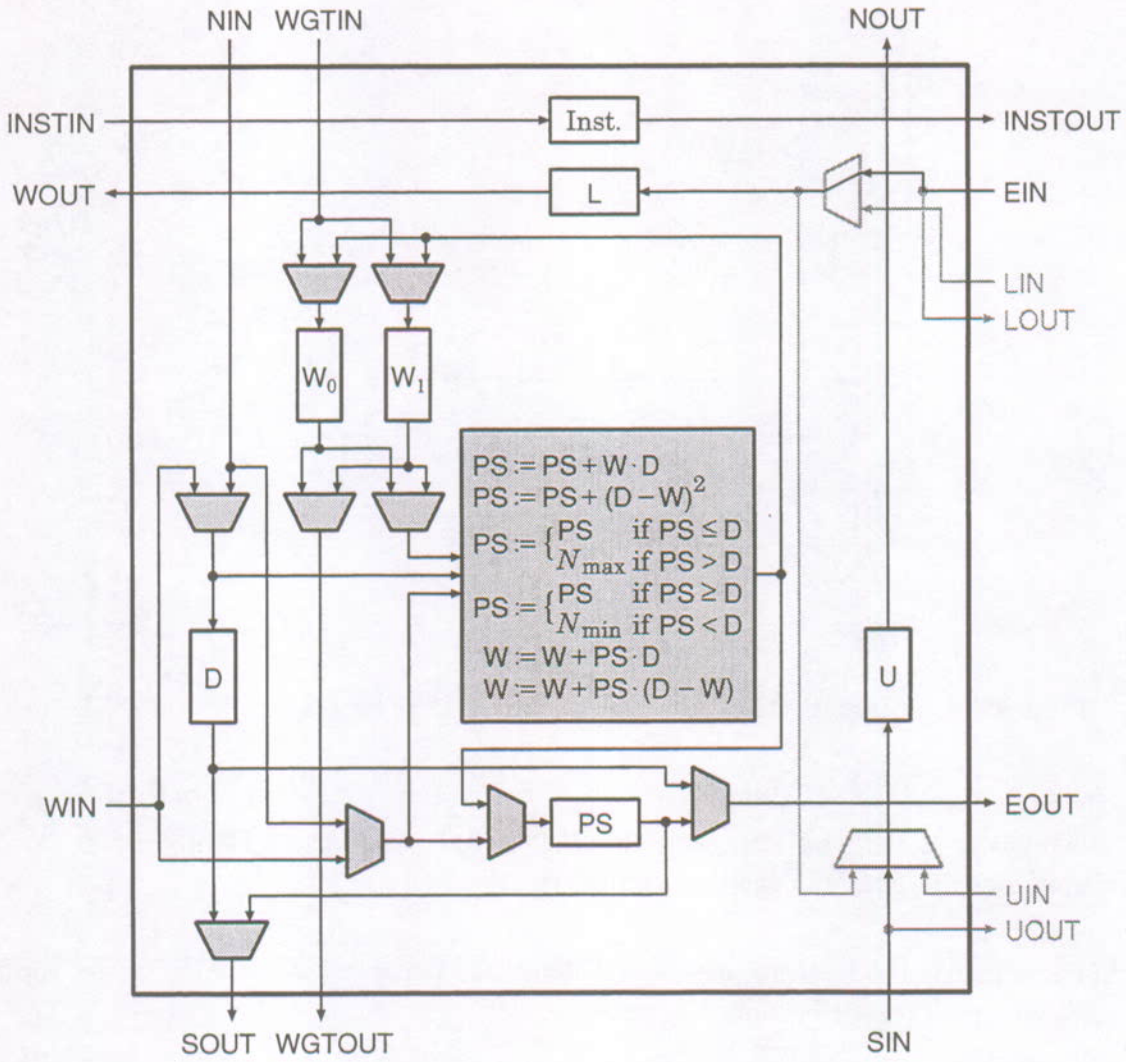


Figure 4.5: *Architecture of the GENES IV non-diagonal processing element (PE). Data paths shown in gray are left unused.*

WGTOUT, and can be used to load and unload sub-matrices in background as described in section 4.2.4.

Data register: D

This register holds the input corresponding to the PE's synaptic weight for the evaluation and learning operations, and one of the vector's elements for the minimum and maximum operations. It is connected on the NIN–SOUT path in normal mode and on the WIN–EOUT path in transpose mode.

Partial sum register: PS

This register carries the partial sum during the evaluation operations. It contains one of the vector's elements for the minimum and maximum operations, and the update coefficient for the learning operations. It is connected on the WIN–EOUT path in normal mode and on the NIN–SOUT path in transpose mode.

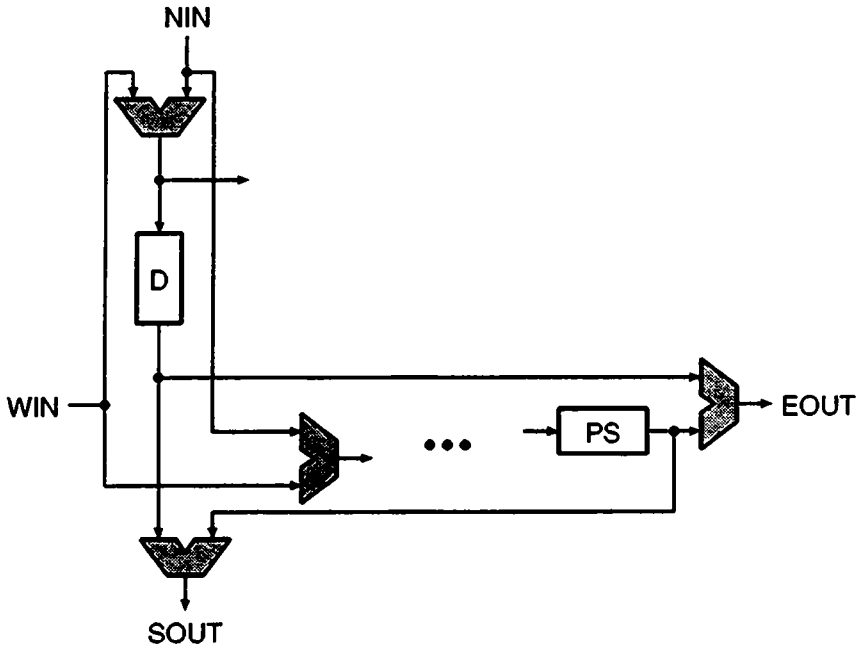


Figure 4.6: *Transposition mechanism of the GENES IV processing element (PE).*

Up register: U

This register is the local storage on the SIN–NOUT path. The incoming values can be retrieved on diagonal PEs and new ones injected.

Left register: L

This register is the local storage on the EIN–WOUT path. The incoming values can be retrieved on diagonal PEs and new ones injected.

4.2.3 Transposition mechanism

The transpose mode, described in section 4.1.2, is implemented by the four multiplexers shown shaded in figure 4.6. The two input multiplexers act as a cross-bar switch between the two input signals NIN and WIN and the D and PS registers. They are controlled by the INTR / $\overline{\text{NL}}$ bit of the instruction register, as described in section 4.2.8. Similarly, the output multiplexers—controlled by the OUTTR / $\overline{\text{NL}}$ bit—form a cross-bar switch between the same registers and the output signals SOUT and EOUT.

When a computation on the transpose matrix begins, the inputs of a PE are first exchanged. However, the values of the registers belong still to the previous computation, and therefore the outputs should be exchanged one macro-cycle later. The same applies when the operation mode reverts to normal.

4.2.4 Matrix-swapping mechanism

When using virtual weight matrices, the physical GENES IV array is time-shared between the sub-matrices. In section 4.1.1, it is shown how the GENES IV operations can be cascaded to yield the desired result with virtual matrices. The learning operations present no difficulties, but

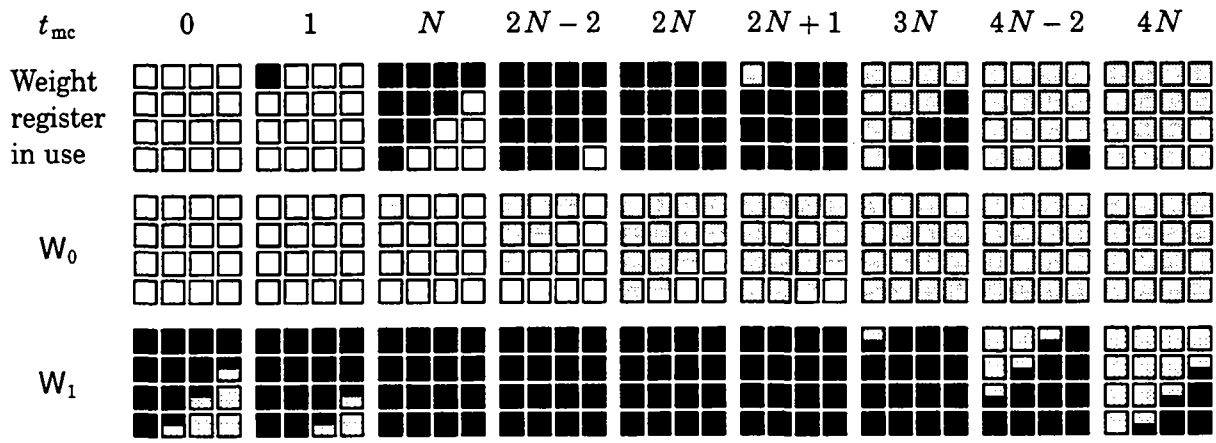


Figure 4.7: *Matrix-swapping mechanism of the GENES IV array (4×4 PEs).*

the evaluation and minimum/maximum operations require the result of a sub-matrix iteration to be re-injected into the next one. Since, for $N \times N$ PEs, the time required to compute an element O_i^h of the result vector (in normal mode) and to re-inject it through the EIN–WOUT path is $2N$ macro-cycles, an efficient sub-matrix swap should take place during this period.

The swapping mechanism is implemented by the two weight registers W_0 and W_1 (see section 4.2.2) and the dedicated WGTIN–WGTOUT path. This path does not appear on figure 4.3, but is parallel to the NIN–SOUT path. Its input ports are located on the north edge and its output ports on the south edge.

Since, over a period of $2N$ macro-cycles, there are only two macro-cycles where all PEs use the same set of weight registers, there is not enough time for a sub-matrix swap to take place in parallel on all columns. Hence, sub-matrices should be loaded or unloaded in a diagonal fashion and each column of PEs should be controlled by a different shift signal. Another possibility would have been to implement a third weight register. However, this solution has been ruled out, to keep the PEs as small as possible. Figure 4.7 shows the matrix-swapping timing over two periods of $2N$ macro-cycles (virtual matrix computation). The first line shows which weight registers and thus which sub-matrix is used. The next two lines show the content of the W_0 and W_1 registers. Half-shaded PEs represent the sub-matrix border in columns currently being shifted.

4.2.5 Arithmetic precision

In the GENES IV array, all quantities are represented as signed integers. Floating-point representations have been discarded for hardware simplicity. Although most of the material presented in this chapter is also valid for floating-point numbers, the arithmetic unit, whose inputs are bit-serial, would be awkward. A floating-point GENES-like array would probably either implement parallel communication or buffer the data both at the input and output.

The two's complement representation for signed numbers has been chosen for compatibility reasons with the host workstation. It should be recalled that this coding yields simpler adders than the sign and magnitude representation, and that multipliers can be as simple as those for unsigned numbers, provided that the operands are sign-extended.

K. Asanović and N. Morgan [ASA91] have studied the impact of reduced precision on the back-propagation algorithm, using the phoneme classifier of a continuous speech-recognition system.

They showed that the training is almost as good with 16-bit weights and 8-bit outputs as with 32-bit floating-point values. To achieve this result, the position of the binary point in the fixed-point representation, or MSB exponent as they call it,[4] was hand-tuned. Not surprisingly, wider representations (18 to 25 bits) are less sensitive to this parameter. They also suggest that the thresholds (which they call biases) should be scaled differently than the weights for improved performance:

> *These histograms show that weight values tend to be clustered around zero in a Gaussian distribution, while bias values tend to be larger and negative.* [...] *These histograms suggested that it would be beneficial to fix the bias exponent separately from the weight exponent.*

As stated in section 1.4, only thresholds treated as weights are considered for the machine. However, a different scaling can be achieved by associating the threshold to a constant non-unity input.

In their simulations, J. Holt and T. Baker [HOL91] show that the back-propagation rule performs almost as well with 16-bit weights and 8-bit inputs/outputs as with 32-bit floating-point numbers. Over five applications, four are labeled of "excellent" and one of "fair." In evaluation mode, they found no significant performance drop if the weights are truncated to 8 bits.

For the Kohonen model, quantization effects have been analyzed by P. Thiran *et al.* [THI94]. They state that a lower bound for the precision is given by $\log_2 (2 \cdot \sqrt{m} + 1)$ for a square mesh of m neurons. Below this value, not even the mapping of a uniform distribution would be possible. However, this result does not take learning into account. Even for this very simple problem, their simulations show that the number of bits should be approximately 1.5 larger to make convergence possible. Though theoretically interesting, this kind of analysis is unfortunately of little help when dealing with real-life problems. However, it can be concluded that 8 bits are too few for large networks ($m > 25^2$).

On the basis of these considerations, the following choices have been made:

- The effective size of the D register during the evaluation operations determines the quantization of inputs. Since a width of 8 bits, representing 256 different values, is often too coarse—especially for the Kohonen algorithm—$N_{\mathrm{D}}^{\mathrm{eval}} = 16$ bits have been chosen. In the Kohonen model the weights belong to the same space as the inputs. Therefore, the size of the W_0 and W_1 registers must be identical: $N_{\mathrm{W}}^{\mathrm{eval}} = 16$ bits.

- The size of the PS register can be chosen as a function of the desired number of inputs $n_{\text{V-lim}}$ that should never generate an overflow. For the matrix-vector product, this value is given by equation (3.14):

$$n_{\text{V-lim}} = 2^{N_{\mathrm{PS}}^{\mathrm{eval}} - N_{\mathrm{W}}^{\mathrm{eval}} - N_{\mathrm{D}}^{\mathrm{eval}} + 1} - 1 \tag{4.19}$$

  Here again, it is desirable to choose a multiple of 8 for the size of the PS register. Since an overflow bit should be implemented, the effective size of this register for the evaluation operations should be 1 bit smaller. For $N_{\mathrm{PS}}^{\mathrm{eval}} = 39$ bits, a neuron with up to $n_{\text{V-lim}} = 255$ inputs will never overflow, which is a fair number for most application.

[4] In section 6.2, data are treated as integers rather than fixed-point numbers, and scaling factors are defined to convert real values into integers. Both techniques are two ways of viewing the same transformation.

For the squared Euclidean distance, the number of inputs $n_{\text{V-lim}}$, that does never generate any overflow, is much smaller:

$$n_{\text{V-lim}} = \left\lfloor \frac{2^{N_{\text{PS}}^{\text{eval}}-1}-1}{\left(2^{N_{\text{W}}^{\text{eval}}-1}+2^{N_{\text{D}}^{\text{eval}}-1}-1\right)^2} \right\rfloor = 64 \text{ inputs}$$

However, this is not a serious problem, because the Kohonen algorithm looks for the smallest Euclidean distance. Hence, the computation is correct if at least one neuron does not overflow.

With the addition of an overflow bit, the size of the PS register becomes $N_{\text{PS}} = 40$ bits.

- In their study on precision for learning [ASA91], K. Asanović and N. Morgan show that 16 bits is the minimum size required for the weights during the learning phase. Since they are using only one application it seems reasonable that some additional precision is necessary for this phase. Moreover, during training, the update quantity $\Delta\mathbf{w}$ decreases toward zero. If the weight precision is too coarse, these amounts may all be rounded to zero after a given number of iterations, preventing any further learning.

  To solve these problems, $N_{\text{W}}^{\text{frc}} = 16$ additional bits are concatenated to the weight registers during the learning operations ($N_{\text{W}}^{\text{lrn}} = 32$ bits). These bits are ignored by the evaluation operations, but they accumulate successive updates eventually producing a "visible change." They may be considered as a fractional part placed on the right of the binary point.

  With the addition of an overflow bit, the size of the weight registers becomes $N_{\text{W}} = 33$ bits.

- For the minimum and maximum operations, the result vector of an evaluation operation should be re-injected both horizontally and vertically:

$$N_{\text{D}}^{\text{min/max}} = N_{\text{PS}}^{\text{min/max}} = N_{\text{PS}}^{\text{eval}} = 39 \text{ bits}$$

  To keep the design orthogonal, the overflow bit is also kept in the D register. Therefore, its size is $N_{\text{D}} = N_{\text{PS}} = 40$ bits.

- The size of the U and L registers is imposed by the wrap-around connections on the edges of the array: $N_{\text{U}} = N_{\text{L}} = 40$ bits.

Since the D register holds the inputs during the learning operations, its effective size must be the same as for the evaluation operations: $N_{\text{D}}^{\text{lrn}} = N_{\text{D}}^{\text{eval}} = 16$ bits. On the other hand, the effective size of the PS register is imposed by the width of the multiplier: $N_{\text{PS}}^{\text{lrn}} = N_{\text{mul}} = 17$ bits.

These results can be summarized by rewriting the PE operations in normal mode (see middle column of table 4.1) in terms of the registers:

$$\text{PS}_{i,j,38..0} := \text{PS}_{i,j-1,38..0} + \text{W}_{i,j,31..16} \cdot \text{D}_{i-1,j,15..0} \tag{4.20}$$

$$\text{PS}_{i,j,38..0} := \text{PS}_{i,j-1,38..0} + \left(\text{D}_{i-1,j,15..0} - \text{W}_{i,j,31..16}\right)^2 \tag{4.21}$$

$$\text{PS}_{i,j,38..0} := \begin{cases} \text{PS}_{i,j-1,38..0} & \text{if } \text{PS}_{i,j-1,38..0} \le \text{D}_{i-1,j,38..0} \\ N_{\max} = 2^{38}-1 & \text{otherwise} \end{cases} \tag{4.22}$$

$$\text{PS}_{i,j,38..0} := \begin{cases} \text{PS}_{i,j-1,38..0} & \text{if } \text{PS}_{i,j-1,38..0} \ge \text{D}_{i-1,j,38..0} \\ N_{\min} = -2^{38} & \text{otherwise} \end{cases} \tag{4.23}$$

$$W_{i,j,31..0} := W_{i,j,31..0} + PS_{i,j-1,16..0} \cdot D_{i-1,j,15..0} \quad (4.24)$$

$$W_{i,j,31..0} := W_{i,j,31..0} + PS_{i,j-1,16..0} \cdot (D_{i-1,j,15..0} - W_{i,j,31..16}) \quad (4.25)$$

where the symbol W represents either of the two weight registers W_0 and W_1. Finally, the communication and computation being bit-serial, the size of the widest register determines the duration of a macro-cycle, that is, 40 clock cycles.

The arithmetic precision of other ANN systems using integer arithmetic is similar to that chosen for the GENES IV circuit. For their chip dedicated to the Hopfield network, M. Weinfeld and his team (see section 2.2.4) code the weights on 9 bits and the neuron states on 1 bit. The computation of the potential is performed by a 12-bit accumulator.

On the Lneuro 1.0 circuit (see section 2.2.5), weights, inputs, and outputs are represented on 8 bits. The weights are extended with 8 additional bits for the update phase. On the Lneuro 2.0 the precision will be increased to 16 bits, and potentials will be internally computed on 32 bits.

The MA16 chip designed for the SYNAPSE-1 machine (see section 2.2.4) uses 16-bit data (weights, inputs, and outputs). However, most of the internal computation (potentials, etc.) is done on 48 bits.

Circuits designed for embedded applications rather than neural computers often feature a smaller precision. For instance, the ANNA chip [BOS92] implements 6-bit weights and 3-bit inputs/outputs.

4.2.6 Arithmetic unit

To compute equations (4.20) to (4.25) three building blocks are required: a multiplier, an adder, and a subtracter. A compact serial-parallel multiplier could have been used for operation (4.20), since one of the operands (the weight $W_{i,j,31..16}$) is local and hence available in parallel form. This is unfortunately not the case for equations (4.21), (4.24), and (4.25), where both operands are serially input. A solution is to buffer the data—that is, to place the D and PS registers—before the arithmetic unit. With such a scheme, the implementation of the saturation mechanism used for the minimum and maximum operations and the handling of overflows (see section 4.2.7) would be far less elegant. Another solution is to place only the D register before the arithmetic unit, leaving the PS register after. With this scheme, the saturation logic would still be easily implementable, but the resulting programming model would be cumbersome. Moreover, an extra register to hold the parallel operand and a parallel subtracter instead of a serial one would be required. This hardware being almost equivalent to the area difference between a parallel-serial and a serial-serial multiplier, the latter has been chosen.

For the GENES IV chip, a serial-serial multiplier should have the following characteristics:

- Operands and result should be two's complement integers.
- Result bits should be output during the same clock cycles that the corresponding operand bits are presented (combinatorial delay).
- A full-precision result should be produced (no truncated or rounded bits).
- The result should be sign-extended to an arbitrarily long size.

Since none of the multipliers found in the literature [LYO76, STR82, GNA83, DAD85] had all these characteristics, a novel design has been developed [IEN94]. Other advantages of this device are

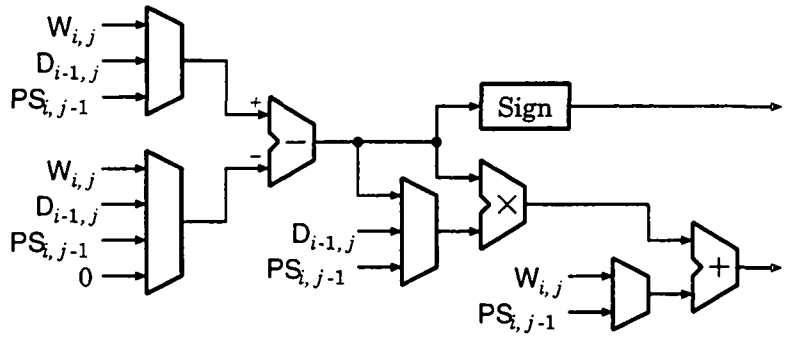


Figure 4.8: *Arithmetic unit of the GENES IV processing element (PE).*

its modularity and the fact that it can be used for both unsigned and two's complement numbers, provided that, in the latter case, operands are sign-extended. A complete description of this multiplier and an example of implementation can be found in an internal report [IEN92]. A 17-bit multiplier has been implemented in each PE, as required for the square in equation (4.21).

Figure 4.8 shows the arithmetic unit implemented on the GENES IV circuit. For simplicity, the operand sign-extension and overflow-handling hardware have been omitted from the figure. The evaluation and learning operations use the data path formed by the subtracter, the multiplier, and the adder. Therefore, in normal mode, equations (4.20), (4.21), (4.24), and (4.25) are computed as:

$$PS_{i,j,38..0} := PS_{i,j-1,38..0} + (W_{i,j,31..16} - 0) \cdot D_{i-1,j,15..0} \quad (4.26)$$

$$PS_{i,j,38..0} := PS_{i,j-1,38..0} + (W_{i,j,31..16} - D_{i-1,j,15..0})^2 \quad (4.27)$$

$$W_{i,j,31..0} := W_{i,j,31..0} + PS_{i,j-1,16..0} \cdot (D_{i-1,j,15..0} - 0) \quad (4.28)$$

$$W_{i,j,31..0} := W_{i,j,31..0} + PS_{i,j-1,16..0} \cdot (D_{i-1,j,15..0} - W_{i,j,31..16}) \quad (4.29)$$

The minimum and maximum operations are implemented by computing $D_{i-1,j,38..0} - PS_{i,j-1,38..0}$ or $PS_{i,j-1,38..0} - D_{i-1,j,38..0}$, respectively, during the 39 first clock cycles. Then, during the last cycle, the PS register is saturated to the largest or smallest representable value if the result of the previous computation is strictly smaller than zero. This saturation presents no problem, because the PS register is placed after the arithmetic unit and hence can be loaded in parallel with the required value. During these operations the output of the multiplier-adder is ignored.

4.2.7 Overflow-handling mechanism

As mentioned in section 3.3.2, the very simple overflow mechanism of previous GENES circuits—which freezes a value once it has overflowed—is ill-suited for the weights. The traditional modulo arithmetic is not satisfactory either, since a very high positive value becomes negative and vice-versa. Therefore, the only solution is to saturate a register on overflow.

For the evaluation operations this mechanism is easily implemented by using the saturation mechanism of the PS register implemented for the minimum and maximum operations. A sticky-bit is also appended to this register, and is propagated with the corresponding partial sum. This bit should initially be cleared. It is then either set or propagated. When set, it means that at

least one overflow occurred. A similar "saturation and sticky-bit" scheme is implemented for each weight register. This bit can be loaded and unloaded together with the corresponding weight.

Besides register overflows described so far, this mechanism is also used to handle multiplier overflows. Being a 17-bit device, the multiplier requires correctly sign-extended operands in the range $[-2^{16}, 2^{16}-1]$. The D register is never checked, because the external hardware is assumed to input a sign-extended 16-bit value. This is not the case for the PS register, which is used as a multiplier operand in the learning operations. When the value $PS_{i,j-1,39..0}$ does not belong to the range $[-2^{16}, 2^{16}-1]$, the weight register is saturated to the largest or smallest representable value according to the sign of $PS_{i,j-1,39..0} \cdot D_{i-1,j,15..0}$ in the case of the Hebbian rule, and to that of $PS_{i,j-1,39..0} \cdot (D_{i-1,j,15..0} - W_{i,j,31..16})$ in the case of the Kohonen rule.

4.2.8 Instruction register

The purpose of the *instruction register* is to control the arithmetic unit (described in section 4.2.6) and the shaded multiplexers of figures 4.4 and 4.5. The other units of are globally controlled on an array or column basis. This device is composed of a shift register and a parallel one. The shift register is used to propagate the instruction from PE to PE in a systolic fashion, while the parallel register holds the current instruction. The value of the former register is copied in the latter at the beginning of each macro-cycle. Each instruction is composed of four independent fields:

Operation code: $OP_{2..0}$
This field encodes the operation executed by the PE:

000: No operation (NOP)
001: Matrix-vector product (scalar product)
010: Squared Euclidean distance
011: Unused (reserved for Manhattan distance)
100: Hebbian learning rule
101: Kohonen learning rule
110: Maximum element of a vector
111: Minimum element of a vector.

Input transpose/normal mode: INTR / $\overline{\text{NL}}$
This bit controls the transpose or normal mode of the input signals NIN and WIN (see section 4.2.3).

Output transpose/normal mode: OUTTR / $\overline{\text{NL}}$
This bit controls the transpose or normal mode of the output signals SOUT and EOUT (see section 4.2.3).

Weight register selection: WGTREG
This bit selects the weight register used for the current operation, the other weight register being used for sub-matrix loading and unloading.

4.2.9 Circuit integration

A VLSI chip, containing 2×2 GENES IV PEs, has been implemented by P. Ienne with standard cells in a $1\,\mu$m CMOS technology. It is composed of 71690 transistors (3179 standard cells) on a die of $6.3 \times 6.1\,\text{mm}^2$, shown in figure 4.9. This circuit has been successfully tested at 20 MHz.

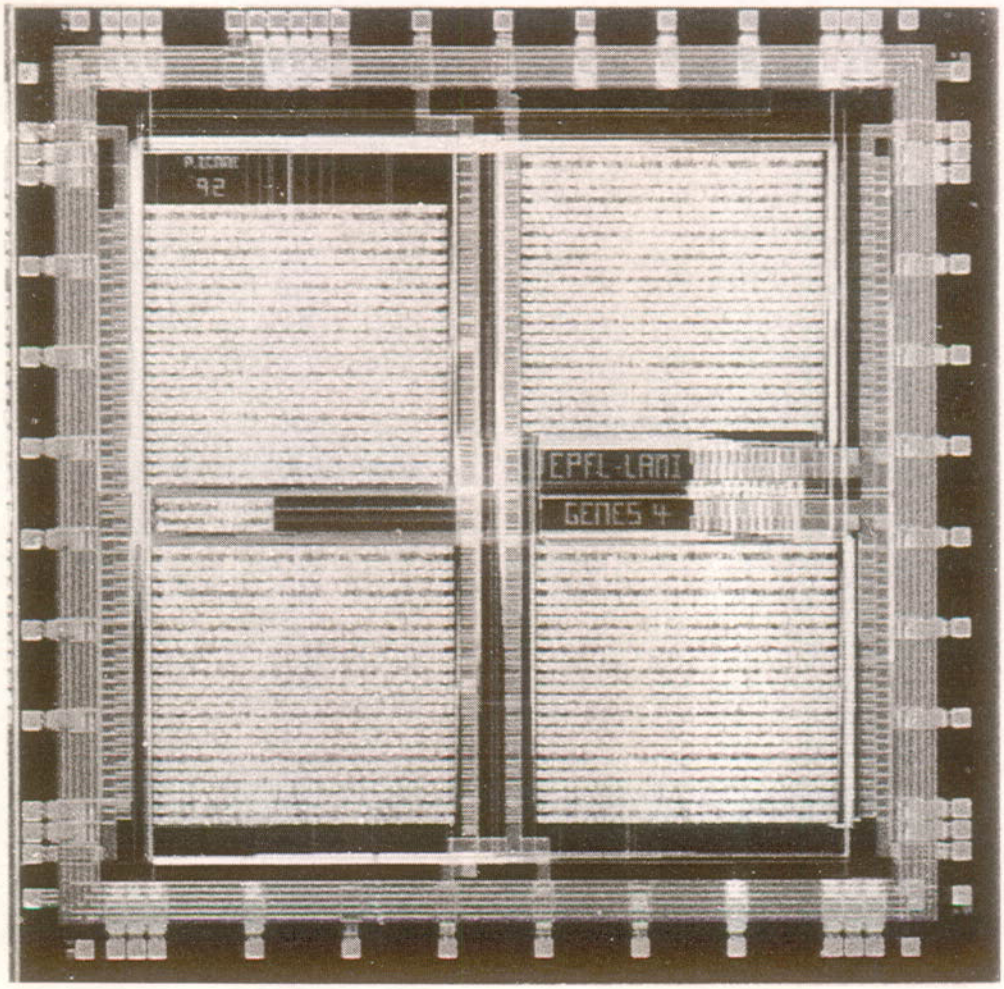


Figure 4.9: *The GENES IV integrated circuit.*

References

[ASA91] K. Asanović and N. Morgan. *Experimental Determination of Precision Requirements for Back-Propagation Training of Artificial Neural Networks.* In U. Ramacher, U. Rückert, and J. A. Nossek (eds.), *Proceedings of the 2nd International Conference on Microelectronics for Neural Networks*, pp. 9–15, Munich (D), October 1991. Kyrill and Method Verlag.

[BLA90] F. Blayo. *Une implantation systolique des algorithmes connexionnistes.* Ph.D. thesis no. 904, EPFL, Lausanne (CH), 1990.

[BOS92] B. E. Boser, E. Sackinger, J. Bromley, Y. LeCun, and L. D. Jackel. *Hardware Requirements for Neural Network Pattern Classifiers: A Case Study and Implementation. IEEE Micro*, 12(1):32–40, February 1992.

[DAD85] L. Dadda. *Fast Multipliers for Two's-Complement Numbers in Serial Form.* In *Proceedings of IEEE 7th Symposium on Computer Arithmetic*, pp. 57–63, 1985.

[GNA83] R. Gnanasekaran. *On a Bit-Serial Input and Bit-Serial Output Multiplier. IEEE Transactions on Computers*, C-32(9):878–880, September 1983.

[HOL91] J. L. Holt and T. E. Baker. *Back Propagation Simulations Using Limited Precision Calculation.* In *Proceedings of the International Joint Conference on Neural Networks*, vol. II, pp. 121–126, Seattle, WA (USA), July 1991. IEEE, INNS.

[IEN92] P. Ienne and M. A. Viredaz. *Bit-Serial Input and Output Multipliers and Squarers.* Technical report no. 92/7, DI, EPFL, Lausanne (CH), October 1992.

[IEN93] P. Ienne and M. A. Viredaz. *GENES IV: A Bit-Serial Processing Element for a Multi-Model Neural-Network Accelerator.* In L. Dadda and B. Wah (eds.), *Proceedings of the International Conference on Application-Specific Array Processors*, pp. 345–356, Venice (I), October 1993. Euromicro, IEEE, IEEE Computer Society Press.

[IEN94] P. Ienne and M. A. Viredaz. *Bit-Serial Multipliers and Squarers. IEEE Transactions on Computers*, 1994. To appear.

[KNU68] D. E. Knuth. *Fundamental Algorithms*, vol. 1 of *The Art of Computer Programming.* Addison-Wesley Series in Computer Science and Information Processing. Addison-Wesley, Reading, MA (USA), 1968.

[LEH93] C. Lehmann, M. Viredaz, and F. Blayo. *A Generic Systolic Array Building Block for Neural Networks with On-Chip Learning. IEEE Transactions on Neural Networks*, 4(3):400–407, May 1993. Special issue on neural network hardware.

[LYO76] R. F. Lyon. *Two's Complement Pipeline Multipliers. IEEE Transactions on Communications*, COM-24(4):418–425, April 1976.

[STR82] N. R. Strader and V. T. Rhyne. *A Canonical Bit-Sequential Multiplier. IEEE Transactions on Computers*, C-31(8):791–795, August 1982.

[THI94] P. Thiran, V. Peiris, P. Heim, and B. Hochet. *Quantization Effects in Digitally Behaving Circuit Implementations of Kohonen Networks. IEEE Transactions on Neural Networks*, 5(3):450–458, May 1994.

5

With an anxiety that almost amounted to agony, I collected the instruments of life around me, that I might infuse a spark of being into the lifeless thing that lay at my feet. It was already one in the morning; the rain pattered dismally against the panes, and my candle was nearly burnt out, when, by the glimmer of the half-extinguished light, I saw the dull yellow eye of the creature open; it breathed hard, and a convulsive motion agitated its limbs.

Mary W. Shelley, *Frankenstein*

The MANTRA I Machine

The *MANTRA I* machine [VIR92, VIR93B, VIR93C] is a neural computer based on the GENES IV systolic array presented in chapter 4. A first prototype has been built and tested. The analysis presented in this thesis is based on this system. A second version is currently under test. The latter consists in a re-design of the control board to solve electrical problems found in the former. They are architecturally equivalent, since only a few conceptual details have been improved in the second implementation.

The aim of this chapter is to discuss the architecture of the MANTRA I machine. Hardware details (memory map, control and status registers, connectors, etc.) are provided by a technical report [VIR93A] along with a copy of the schematics and the listings of the PAL/EPLD source files.

Like any SIMD system, the MANTRA I machine consists of a *parallel* or *SIMD module* (see section 5.1) and a *control module* (see section 5.2). Figure 5.1 presents the architecture of the machine, the dashed box representing the SIMD module. Since a macro-cycle lasts 40 clock cycles, all parallel units can be shared among 40 rows or columns. Hence, the maximum GENES IV array has been chosen as $40 \times 40 = 1600$ PEs. In principle, larger machines could be built by simply duplicating these units. The clock frequency of the array is 10 MHz.

5.1 The SIMD module

As described in section 4.1, the GENES IV array is used for $O(m \cdot n)$ or $O(m^2)$ operations. The rest of the computation (i.e., $O(m)$ operations) is performed by dedicated hardware units in the SIMD module: the delta, sigma, and function-of-Y units, described in sections 5.1.1, 5.1.2, and 5.1.3. The rest of the SIMD module is composed of the storage components needed to sustain the required input/output streams, and of parallel-to-serial and serial-to-parallel converters.

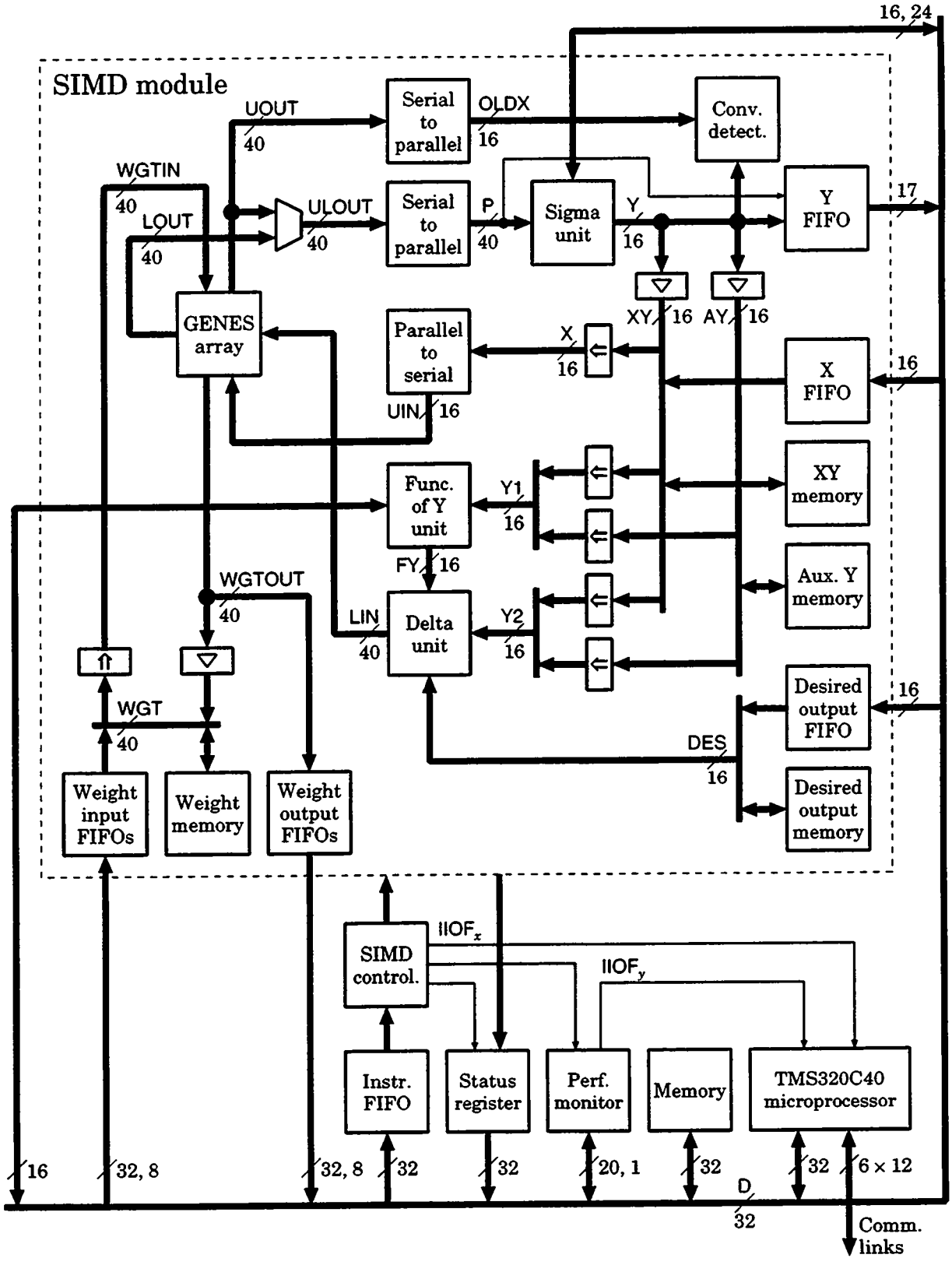


Figure 5.1: *Architecture of the MANTRA I machine.*

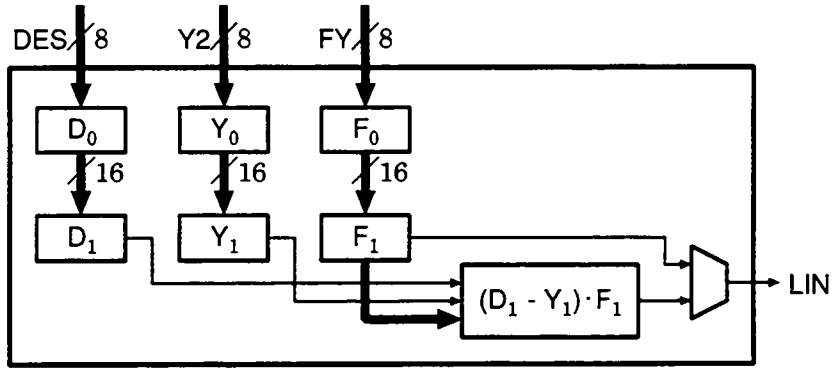


Figure 5.2: *Architecture of the delta-unit cell (16-bit configuration).*

5.1.1 The delta unit

The *delta unit* is dedicated to the computation of the error vector $\vec{\varepsilon}$ and error signal vector $\vec{\delta}$ for the supervised learning rules: Perceptron (see section 1.6), delta (see section 1.7), and back-propagation (see section 1.8). The most general form of this operation appears in the delta rule:

$$\vec{\delta}(t,s) = \vec{\varepsilon}(t,s) \circ \sigma'(\vec{\mathbf{p}}^*(t,s)) = \left(\vec{\mathbf{d}}(s) - \vec{\mathbf{y}}(t,s)\right) \circ \sigma'(\vec{\mathbf{p}}^*(t,s)) \tag{5.1}$$

where the symbol $\circ$ represents the *Hadamard product* or term-to-term matrix multiplication. For the Perceptron rule, described by equation (1.28), only the error vector $\vec{\varepsilon}$ is required. Therefore, it is computed as: $\vec{\varepsilon}(t,s) = (\vec{\mathbf{d}}(s) - \vec{\mathbf{y}}(t,s)) \circ \vec{\mathbf{1}}$, where $\vec{\mathbf{1}}$ refers to a vector of ones. On the last layer of the back-propagation rule the error signal vector $\vec{\delta}^{[L]}$, given by equation (1.36), is computed in an identical way to that of the delta rule. For other layers, equation (1.37) is decomposed into the matrix-vector product $\vec{\varepsilon}^{[k]}(t,s) = \mathbf{w}^{[k+1]^{\mathrm{T}}}(t) \cdot \vec{\delta}^{[k+1]}(t,s)$ executed on the GENES IV array, and the Hadamard product $\vec{\delta}^{[k]}(t,s) = \vec{\varepsilon}^{[k]}(t,s) \circ \sigma'(\vec{\mathbf{p}}^{[k]}(t,s))$ performed by the delta unit.

The delta unit is implemented as an additional column of dedicated cells connected to the LIN inputs of the GENES IV array. Such a cell is shown in figure 5.2. A dedicated VLSI chip, called *GACD1*,[1] has been integrated [IEN92]. At design time, it was not yet clear whether the data representation in the MANTRA I machine would have been 8 or 16 bits. Therefore, a circuit with six 8-bit cells cascadable into three 16-bit ones has been implemented. Since the three input busses are 8 bits wide, the most-significant byte of the 16-bit data is first stored into the D_0, Y_0, and F_0 registers, followed by the least-significant one a clock cycle later. This has the unfortunate consequence that some additional logic is required to convert 16-bit quantities into two 8-bit words. The input busses are shared by all cells of a chip.

At the start of a macro-cycle, operands are transferred into the D_1, Y_1, and F_1 registers. The serial result is output in 33 clock cycles and then sign-extended. The arithmetic unit consists of a serial subtracter and a parallel-serial multiplier. When the computation of equation (5.1) is not required, this unit can be bypassed and the delta unit acts as a parallel-to-serial converter.

The GACD1 circuit has been used to test the 1 μm CMOS technology and standard-cell library used for the GENES IV chip. The die contains 16896 transistors (769 standard cells) and measures $3.2 \times 3.2\,\mathrm{mm}^2$. This chip has been successfully tested at 20 MHz.

[1] GACD1 is the acronym of *GENES Auxiliary Circuit for the Delta Rule, 1st version.*

5.1.2 The sigma unit

The main purpose of the *sigma unit* is to compute the activation function σ of ANNs. Since the outputs of the GENES IV array are $N_{PS} = 40$ bits long (including an overflow bit) and the result of the sigma unit must fit on $N_Y = 16$ bits, this unit performs also data-format conversion and thus is always used when data are retrieved from the array.

The most general approach to the computation of the activation function is to use a look-up table. Since a 16-bit table with 2^{40} entries (i.e., 2 Tbyte) is obviously unrealistic, the classical solution is to reduce the table size by truncating the LSBs. However, this solution maps very poorly the steep part of typical activation functions (see appendix A.1). Moreover, some algorithms require the implementation of the identity function (with saturation to the smallest and largest representable output values), whose implementation is impossible if the LSBs are truncated. Therefore, another solution, taking into account the shape of typical activation functions, should be found.

A first possibility is to take advantage of the fact that activation functions are monotonic in all algorithms presented in chapter 1. When using integer arithmetic, this class of functions can be viewed as a sum of $2^{N_Y} - 1$ step functions. A non-decreasing function is expressed as:

$$\sigma(v) = -2^{N_Y-1} + \sum_{k=1}^{2^{N_Y}-1} u_{\geq}(v - \Theta_k) \tag{5.2}$$

and a non-increasing one as:

$$\sigma(v) = 2^{N_Y-1} - 1 - \sum_{k=1}^{2^{N_Y}-1} u_{\geq}(v - \Theta_k) \tag{5.3}$$

If the left asymptote β_{left} of the activation function σ is larger than -2^{N_Y-1} (or smaller than $2^{N_Y-1} - 1$), the first $2^{N_Y-1} + \beta_{\text{left}}$ thresholds (respectively the first $2^{N_Y-1} - \beta_{\text{left}} - 1$ ones) should be set to $\Theta_k = -2^{N_{PS}^{eval}-1}$. For the right asymptote β_{right}, the problem is slightly different. If it is different from $2^{N_Y-1} - 1$ (or -2^{N_Y-1}), the last $2^{N_Y-1} - \beta_{\text{right}} - 1$ thresholds (respectively the last $2^{N_Y-1} + \beta_{\text{right}}$ ones) should be left unused: $\Theta_k \geq 2^{N_{PS}^{eval}-1}$. This can be achieved either by encoding the thresholds on $N_{PS}^{eval} + 1$ bits or by using an enable bit. Concerning the quantity of information, both solutions are equivalent. Compared with a full look-up table, the storage ratio is:

$$\frac{(2^{N_Y} - 1) \cdot (N_{PS}^{eval} + 1)}{2^{N_{PS}^{eval}} \cdot N_Y} = 2.9 \cdot 10^{-7} = 0.29\,\text{ppm}$$

Although huge, this gain deserves a further discussion. Though much smaller than for a full look-up table, the required storage (320 Kbyte) is still big, especially because it should be implemented in ASICs where the storage is very expensive. On the other hand, solutions involving look-up tables (even with truncation) can use commercial RAM chips designed with an *ad hoc* technology. There are also other problems, for instance, it is not clear how the overflow bit should be handled. Several architectures using parallel comparators have been evaluated. The conclusion is that a realistic solution could be found for 8-bit outputs, but is almost impossible when 16 bits should be produced.

From this discussion, it can be concluded that commercial RAM chips should be used. An

analysis of typical activation functions shows that high precision is only required for a small part of the input range. The chosen implementation is a double look-up table, consisting of a *coarse-grain table* mapping the whole input space at reduced precision, and a *fine-grain table* mapping a small window at high precision.

The loading time of these tables being rather long, having to initialize them too often is not desirable. Since some algorithms need to switch back and forth from an output function to another, two sets of coarse- and fine-grain tables should be implemented.[2] Since the largest fast-enough static RAMs available at design time have 18 address lines, each table covers 2^{17} entries. The basic idea is to have the MSBs of the potential connected to the coarse-grain table while the LSBs are input to the fine-grain table. The latter is selected when the potential's *prefix* — that is, the remaining bits — is equal to a predetermined value.

This basic scheme has been slightly modified to improve its effectiveness. A drawback is that the fine-grain table can only be aligned on multiples of its size. This may create problems if the activation function requires a high precision around one of these boundaries. Therefore, a 1-bit overlap between the comparator and the fine-grain table input has been implemented. The fine-grain table is then selected when the prefix matches any of two consecutive values. With this mechanism, the fine-grain table can be aligned on multiples of half of its size.

Another improvement is the possibility to shift the input of the fine-grain table, from 0 to 7 bits towards the MSB (as programmed by the $\mathsf{SFGSH}_{2..0}$ field of the configuration register, described in section 5.2.2). This zooming mechanism makes it possible to increase the size of the fine-grain window, at the price of a reduced precision. Similarly, the machine also offers the possibility to shift the input of the coarse-grain table ($\mathsf{SCGSH}_{2..0}$ field of the configuration register). This feature can be used when the number of inputs is small and the potential is known to be bounded in the range $[-2^{38-\mathsf{SCGSH}_{2..0}},\ 2^{38-\mathsf{SCGSH}_{2..0}} - 1]$.

Figure 5.3 shows an equivalent block diagram of the resulting architecture. The actual implementation takes advantage from the fact that the potential $\mathsf{P}_{39..0}$ is output bit-serially. The shift registers of figure 5.3 and the serial-to-parallel converter associated with the delta unit can be grouped in a single device. This converter generates directly the values $\mathsf{PH}_{23..0}$ and $\mathsf{PL}_{16..0}$. Figure 5.4 summarizes the relation between the potential and the sigma unit.

5.1.3 The function-of-Y unit

The delta and back-propagation rules, which use a gradient descent to minimize the quadratic error $\vec{\xi}$, require not only the computation of the output vector $\vec{\mathbf{y}}^{*[k]}(t,s) = \sigma(\vec{\mathbf{p}}^{*[k]}(t,s))$, but also that of its derivative $\sigma'(\vec{\mathbf{p}}^{*[k]}(t,s))$. At first sight, a second piece of hardware, identical to the sigma unit, is required for this operation. Moreover, the resulting value should be stored together with the output in memory.

As mentioned in section 5.1.2, the activation function σ is always monotonic, and an inverse function σ^{-1} can be defined.[3] Therefore, the derivative can be expressed as a function of the output: $\sigma'(\vec{\mathbf{p}}^{*[k]}(t,s)) = \sigma'(\sigma^{-1}(\vec{\mathbf{y}}^{*[k]}(t,s)))$. In this way, the computation can be implemented by a 16-bit look-up table with 2^{16} entries. Moreover, there is no need to store the derivative. Four such tables are implemented in the *function-of-Y unit*. The current one is selected by the $\mathsf{FYS}_{1..0}$ field of the configuration register (see section 5.2.2).

[2] The current set is selected by the SIGMAS bit of the configuration register (see section 5.2.2).

[3] This function is usually not defined in $\mathbb{R}$, but only in the range $[\beta_{\text{left}},\ \beta_{\text{right}}]$, defined by the asymptotes of σ.

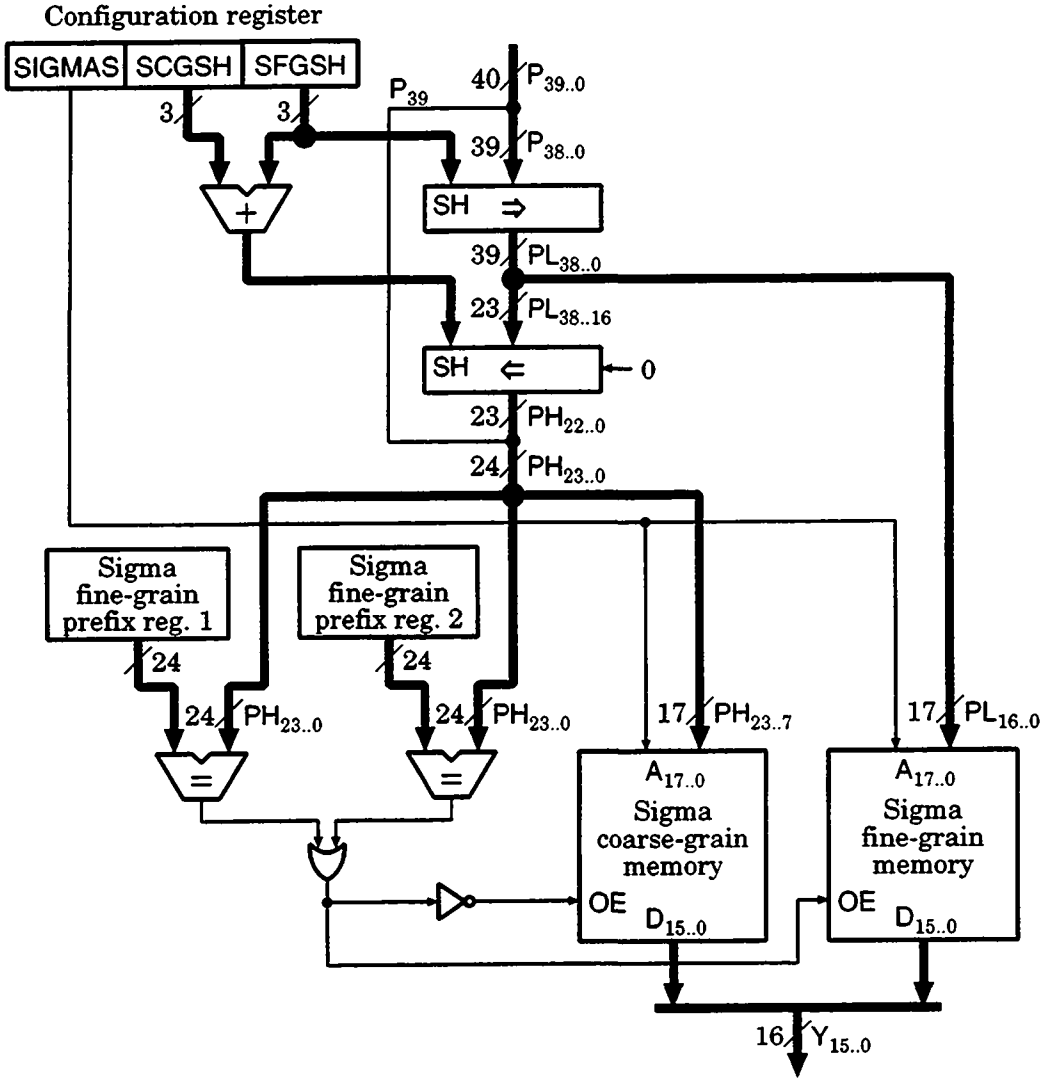


Figure 5.3: *Functional structure of the sigma unit.*

This unit serves also other purposes. In particular, it is used to implement the learning coefficient α, as described in chapter 6. In this case, the table outputs a value $f_i^{[k]}$ proportional to $\alpha(t) \cdot \sigma'(\sigma^{-1}(y_i^{[k]}(t,s)))$.

5.1.4 The data units

ANNs deal with different types of data: weights, inputs, outputs, and desired outputs. Since the output vector of an iteration is often the input vector of the next one, inputs and outputs should be stored together. On the MANTRA I machine, three independent *data units* are used for the weights, the inputs/outputs, and the desired outputs. The problems, due to the lack of a link between the $Y_{15..0}$ bus (used for outputs) and the $DES_{15..0}$ bus (used for desired outputs), are analyzed in sections 6.1.2 and 8.1.1.

Though similar, these units fulfill different requirements: it should be possible to perform a write and a read access per clock cycle in the weight unit, a write and two read accesses in

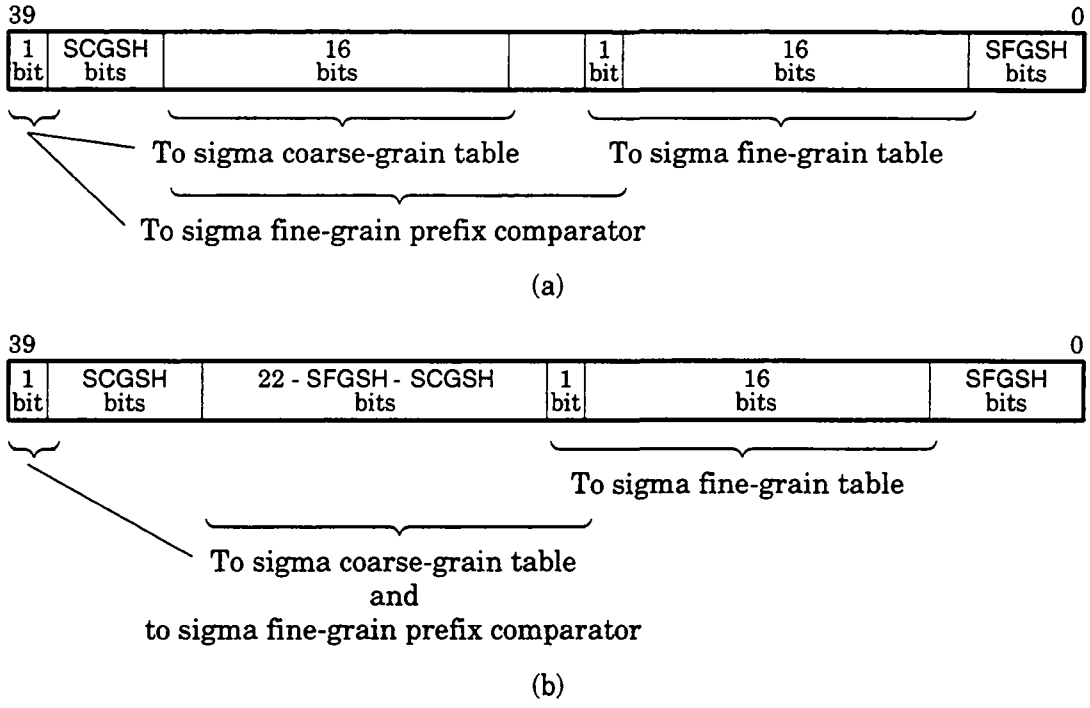


Figure 5.4: *Relation between the potential* $P_{39..0}$ *and the sigma unit (i.e., tables and comparator).* (a) *For* $SCGSH_{2..0} + SFGSH_{2..0} < 7$. (b) *For* $SCGSH_{2..0} + SFGSH_{2..0} \geq 7$.

the input/output unit, and a read access in the desired output unit. Furthermore, the control microprocessor should be able to enter or retrieve data from these units without affecting the SIMD module's computation. Four solutions, involving static RAMs, FIFOs, dual-port RAMs, and video RAMs [NIC88] have been evaluated, before committing to a combination of static RAMs and FIFOs.

The simplest implementation is to use static RAMs and to multiplex the different accesses in time. This implies that up to three accesses should be performed each clock cycle: a write and a read access for the GENES IV array, and a microprocessor access.[4] A unique RAM bank in the input/output unit would even require four accesses. This number can be reduced to three by using two memory banks. Even at the target speed (10 MHz), this would require very fast and hence small memory chips. Moreover, the SIMD and control modules running on different clocks, a synchronization mechanism would be required.

The second solution features FIFOs for each data stream. This implementation would result in a very simple hardware, since FIFO chips with asynchronous ports are broadly available and no address generation is required. However, being in charge of storing temporary results and re-injecting them when appropriate, the microprocessor may become a bottleneck. Since it is very hard to evaluate the microprocessor's workload at design time, this solution has also been ruled out.

The third possibility is to use dual-port static RAMs. One of their two asynchronous busses could be used by the SIMD module, while the other could be connected to the microprocessor. This configuration requires only two accesses per clock cycle on the SIMD module's port, what

[4] Only two accesses are required for the desired output unit.

presents no problem with current chips. However, dual-port RAM chips have a rather small capacity (typically 4 Kbyte at design time, i.e., eight times less than static RAM chips of comparable technology) and come in huge packages. Therefore, this solution has also been discarded.

Other very attractive devices are video RAMs. At first sight, it appears that their parallel bus could be connected to the microprocessor, while their serial port suits the SIMD module well. However, due to the parallel nature of the sigma and function-of-Y units (see sections 5.1.2 and 5.1.3), no scheme could be found that avoids the use of even a single converter. Other problems come from the RAM-SAM transfers of video RAMs. In a software implementation, a fine synchronization is required between the microprocessor and the SIMD module, while in a hardware solution, a non-trivial arbiter would be required to resolve collisions between RAM-SAM transfers and microprocessor accesses. Therefore, video RAMs have not been adopted for the machine. Thanks to their large capacity, they remain interesting for future designs, especially if the sigma unit is serially implemented.

The chosen implementation is a mixture of the first two solutions. The weight unit consists of an input FIFO, a memory, and an output FIFO connected as shown in figure 5.1. During the first half of each clock cycle, weights can be loaded in the GENES IV array from the input FIFO or from memory. When the FIFO is used, it is also possible to write simultaneously a data in memory for a future use. During the second half of each clock cycle, weights retrieved from the array can be written into the output FIFO or in memory. Since both devices are independently controlled, it is possible to use none of them or both at the same time. This scheme makes it possible to keep all temporary results in memory (if it is large enough) and to output final results and snapshots of the ANN state.

The input/output unit is identical, except that there are two memory banks: the *XY memory* and the *auxiliary Y memory* (see figure 5.1). During the first phase, data on the $XY_{15..0}$ bus are read either from the *X FIFO* or from the XY memory (or copied from the former to the latter), while data on the $AY_{15..0}$ bus are read from the auxiliary Y memory. During the second phase, data can be independently written in the *Y FIFO* or in the two memory banks.

The desired output unit is somewhat simpler. Since no result is ever produced, there is only one phase and no output FIFO.

The data are stored in parallel form in the input/output and desired output units, and in serial form in the weight unit. Hence, the microprocessor must store and retrieve data from the weight FIFOs in a pre-arranged form [VIR93A], to convert them in serial form and to account for their "diagonal" loading (see section 4.2.4). Since the chosen FIFO chips are 9 bits wide, some extra bits are available. One of them is used to propagate, in the Y FIFO, the overflow bit associated with the potential.

5.1.5 The convergence unit

In the three previous implementations of the GENES array, described in chapter 3, a built-in mechanism detects the convergence of a pattern in a recurrent network. This mechanism has been integrated in the GENES cells, because the computation of the activation function takes place on the east edge. With the MANTRA I machine, the serial-to-parallel converter should be placed on the diagonal of the GENES IV array, because data are input and output there. Since the sigma unit is based on look-up tables and hence is inherently bit-parallel, it could not be placed on the east edge (unless an additional pair of converters would be implemented). Therefore, the

convergence detection mechanism has been removed from the GENES IV PEs and placed after the sigma unit.

The *convergence unit* consists of a 16-bit comparator and a 7-bit counter. The counter is incremented each time a valid output $Y_{15..0}$ is equal to the corresponding input $OLDX_{15..0}$. The counter's value can be latched as a part of the status register by a special instruction issued by the control microprocessor (see section 5.2.2). This operation also resets the counter.

5.2 The control module

Any SIMD machine requires a controller or instruction unit to dispatch the common instruction stream (see section 2.1.1). The classical approach is to use a sequencer or a microcontroller, as for instance in the Connection Machines CM-1 and CM-2 [HIL85, TM87]. Another possibility is to make use of a complete SISD system, which can execute its own code as well as dispatch instructions to the parallel module. This solution, implemented for example on the MasPar MP-1 [BLA90, NIC90], lets the serial part of an application run on the controller. The latter solution has been retained for the MANTRA I machine, because of the added flexibility and the ease of implementation of microprocessor-based systems. This flexibility is especially important to implement the communication with a host computer, other MANTRA I machines, or a number cruncher. Another advantage of this approach is that a commercial microprocessor can be programmed in a high-level language like C, using a user-friendly environment (debugger, etc.), while development tools for sequencers are often less sophisticated.

In comparison, a hybrid solution has been chosen for the SYNAPSE-1 machine (see section 2.2.4), featuring both a sequencer and a microprocessor. This combines the speed of a sequencer and the versatility of a microprocessor to handle the communication and pre- or post-process the data.

5.2.1 The microprocessor

The choice of the control microprocessor is not especially critical, since the only requirements are that this unit should not become a bottleneck and should offer a convenient platform to the programmer. High-end RISC or CISC microprocessors would have performed well as far as speed is concerned. However, they provide a lot of hardware that is of little use in an embedded system (memory management units, privileged modes, etc.), but often do not offer functions that are useful to controllers (DMA channels, communication ports, etc.). On the other hand, microcontrollers fulfill these requirements but are often slower, due to their smaller word size. Finally *digital signal processors (DSPs)* are particularly well adapted to the type of pre- or post-processing encountered with ANNs.

As stated above, one of the main tasks of the processor, beside controlling the SIMD module, is to handle the communication with the external world. This makes microprocessors with integrated communication links very attractive. Among those is the Transputer family. The IMS T800 microprocessor [IMS88] is already an old design with slow channels according to today's standards (aggregate bandwidth of 4×2.4 Mbyte/s in bi-directional mode), and the new IMS T9000 circuit (4×20 Mbyte/s) [IMS93] was not available when the project started. The iWarp chip from Intel (see section 2.1.3) has much faster links (8×40 Mbyte/s, 4 input and 4 output ports) but Intel decided to commercialize only iWarp-based machines and not to support the circuit by itself. Finally, the

TMS320C40 microprocessor from Texas Instruments offers six 20-Mbyte/s channels. This processor has been chosen because of its DSP nature and its built-in communication links. Its main features are:

- Six bi-directional communication ports (8 bits, 20 Mbyte/s).
- Six general-purpose DMA channels.
- Two memory interfaces (busses).
- Two sets of hand-shake signals ($\overline{\text{STRB}}$, R / $\overline{\text{W}}$, $\overline{\text{RDY}}$, etc.) per bus.
- Internal (on-chip) 512-byte instruction cache.
- Two internal (on-chip) 4-Kbyte RAM banks.
- Internal (on-chip) 16-Kbyte ROM, containing a boot loader.
- Simultaneous accesses to three units among one of the communication ports, the two busses, the instruction cache, the two internal RAM banks, and the internal ROM. (Two accesses per clock cycle to the internal RAM banks.)

Since a further analysis of the microprocessor's architecture is not directly relevant to this thesis, the interested reader is referred to the user's manual [TI91].

A TMS320C40-based parallel system has been developed by C. Marguerat at the Microcomputing Laboratory [MAR93]. For this project, a processor board running at 20 MHz has been designed as a versatile platform for hardware development [MAR92]. This board is used as the mother-board of the MANTRA I machine. The choice of the TMS320C40 microprocessor makes it possible to connect the machine to any system using this chip, in particular the above-mentioned parallel computer.

5.2.2 The SIMD interface

The microprocessor controls the SIMD module by dispatching instructions to the *SIMD controller*, which generates all signals required by the parallel module of the machine. The MANTRA I machine follows the *very long instruction word (VLIW)* philosophy. This means that different fields of the instruction word are used to control the different units of the machine. Hence, resource parallelism must be explicitly controlled by software. The *instruction FIFO* serves as an elastic buffer. If this FIFO becomes empty, the SIMD module is frozen until a new instruction is available.

As discussed in section 5.1.4, FIFOs are also used to transfer data from the microprocessor to the SIMD module and vice-versa. So, during computation, the SIMD module and the controller communicate only through asynchronous FIFOs, and the two modules of the machine can run on separate clock signals with no frequency or phase relationship. Although the sigma and function-of-Y units (see sections 5.1.2 and 5.1.3) are directly interfaced to the microprocessor, it is still possible to have different clock signals, because these units are only initialized when the SIMD module is inactive (frozen).

Finally, the *status register* provides a feed-back from the SIMD module to the microprocessor. It gives the state of the parallel module (running or frozen), the status of each FIFO (empty, full, or half-full), and the convergence unit's result. All these signals are double-synchronized to avoid metastability problems.

The SIMD controller and the instruction FIFO

The instruction FIFO is a 36-bit write-only queue used to dispatch instructions to the SIMD module of the MANTRA I machine. Four address lines are stored into the FIFO together with the 32 data bits. These signals address 13 different registers (out of the 16 possible ones) that are all queued in the same device:

- Instruction register.
- Instruction register (interrupt).
- Instruction register (CCNTOUT load).
- Instruction register (interrupt and CCNTOUT load).
- Input/output counter register.
- Configuration register.
- XY memory input address register.
- XY memory output address register.
- Auxiliary Y memory input address register.
- Auxiliary Y memory output address register.
- Desired output memory address register.
- Weight memory input address register.
- Weight memory output address register.

Any of the first four registers addresses the actual VLIW, while the nine others may be considered as "command modifiers" or "auxiliary data," such as configuration information or addresses. They are referred to as *auxiliary registers*.

Instructions are executed on a macro-cycle basis. The first instruction field is a counter that makes it possible to repeat the same instruction up to 128 times. The four instruction registers have identical fields. The difference between them, is the action(s) taken at the end of the instruction, that is, after it has been repeated as many times as programmed. The "interrupt" versions activate an interrupt signal for the TMS320C40 microprocessor.[5] Similarly, the "CCNTOUT load" versions latch the value of the convergence counter into the corresponding field of the status register.

The auxiliary registers are pre-processed while an instruction is executed or the machine is idle, becoming active with the next instruction. Any number of auxiliary registers can be accessed between two instructions. Only the last access to a given register is effective. The relative order between auxiliary register accesses is irrelevant. Since each instruction lasts at least 40 GENES IV clock cycles (repeat counter set to 1) and since auxiliary registers are pre-processed in 2 clock cycles, there is enough time to access each of them without performance penalty.

The *input/output counter register* implements three counters defining how many words are transferred through the delta unit, the parallel-to-serial converter, and the serial-to-parallel converters.

[5] Since the state of this signal can be tested by the microprocessor, polling may also be implemented by disabling the corresponding interrupt.

The *configuration register* is used to configure some units of the SIMD module. There is no intrinsic difference between the functionality of this register and that of the instruction register, except that the fields of the latter are expected to change more often than those of the former.

The seven *address registers* are used to initialize the address counters of the XY, auxiliary Y, desired output, and weight memories. These counters can be set between any two instructions, and are incremented after each memory access. Except for the desired output memory, there are two counters associated with each memory. The "input" counter is used when a value is read from this memory or when it is read from a FIFO and simultaneously written into this memory (input phase). Similarly, the "output" counter is used when a value produced by the GENES IV array is written into this memory (output phase).

5.2.3 The performance monitor

Performance degradation in the MANTRA I machine can be of two types: static and dynamic (see section 7.1.1). The former type occurs when only a subset of the PEs is used during all or part of the computation. This is the case when the weight matrix (or sub-matrix) is smaller than the GENES IV array or the epoch length is smaller than the pipeline depth. On the other hand, dynamic performance degradation is due to stall cycles when instructions are lacking and the SIMD module is frozen.

Knowing the implementation of a model, the static performance can be calculated by a simple formula. Therefore, there is no need for any special hardware. This is not true for the non-deterministic dynamic performance, which can, for instance, be jeopardize by delays in the communication with the host computer. A dedicated unit, the *performance monitor*, has been implemented to measure the percentage of clock cycles that the SIMD module is active.

This unit is composed of two 20-bit counters, the *reference counter* incremented at each clock cycle and the *activity counter* incremented only on clock cycles when the SIMD module is busy. The monitor can be switched on and off. After $2^{20} - 1$ clock cycles, the value of the activity counter (in the range $[0,\ 2^{20} - 1]$) is latched in a register and an interrupt is sent to the microprocessor.

The current dynamic utilization rate can also be read on four LEDs. The first one is lighted over 25 %, the second over 50 %, the third over 75 %, and all four are lighted at 100 % utilization rate. These values are averaged over a period of 105 ms ($2^{20} - 1$ clock cycles).

5.3 Heterogeneous-node network

As described in section 5.2.1, a key feature of the TMS320C40 microprocessor is the availability of built-in communication channels. This makes it possible to connect the MANTRA I machine to a network of heterogeneous TMS320C40-based nodes, as shown in figure 5.5.

First, MANTRA I machines can be connected to a host computer. An SBus board containing a TMS320C40 has been developed for SUN 4 or SUN SPARCstation systems. It implements *direct virtual memory access (DVMA)*.

Another important element of this network, is a TMS320C40-based parallel computer. As mentioned in section 2.2, such a machine is very important for pre- or post-processing, because Amdahl's law limits the achievable speed-up, when a fraction of an application (in this case the non-neural computation) is not accelerated. This computer has been developed at the Microcomputing Laboratory [MAR93]. In this system, processors are interconnected by cables. Though

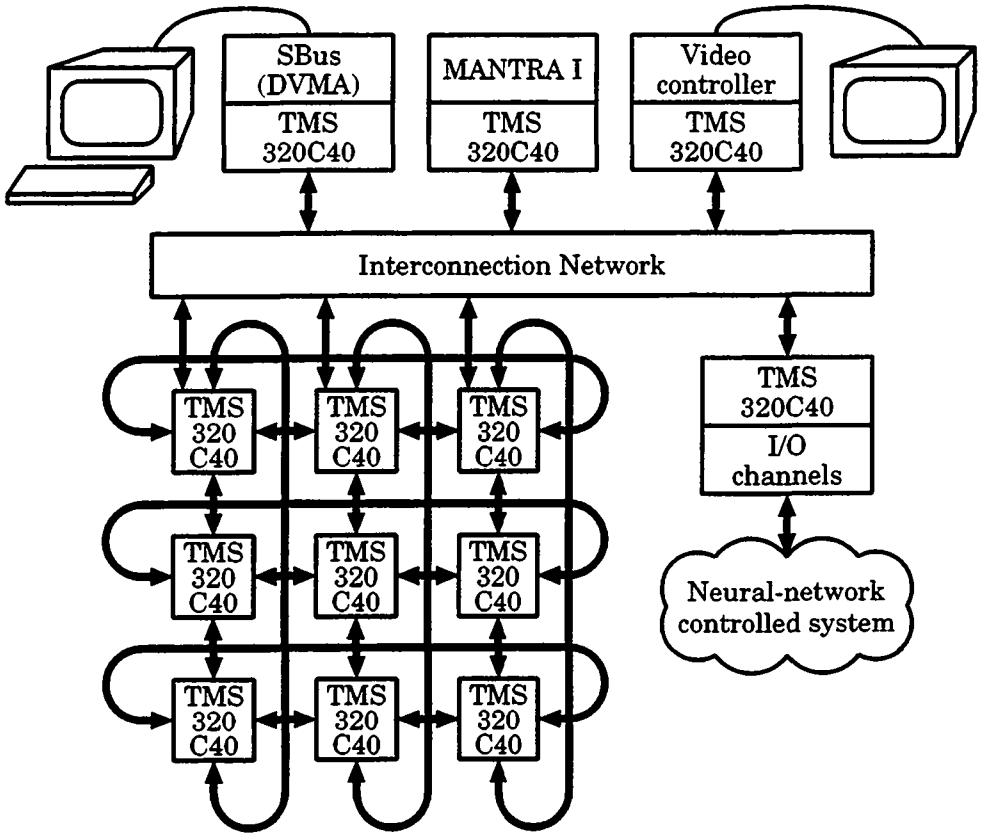


Figure 5.5: *TMS320C40-based heterogeneous-node network.*

figure 5.5 shows a torus, any topology requiring no more than six links per node can be hard-wired: rings (linear arrays), tori (meshes), tri-dimensional hyper-tori (k-ary cubes), hyper-cubes up to six dimensions, trees with up to five siblings, etc. If all six communication ports are used by the parallel computer (hyper-tori or hexa-dimensional hyper-cubes), the topology should be slightly altered to connect the computer to the heterogeneous-node network.

As mentioned in section 5.2.1, this computer's processor board has been designed as a versatile hardware platform for TMS320C40-based systems. For instance, the MANTRA I machine is built as such a system. In this way, any input/output peripheral (video-controller, etc.) can be easily interfaced to the network.

5.4 Hardware

A first prototype of the MANTRA I machine, shown in figure 5.6, has been built and tested. This configuration is based on a GENES IV array of 20×20 PEs. The system is composed of four different *printed circuit boards (PCBs)*, that is, a general-purpose processor board and three dedicated PCBs:

C40: TMS320C40 microprocessor board.
This PCB contains the control microprocessor (33 chips). It has been designed by C. Marguerat [MAR92] as a versatile platform for TMS320C40-based systems and parallel processors.

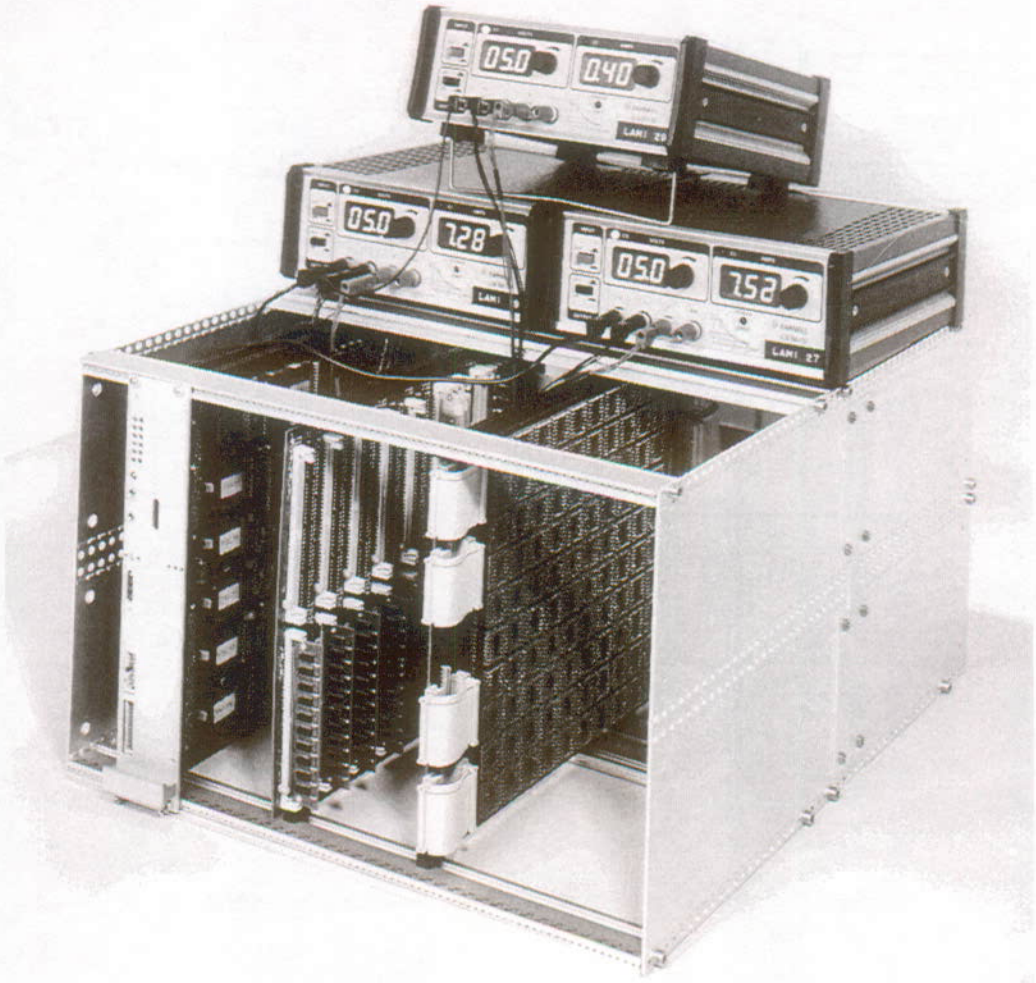


Figure 5.6: *The MANTRA I prototype machine (photograph by H. Viredaz-Bader).*

MANTRA1: MANTRA I control and storage board.
This PCB, shown in figure 5.7, implements the SIMD controller, the sigma unit, the function-of-Y unit, the data unit, the convergence unit, and the performance monitor (135 chips).

GENESIO: GENES IV Input/Output board.
This PCB, shown in figure 5.8, contains the delta unit, and the parallel-to-serial and serial-to-parallel converters (465 chips).

GA10x10: GENES IV Array 10×10 chip board.
This PCB, shown in figure 5.9, implements a matrix of $10 \times 10 = 100$ GENES IV chips. One such board is required for configurations of the machine up to 20×20 PEs. For larger configurations—up to 40×40 PEs—four such boards should be used.

Several electrical problems have been found on this first prototype, chiefly ground bounce and reflections [VIR93A]. Figure 5.7 clearly shows extra capacitors (directly soldered across the power pins of sensible chips) and coaxial cables (used to carry clock signals), which had to be added to overcome these problems. The clock frequencies of the control microprocessor and the SIMD module have been respectively decreased to 16 MHz and 8 MHz (instead of 20 MHz and 10 MHz).

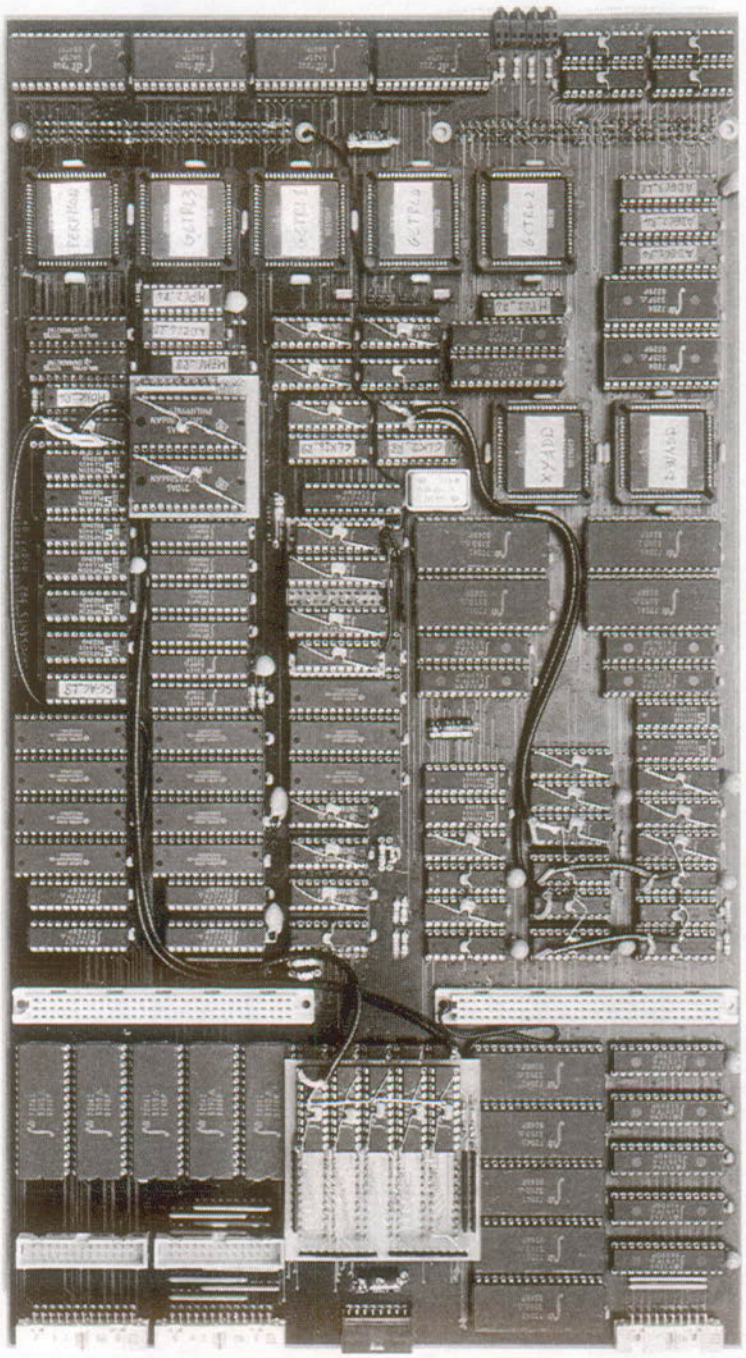


Figure 5.7: *The MANTRA I control and storage board (photograph by H. Viredaz-Bader).*

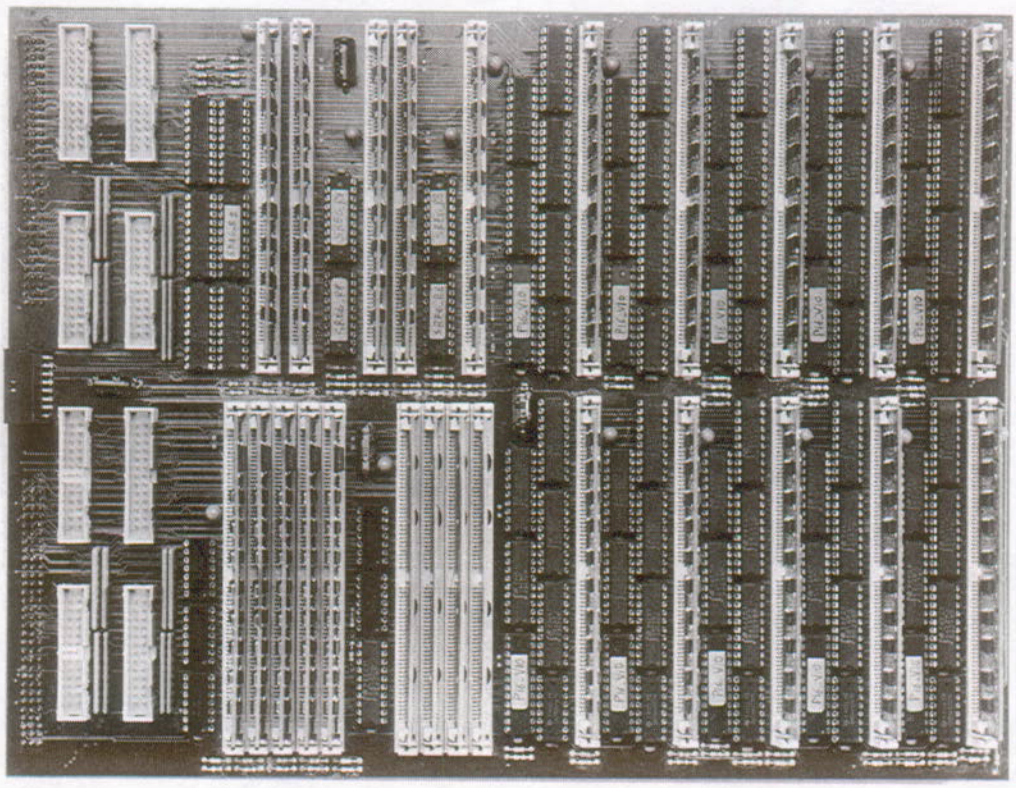


Figure 5.8: *The GENES IV input/output board (photograph by H. Viredaz-Bader).*

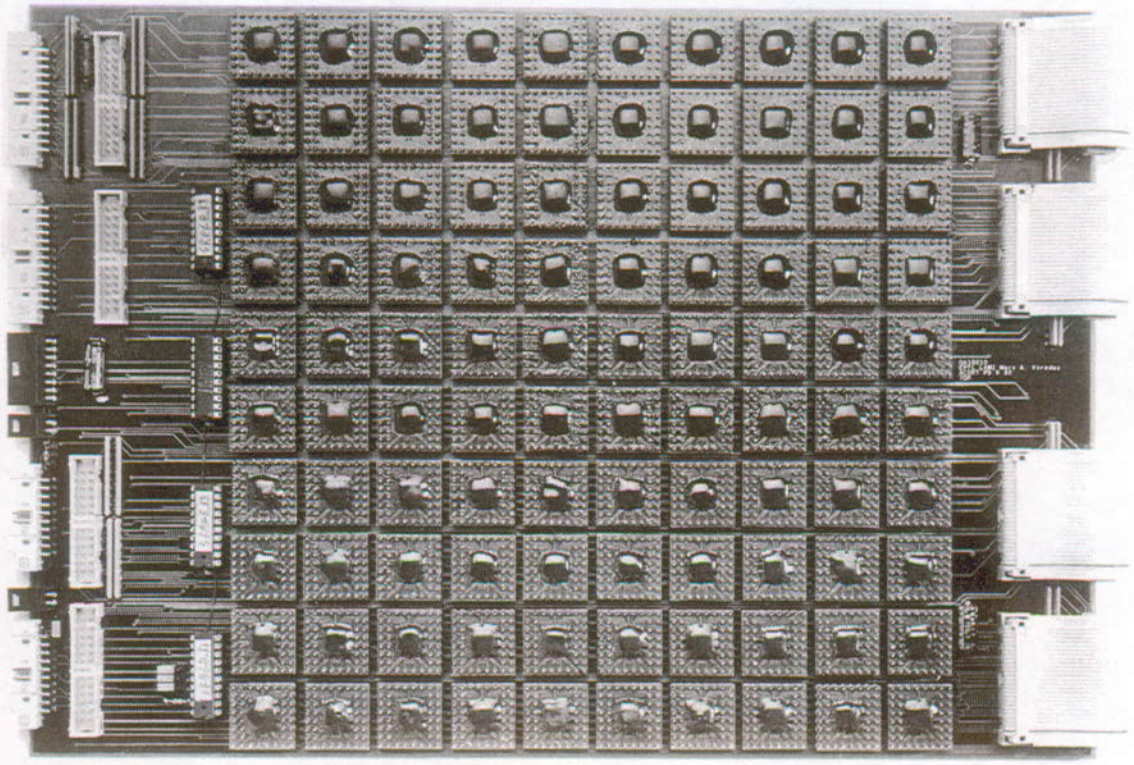


Figure 5.9: *The GENES IV array board (photograph by H. Viredaz-Bader).*

A second version of the control board (MANTRA1) has been designed. This PCB should avoid ground bounce and reflections thanks to the following techniques:

- Use of less sensitive integrated circuits (low-noise family, devices with multiple power pins, etc.).
- Use of *surface mounted devices (SMDs).*
- Clock and FIFO read/write signals routed on a dedicated layer and guarded by ground tracks (besides resistive terminations, already implemented on the first version).

This board is currently under test. This version is expected to run at the specified clock frequency. The GENES IV array will also be scaled up, by the addition of three GA10x10 boards.

References

[BLA90] T. Blank. *The MasPar MP-1 Architecture.* In *Proceedings of the IEEE COMPCON Spring.* IEEE Computer Society, February 1990.

[HIL85] W. D. Hillis. *The Connection Machine.* MIT Press, Cambridge, MA (USA), 1985.

[IEN92] P. Ienne and M. A. Viredaz. *The GENES Auxiliary Circuit for the Delta Rule.* Internal report no. R92.19I, LAMI, EPFL, Lausanne (CH), October 1992.

[IMS88] INMOS. *The Transputer Databook*, 1st edition, November 1988.

[IMS93] INMOS. *The T900 Transputer Hardware Reference Manual*, 1st edition, 1993.

[MAR92] C. Marguerat. *Développement avec le TMS320C40.* Internal report no. R92.44S, LAMI, EPFL, Lausanne (CH), October 1992.

[MAR93] C. Marguerat. *Artificial Neural Network Algorithms on a Parallel DSP System.* Internal report no. R93.23C, LAMI, EPFL, Lausanne (CH), 1993.

[NIC88] J.-D. Nicoud. *Video RAMs: Structure and Applications. IEEE Micro*, 8(1):8–27, February 1988.

[NIC90] J. R. Nickolls. *The Design of the MasPar MP-1: A Cost Effective Massively Parallel Computer.* In *Proceedings of the IEEE COMPCON Spring.* IEEE Computer Society, February 1990.

[TI91] Texas Instruments. *TMS320C4x User's Guide*, May 1991. Revision A.

[TM87] *Connection Machine® Model CM-2 Technical Summary.* Technical report HA87-4, Thinking Machines, April 1987.

[VIR92] M. A. Viredaz, C. Lehmann, F. Blayo, and P. Ienne. *MANTRA: A Multi-Model Neural-Network Computer.* In J. G. Delgado-Frias and W. R. Moore (eds.), *Proceedings of the 3rd International Workshop on VLSI for Neural Networks and Artificial Intelligence*, Oxford (GB), September 1992.

[VIR93A] M. A. Viredaz. *The MANTRA I Prototype Machine: Hardware Description.* MANTRA internal report no. 93/2, EPFL, Lausanne (CH), September 1993.

[VIR93B] M. A. Viredaz. *MANTRA I: An SIMD Processor Array for Neural Computation.* In P. P. Spies (ed.), *Euro-ARCH'93,* Munich, Informatik aktuell, pp. 99–110. Springer-Verlag, Berlin Heidelberg (D), October 1993.

[VIR93C] M. A. Viredaz and P. Ienne. *MANTRA I: A Systolic Neuro-Computer.* In *Proceedings of the International Joint Conference on Neural Networks,* vol. 3, pp. 3054–3057, Nagoya (J), October 1993. IEEE, INNS.

Customers should write checks, not software.
Earl E. Swartzlander, Jr., *Panel Session, ASAP'93*

6

Programming the MANTRA I Machine

Since the MANTRA I machine follows the VLIW philosophy, resource parallelism is managed by software. Therefore, the sustained performance of the machine depends heavily on the programmer's skills. However, the programming model of the machine is far too complex to be directly accessed by the average user. Classical tools to hide hardware details are compilers and libraries. Since writing a compiler for a parallel machine is a task beyond the capacity of a hardware laboratory, the library approach has been chosen.

The details of the programmer's model of the machine and a sample program can be found in an internal report [VIR93]. In this code, the instruction and data FIFOs composing the SIMD interface are directly accessed. This programming style may be considered as assembly, because MANTRA I instructions are directly coded in the source file. More precisely, the SIMD module is programmed in assembly language, while the control module is programmed in C. In a highly optimized version, assembly language could also be used for the control microprocessor.

To ease the task of writing library routines, a software scheduler has been written by P. Ienne [IEN94A, IEN94B]. This scheduler takes as input a serial description of an algorithm, consisting of basic matrix and vector operations. The parallelism is extracted by overlapping these operations as much as possible, while avoiding structural or data conflicts. This tool is currently used to port to the machine a power-system security application based on the Kohonen model [COR94].

This chapter discusses two issues related to the implementation of a library: the mapping of ANN models in section 6.1 and the integer data representation in section 6.2. These are algorithmic issues that are independent of the assembly or scheduler programming style.

6.1 Algorithm mapping

This section describes the mapping of the delta rule, back-propagation rule, and Kohonen model on the MANTRA I machine. Neither the Hopfield model nor the Kohonen model with recurrent network are discussed. The implementation of the former can easily be derived from the description

	Function-of-Y unit	Delta unit	GENES IV array	Sigma unit
A			$\vec{\mathbf{p}}^* := \mathbf{w}^* \cdot \vec{\mathbf{x}}^*$	$\vec{\mathbf{y}} := \sigma(\vec{\mathbf{p}}^*)$
B	$\vec{\mathbf{f}} := \alpha \cdot \sigma'(\sigma^{-1}(\vec{\mathbf{y}}))$	$\vec{\delta} := (\vec{\mathbf{d}} - \vec{\mathbf{y}}) \circ \vec{\mathbf{f}}$	$\mathbf{w}^* := \mathbf{w}^* + \vec{\delta} \cdot \vec{\mathbf{x}}^{*\mathrm{T}}$	

Algorithm 6.1: *The delta rule.*

of the GENES HN8 (see section 3.2), which is dedicated to this algorithm. The Kohonen model with recurrent network is close enough to the version with minimum, so that the mapping of the latter can be easily modified to yield that of the former.

6.1.1 The delta rule

The simplest ANN model that can be programmed on the MANTRA I machine is the delta rule, described in section 1.7. Although derived from different assumptions, the Perceptron rule (see section 1.6) updates the weights in a very similar way.[1] Its mapping on the hardware is identical to that of the delta rule, and the same routine can be used.

The systolic nature of the GENES IV array is well suited to the implementation of the epoch updating version (see section 1.7.1). Since the epoch length is given as a parameter, on-line or batch modes can also be used. A minimum epoch equal to the pipeline depth — that is, $2N + 3$ prototypes, where N is the number of PEs per array's edge — is required for an optimal utilization of the hardware. This restricts the use of on-line mode to small problems, for comparison purposes.

First, the weights are initialized to some predefined or random values. Each iteration consists of the evaluation of equation (1.7) for each prototype of the current epoch E, followed by the update of the weight matrix according to equation (1.32). In equation (1.7), thresholds are treated as weights. Since the machine can only handle this kind of networks and those without thresholds (i.e., described by equation (1.4) with $\vec{\theta}(t) = \vec{\mathbf{0}}$), the $*$-notation introduced in chapter 1 is used here as well.

In the simplest case, the weight matrix $\mathbf{w}^*$ is smaller or of the same size as the GENES IV array. Algorithm 6.1 shows how the different operations of an iteration (epoch) are mapped on the machine. The columns represent the pipeline formed by the four computing units of the SIMD module. The computation starts in the function-of-Y unit. The prototypes enter the delta unit a macro-cycle later, and the GENES IV array on the next macro-cycle. Finally, they are processed by the sigma unit $2N$ macro-cycles later. During phase A, the expressions given by algorithm 6.1 are iterated for each prototype $s \in E(t)$. There is an equal number of iterations during phase B.

To simplify the notations, the dependency on the prototype number s and update time t have been removed from the table, and ANN data are treated as Pascal variables.[2] Variable names are identical to the corresponding mathematical quantities, with the exception that, once a quantity has been multiplied by the learning coefficient α, it is not explicitly indicated. For example, the variable $\vec{\delta}$ is used to hold the quantity $\alpha(t) \cdot \vec{\delta}(t, s)$.

When the size e of an epoch is smaller than $2N + 3$, each phase A should be followed by $2N + 3 - e$ NOP instructions. This is not required for phase B.

[1] The Perceptron rule minimizes the error ε_i of a binary neuron, while the delta rule minimizes the quadratic error ξ_i of a neuron whose activation function is differentiable.

[2] This explains the use of the assignment symbol := in all expressions.

When a weight matrix $\mathbf{w}^*$ is larger than the array, it is processed using the virtual matrix technique. The matrix is divided into sub-matrices that fit the system, as shown by equation (3.22). Except for the last row and column, all sub-matrices have a size of $N \times N$. The implementation of an epoch is given by algorithm 6.2. During the multiplication of a sub-matrix row by the input vectors — that is, during phases $A_{[i,1]}$ to $A_{[i,r]}$ — data flow in a circular way through the horizontal path of the GENES IV array. Hence, the maximum and optimal epoch is $2N$ prototypes.[3] When the epoch length e is smaller, each phase $A_{[i,j]}$ should be followed by $2N - e$ additional NOP instructions. Again, this is not necessary for the phases $B_{[i,j]}$, because the function-of-Y and delta units perform r times the same computation during phases $B_{[i,1]}$ to $B_{[i,r]}$. If the epoch length is exactly $2N$, this redundant computation could be avoided by letting the update vector $\vec{\delta}_{[i]}$ flow in a circular way through the array. However, this modification does not speed up the algorithm and is omitted in practice.

In this implementation, weights are updated sub-matrix row by sub-matrix row. This is possible since all neurons are independent. Another possible implementation would be to compute all potentials first, and then to update all weights (i.e., having phases $A_{[1,1]} - A_{[1,r]}$ to $A_{[q,1]} - A_{[q,r]}$ followed by phases $B_{[1,1]} - B_{[1,r]}$ to $B_{[q,1]} - B_{[q,r]}$). This version requires more storage (in the auxiliary Y memory), but is closer to the implementation of the back-propagation rule (see section 6.1.2).

Another characteristic of algorithm 6.2 is that epochs may not be larger than $2N$ prototypes. Longer epochs require a modification of the implementation. Each block $A_{[i,1]} - A_{[i,r]}$ should be duplicated as many times as required, followed by the same number of duplications of the block $B_{[i,1]} - B_{[i,r]}$. An optimal utilization of the hardware is then achieved for epoch lengths that are multiples of $2N$. This is opposed to the case without virtual matrix, where the system is optimally used for any epoch larger than $2N + 3$ prototypes.

When a weight matrix or sub-matrix is smaller than the physical array, zeroes are injected in the unused rows and columns instead of the missing operands. As mentioned above, the Perceptron rule is implemented by the same routine. The function-of-Y unit should simply be initialized to compute $\vec{\mathbf{f}}_{[i]} := \alpha \cdot \vec{\mathbf{1}}$ during the phases $B_{[i,j]}$.

Momentum

To reduce the oscillations caused by the gradient descent, the *momentum* technique has been developed, which consist in adding to the weight-update matrix $\Delta\mathbf{w}^*$ a fraction of the previous one:

$$\Delta\mathbf{w}^*(t) = \gamma(t) \cdot \Delta\mathbf{w}^*(t-1) + \alpha(t) \cdot \sum_{s \in E(t)} \vec{\delta}(t,s) \cdot \vec{\mathbf{x}}^{*\mathrm{T}}(s) \tag{6.1}$$

This algorithm, as it stands, is not implementable by the MANTRA I machine. However, a satisfactory mapping on the machine can be found if the momentum term is approximated as follows:

$$\Delta\mathbf{w}^*(t) = \gamma(t) \cdot \alpha(t) \cdot \sum_{s \in E(t-1)} \vec{\delta}(t-1,s) \cdot \vec{\mathbf{x}}^{*\mathrm{T}}(s) + \alpha(t) \cdot \sum_{s \in E(t)} \vec{\delta}(t,s) \cdot \vec{\mathbf{x}}^{*\mathrm{T}}(s) \tag{6.2}$$

[3] Since this epoch size is slightly smaller than the pipeline depth $2N + 3$, a 100 % static utilization rate may not be achieved, because PEs are idle for a few cycles at the beginning and at the end of the block $A_{[i,1]} — A_{[i,r]}$. However, the resulting performance drop is usually negligible.

	Function-of-Y unit	Delta unit	GENES IV array	Sigma unit
$A_{[1,1]}$			$\vec{\mathbf{p}}^*_{[1]} := \mathbf{w}^*_{[1,1]} \cdot \vec{\mathbf{x}}^*_{[1]}$	
$A_{[1,2]}$			$\vec{\mathbf{p}}^*_{[1]} := \vec{\mathbf{p}}^*_{[1]} + \mathbf{w}^*_{[1,2]} \cdot \vec{\mathbf{x}}^*_{[2]}$	
⋮			⋮	
$A_{[1,r]}$			$\vec{\mathbf{p}}^*_{[1]} := \vec{\mathbf{p}}^*_{[1]} + \mathbf{w}^*_{[1,r]} \cdot \vec{\mathbf{x}}^*_{[r]}$	$\vec{\mathbf{y}}_{[1]} := \sigma(\vec{\mathbf{p}}^*_{[1]})$
$B_{[1,1]}$	$\vec{\mathbf{f}}_{[1]} := \alpha \cdot \sigma'(\sigma^{-1}(\vec{\mathbf{y}}_{[1]}))$	$\vec{\delta}_{[1]} := (\vec{\mathbf{d}}_{[1]} - \vec{\mathbf{y}}_{[1]}) \circ \vec{\mathbf{f}}_{[1]}$	$\mathbf{w}^*_{[1,1]} := \mathbf{w}^*_{[1,1]} + \vec{\delta}_{[1]} \cdot \vec{\mathbf{x}}^{*T}_{[1]}$	
⋮	⋮	⋮	⋮	
$B_{[1,r]}$	$\vec{\mathbf{f}}_{[1]} := \alpha \cdot \sigma'(\sigma^{-1}(\vec{\mathbf{y}}_{[1]}))$	$\vec{\delta}_{[1]} := (\vec{\mathbf{d}}_{[1]} - \vec{\mathbf{y}}_{[1]}) \circ \vec{\mathbf{f}}_{[1]}$	$\mathbf{w}^*_{[1,r]} := \mathbf{w}^*_{[1,r]} + \vec{\delta}_{[1]} \cdot \vec{\mathbf{x}}^{*T}_{[r]}$	
$A_{[q,1]}$			$\vec{\mathbf{p}}^*_{[q]} := \mathbf{w}^*_{[q,1]} \cdot \vec{\mathbf{x}}^*_{[1]}$	
$A_{[q,2]}$			$\vec{\mathbf{p}}^*_{[q]} := \vec{\mathbf{p}}^*_{[q]} + \mathbf{w}^*_{[q,2]} \cdot \vec{\mathbf{x}}^*_{[2]}$	
⋮			⋮	
$A_{[q,r]}$			$\vec{\mathbf{p}}^*_{[q]} := \vec{\mathbf{p}}^*_{[q]} + \mathbf{w}^*_{[q,r]} \cdot \vec{\mathbf{x}}^*_{[r]}$	$\vec{\mathbf{y}}_{[q]} := \sigma(\vec{\mathbf{p}}^*_{[q]})$
$B_{[q,1]}$	$\vec{\mathbf{f}}_{[q]} := \alpha \cdot \sigma'(\sigma^{-1}(\vec{\mathbf{y}}_{[q]}))$	$\vec{\delta}_{[q]} := (\vec{\mathbf{d}}_{[q]} - \vec{\mathbf{y}}_{[q]}) \circ \vec{\mathbf{f}}_{[q]}$	$\mathbf{w}^*_{[q,1]} := \mathbf{w}^*_{[q,1]} + \vec{\delta}_{[q]} \cdot \vec{\mathbf{x}}^{*T}_{[1]}$	
⋮	⋮	⋮	⋮	
$B_{[q,r]}$	$\vec{\mathbf{f}}_{[q]} := \alpha \cdot \sigma'(\sigma^{-1}(\vec{\mathbf{y}}_{[q]}))$	$\vec{\delta}_{[q]} := (\vec{\mathbf{d}}_{[q]} - \vec{\mathbf{y}}_{[q]}) \circ \vec{\mathbf{f}}_{[q]}$	$\mathbf{w}^*_{[q,r]} := \mathbf{w}^*_{[q,r]} + \vec{\delta}_{[q]} \cdot \vec{\mathbf{x}}^{*T}_{[r]}$	

Algorithm 6.2: *The delta rule using a virtual matrix (where* $q = \lceil \frac{m}{N} \rceil$ *and* $r = \lceil \frac{n}{N} \rceil$*).*

This is simply done by keeping the output vectors $\vec{\mathbf{y}}(t-1,s)$ of the previous epoch and by duplicating the phases B or $\mathrm{B}_{[i,j]}$ with a different function-of-Y table.

Equation (6.2) is a first order approximation. In principle, approximations of any order could be implemented (as long as there are enough function-of-Y tables). In practice, the first order approximation is already a rather inefficient process (in term of memory and computation time), and higher order approximations are prohibitively slow.

6.1.2 The back-propagation rule

The back-propagation rule (see section 1.8) is the most popular supervised learning rule for multi-layer feed-forward ANNs. As for the delta rule, only networks without thresholds—described by equation (1.18) with $\vec{\theta}^{[k]}(t) = \vec{\mathbf{0}}$—and with thresholds treated as weights—given by equation (1.21)—are considered. The latter type can be handled on the MANTRA I machine with a slight algorithmic modification. Appendix A.3 discusses how the basic rule should be rewritten. A possible implementation is to force the component $\delta^{\theta\,[k]}_{m_k+1}$ of the error signal vectors to zero. This can easily be achieved on the machine because, before injecting a vector (or a sub-vector) into the GENES IV array, its length should be stored in the input/output counter register (see section 5.2.2), the remaining components being set to zero. Therefore, it is sufficient to set the number of components to m_k instead of $m_k + 1$ when injecting the error vector $\vec{\delta}^{\theta\,[k]}$.

As in section 6.1.1, the $*$-notation is used. Since the first m_k elements of $\vec{\mathbf{y}}^{\theta\,[k]}$ and $\vec{\delta}^{\theta\,[k]}$ are respectively equal to those of $\vec{\mathbf{y}}^{[k]}$ and $\vec{\delta}^{[k]}$, the normal notation is used when only the first m_k components are computed or used as operands, and the $*$-notation when the last one is considered as well. A similar consideration applies to the top-left $m_k \times m_{k-1}$ sub-matrix of $\mathbf{w}^{\theta\,[k]}$.

The back-propagation rule with epoch updating is described by equations (1.36) to (1.38). A first possible mapping, without virtual matrices, is given by algorithm 6.3. The variable $\vec{\varepsilon}^{[k]}$ is used to represent the quantity $\alpha(t) \cdot \mathbf{w}^{[k+1]^{\mathrm{T}}}(t) \cdot \vec{\delta}^{[k+1]}(t,s)$ for all hidden layers. The maximum epoch size is $2N$. Longer epochs can be implemented using a technique similar to that described for the delta rule. The feed-forward phases $\mathrm{A}^{[1]}$ to $\mathrm{A}^{[L]}$ present no problem since the epoch length can be arbitrarily long. The blocks $\mathrm{B}^{[k]}$–$\mathrm{C}^{[k]}$ should then be duplicated as required. Here again, the hardware is optimally used for epochs with multiples of $2N$ prototypes. However, this mapping of the algorithm presents several drawbacks:

- The maximum epoch size is $2N$.
- When the epoch length e is smaller than $2N$, each phase $\mathrm{B}^{[k]}$ should be followed by $2N - e$ additional NOP instructions for all hidden layers (i.e., $k < L$).[4]
- During phase $\mathrm{B}^{[1]}$, the GENES IV array is idle for $2N$ macro-cycles.
- This scheme may not be extended to virtual matrices.

The common cause of these problems is that, on the hidden layers ($k < L$), it is not possible to repeat, during phase $\mathrm{C}^{[k]}$, the computation of the function-of-Y and delta units during phase $\mathrm{B}^{[k]}$. This is due to the architecture of the machine, where only two different values (i.e., $\mathsf{XY}_{15..0}$ and $\mathsf{AY}_{15..0}$) can be fed to the three busses $\mathsf{X}_{15..0}$, $\mathsf{Y1}_{15..0}$, and $\mathsf{Y2}_{15..0}$ (see figure 5.1).

[4] This is also true for all phases $\mathrm{A}^{[k]}$, but nothing can be done to avoid this inefficiency.

	Function-of-Y unit	Delta unit	GENES IV array	Sigma unit
$A^{[1]}$			$\vec{p}^{*[1]} := w^{*[1]} \cdot \vec{x}^{*}$	$\vec{y}^{*[1]} := \sigma(\vec{p}^{*[1]})$
$\vdots$			$\vdots$	$\vdots$
$A^{[k]}$			$\vec{p}^{*[k]} := w^{*[k]} \cdot \vec{y}^{*[k-1]}$	$\vec{y}^{*[k]} := \sigma(\vec{p}^{*[k]})$
$\vdots$			$\vdots$	$\vdots$
$A^{[L]}$			$\vec{p}^{*[L]} := w^{*[L]} \cdot \vec{y}^{*[L-1]}$	$\vec{y}^{[L]} := \sigma(\vec{p}^{*[L]})$
$B^{[L]}$	$\vec{f}^{[L]} := \alpha \cdot \sigma'(\sigma^{-1}(\vec{y}^{[L]}))$	$\vec{\delta}^{[L]} := (\vec{d} - \vec{y}^{[L]}) \circ \vec{f}^{[L]}$	$\vec{\varepsilon}^{[L-1]} := w^{[L]T} \cdot \vec{\delta}^{[L]}$	$\vec{\varepsilon}^{[L-1]} := \vec{\varepsilon}^{[L-1]}$
$C^{[L]}$	$\vec{f}^{[L]} := \alpha \cdot \sigma'(\sigma^{-1}(\vec{y}^{[L]}))$	$\vec{\delta}^{[L]} := (\vec{d} - \vec{y}^{[L]}) \circ \vec{f}^{[L]}$	$w^{*[L]} := w^{*[L]} + \vec{\delta}^{[L]} \cdot \vec{y}^{*[L-1]T}$	
$\vdots$				
$B^{[k]}$	$\vec{f}^{[k]} := -\sigma'(\sigma^{-1}(\vec{y}^{[k]}))$	$\vec{\delta}^{[k]} := (\vec{0} - \vec{\varepsilon}^{[k]}) \circ \vec{f}^{[k]}$	$\vec{\varepsilon}^{[k-1]} := w^{[k]T} \cdot \vec{\delta}^{[k]}$	$\vec{\varepsilon}^{[k-1]} := \vec{\varepsilon}^{[k-1]}$
$C^{[k]}$			$w^{*[k]} := w^{*[k]} + \vec{\delta}^{*[k]} \cdot \vec{y}^{*[k-1]T}$	
$\vdots$				
$B^{[1]}$	$\vec{f}^{[1]} := -\sigma'(\sigma^{-1}(\vec{y}^{[1]}))$	$\vec{\delta}^{[1]} := (\vec{0} - \vec{\varepsilon}^{[1]}) \circ \vec{f}^{[1]}$		
$C^{[1]}$			$w^{*[1]} := w^{*[1]} + \vec{\delta}^{*[1]} \cdot \vec{x}^{*T}$	

Algorithm 6.3: *The back-propagation rule (version I).*

A possible solution is to impose a value of zero on the $Y2_{15..0}$ bus (the design should be modified accordingly) and to inject the error vectors $\vec{\varepsilon}^{[k]}$ through the desired output unit. The error vector $\vec{\varepsilon}^{[k]}$ should be stored into the Y FIFO, during phase $B^{[k+1]}$, and retrieved from the desired output FIFO, during phase $B^{[k]}$, or from the desired output memory, during phase $C^{[k]}$, as shown in algorithm 6.4. The microprocessor is in charge of copying theses vectors from the Y FIFO to the desired output FIFO. The epoch size e can be arbitrarily chosen, but the optimal utilization rate is reached for any epoch with more than $2N+3$ prototypes. For smaller epochs, every phase $A^{[k]}$ should be padded with $2N+3-e$ NOP instructions. Moreover, if the double epoch size $2e$ is smaller than $2N+3$, each group $B^{[k]}-C^{[k]}$ should be padded with $2N+3-2e$ NOP instructions.

The learning coefficient α is treated differently than in algorithm 6.3, and the variable $\vec{\varepsilon}^{[k]}$ represents here the quantity $\mathbf{w}^{[k+1]^T}(t) \cdot \vec{\delta}^{[k+1]}(t,s)$. If required, the initialization of the function-of-Y unit may be modified to revert to the previous learning-coefficient processing scheme.

This algorithm can be extended to virtual matrices using a similar technique to that described for the delta rule (see appendix B.1). Similarly, the considerations on the momentum technique for the delta rule (see section 6.1.1) apply also to the back-propagation rule. However, the restricted number of function-of-Y tables would require a modification of the analysis presented in section 6.2.2.

6.1.3 The Kohonen model with minimum

The mapping of the Kohonen model is studied with its most popular version, that is, the algorithm with minimum and Euclidean distance (see section 1.10.1). The Kohonen learning operation of the GENES IV array (see section 4.1.1) can only be used for the semi epoch updating version, described by equations (1.45), (1.46), and (1.48).

The neighborhood relationship between neurons is described by the matrix λ. Hence, any map topology (n-dimensional meshes, bi-dimensional hexagonal arrays, etc.) is implementable.

As for previous ANN algorithms, the mapping is first studied for weight matrices fitting the array, before being extended to virtual matrices (see appendix B.2). Algorithm 6.5 shows the target implementation. The delta unit has been removed from the table, because it is computationally passive (i.e., the arithmetic unit is bypassed, as shown in section 5.1.1).

From phase A to phase B, data flow in a circular way through the GENES IV array. This implies that the maximum epoch size is $2N$, and that phase A (as well as phase B and C) should be followed by $2N-e$ additional NOP instructions if $e < 2N$. Since the squared Euclidean distances p_i are coded on $N_{PS} = 40$ bits (overflow bit included), the only way to find the smallest element with full precision is to keep the potential vector $\vec{\mathbf{p}}$ in the array. This is obviously not compatible with virtual matrices. However, since the goal of the minimum operation is to find the index of the winner, any loss of precision on the potentials p_i is tolerable as long as this does not affect the identification of the winner. Since a few false updates should not significantly hinder the convergence of the learning process, it has been decided to implement 16-bit paths and storage in the SIMD module. A carefully chosen function converting 40-bit potentials into 16-bit values, can significantly reduce the probability of choosing a wrong minimum.

However, a bug of the MANTRA I machine prevents the use of algorithm 6.5 if the number of neurons m is not strictly equal to the number of processors N per array's edge. To disable extra rows and columns during the search for the minimum and maximum elements (see section 4.1.1),

	Function-of-Y unit	Delta unit	GENES IV array	Sigma unit
$A^{[1]}$			$\vec{\mathbf{p}}^{*[1]} := \mathbf{w}^{*[1]} \cdot \vec{\mathbf{x}}^{*}$	$\vec{\mathbf{y}}^{*[1]} := \sigma\left(\vec{\mathbf{p}}^{*[1]}\right)$
$\vdots$			$\vdots$	$\vdots$
$A^{[k]}$			$\vec{\mathbf{p}}^{*[k]} := \mathbf{w}^{*[k]} \cdot \vec{\mathbf{y}}^{*[k-1]}$	$\vec{\mathbf{y}}^{*[k]} := \sigma\left(\vec{\mathbf{p}}^{*[k]}\right)$
$\vdots$			$\vdots$	$\vdots$
$A^{[L]}$			$\vec{\mathbf{p}}^{*[L]} := \mathbf{w}^{*[L]} \cdot \vec{\mathbf{y}}^{*[L-1]}$	$\vec{\mathbf{y}}^{[L]} := \sigma\left(\vec{\mathbf{p}}^{*[L]}\right)$
$B^{[L]}$	$\vec{\mathbf{f}}^{[L]} := \sigma'\left(\sigma^{-1}\left(\vec{\mathbf{y}}^{[L]}\right)\right)$	$\vec{\delta}^{[L]} := \left(\vec{\mathbf{d}} - \vec{\mathbf{y}}^{[L]}\right) \circ \vec{\mathbf{f}}^{[L]}$	$\vec{\varepsilon}^{[L-1]} := \mathbf{w}^{[L]\mathrm{T}} \cdot \vec{\delta}^{[L]}$	$\vec{\varepsilon}^{[L-1]} := \vec{\varepsilon}^{[L-1]}$
$C^{[L]}$	$\vec{\mathbf{f}}^{[L]} := \alpha \cdot \sigma'\left(\sigma^{-1}\left(\vec{\mathbf{y}}^{[L]}\right)\right)$	$\vec{\delta}^{[L]} := \left(\vec{\mathbf{d}} - \vec{\mathbf{y}}^{[L]}\right) \circ \vec{\mathbf{f}}^{[L]}$	$\mathbf{w}^{*[L]} := \mathbf{w}^{*[L]} + \vec{\delta}^{[L]} \cdot \vec{\mathbf{y}}^{*[L-1]\mathrm{T}}$	
$\vdots$				
$B^{[k]}$	$\vec{\mathbf{f}}^{[k]} := \sigma'\left(\sigma^{-1}\left(\vec{\mathbf{y}}^{[k]}\right)\right)$	$\vec{\delta}^{[k]} := \left(\vec{\varepsilon}^{[k]} - \vec{\mathbf{0}}\right) \circ \vec{\mathbf{f}}^{[k]}$	$\vec{\varepsilon}^{[k-1]} := \mathbf{w}^{[k]\mathrm{T}} \cdot \vec{\delta}^{[k]}$	$\vec{\varepsilon}^{[k-1]} := \vec{\varepsilon}^{[k-1]}$
$C^{[k]}$	$\vec{\mathbf{f}}^{[k]} := \alpha \cdot \sigma'\left(\sigma^{-1}\left(\vec{\mathbf{y}}^{[k]}\right)\right)$	$\vec{\delta}^{[k]} := \left(\vec{\varepsilon}^{[k]} - \vec{\mathbf{0}}\right) \circ \vec{\mathbf{f}}^{[k]}$	$\mathbf{w}^{*[k]} := \mathbf{w}^{*[k]} + \vec{\delta}^{*[k]} \cdot \vec{\mathbf{y}}^{*[k-1]\mathrm{T}}$	
$\vdots$				
$C^{[1]}$	$\vec{\mathbf{f}}^{[1]} := \alpha \cdot \sigma'\left(\sigma^{-1}\left(\vec{\mathbf{y}}^{[1]}\right)\right)$	$\vec{\delta}^{[1]} := \left(\vec{\varepsilon}^{[1]} - \vec{\mathbf{0}}\right) \circ \vec{\mathbf{f}}^{[1]}$	$\mathbf{w}^{*[1]} := \mathbf{w}^{*[1]} + \vec{\delta}^{*[1]} \cdot \vec{\mathbf{x}}^{*\mathrm{T}}$	

Algorithm 6.4: *The back-propagation rule (version II).*

	Function-of-Y unit	GENES IV array	Sigma unit
A		$p_i := \lvert\vec{\mathbf{x}} - \mathbf{w}_i^{\mathrm{T}}\rvert^2$	
B		$\mu_i := \begin{cases} p_i & \text{if } p_i = \min(\vec{\mathbf{p}}) \\ N_{\max} & \text{otherwise} \end{cases}$	$\upsilon_i := \begin{cases} 1 & \text{if } \mu_i \neq N_{\max} \\ 0 & \text{if } \mu_i = N_{\max} \end{cases}$
C		$\vec{\mathbf{v}} := \alpha \cdot \boldsymbol{\lambda} \cdot \vec{\upsilon}$	$\vec{\mathbf{v}} := \vec{\mathbf{v}}$
D	$\vec{\mathbf{v}} := \vec{\mathbf{v}}$	$\mathbf{w}_i := \mathbf{w}_i + \nu_i \cdot (\vec{\mathbf{x}}^{\mathrm{T}} - \mathbf{w}_i)$	

Algorithm 6.5: *The Kohonen model with minimum and Euclidean distance (version I).*

	Function-of-Y unit	GENES IV array	Sigma unit
A		$p_i := \lvert\vec{\mathbf{x}} - \mathbf{w}_i^{\mathrm{T}}\rvert^2$	$\vec{\mathbf{p}} := \vec{\mathbf{p}}$
B	$p_i := N_{\max} - p_i$	$\mu_i := \begin{cases} p_i & \text{if } p_i = \max(\vec{\mathbf{p}}) \\ N_{\min} & \text{otherwise} \end{cases}$	$\upsilon_i := \begin{cases} 1 & \text{if } \mu_i \neq N_{\min} \\ 0 & \text{if } \mu_i = N_{\min} \end{cases}$
C		$\vec{\mathbf{v}} := \alpha \cdot \boldsymbol{\lambda} \cdot \vec{\upsilon}$	$\vec{\mathbf{v}} := \vec{\mathbf{v}}$
D	$\vec{\mathbf{v}} := \vec{\mathbf{v}}$	$\mathbf{w}_i := \mathbf{w}_i + \nu_i \cdot (\vec{\mathbf{x}}^{\mathrm{T}} - \mathbf{w}_i)$	

Algorithm 6.6: *The Kohonen model with minimum and Euclidean distance (version II).*

the largest representable value $N_{\max}$, respectively the smallest one $N_{\min}$, should be injected in these rows and columns. However, the machine injects zeroes instead, which means that, when $m < N$, the minimum operation is only correct for negative numbers and the maximum operation for positive ones. This problem could easily be corrected in a future version of the control board (see section 5.4). Since the squared Euclidean distances p_i are always positive, a work-around solution can be found. The potentials p_i should be subtracted from the largest number $N_{\max}$, and the maximum element should be searched. This results in algorithm 6.6. This implementation prevents the use of the full precision (40 bits). However, this should not be an important drawback, since full precision can anyway not be used with virtual matrices (see appendix B.2).

Further modifications are required to account for the integer data representation. For instance, during phase C the GENES IV array should compute $\vec{\mathbf{v}} := -\frac{1}{2} \cdot \alpha \cdot \boldsymbol{\lambda} \cdot \vec{\upsilon}$, the vector $\vec{\mathbf{v}}$ being then multiplied by -2 in the delta unit during phase D. Since this level of details is irrelevant to the present discussion, it does not appear in algorithms 6.5, 6.6, and B.2.

6.2 Integer data representation

In the MANTRA I machine, numbers are represented as two's complement integers. Since the majority of real-life applications use floating-point numbers, data should be scaled and converted to integers. This section discusses the constraints imposed by the hardware on this scaling. To distinguish the integer representations on the machine from the real values, the former quantities are represented with a superscript M.

For the sake of clarity, the dependency of data on the prototype number s and update time t has been suppressed in this section, unless relevant to the formula.

6.2.1 The delta rule

Three scaling factors a_w, a_x, and a_y are introduced to represent adequately the weights, inputs, and outputs used by the delta rule (see section 1.7):

$$\mathbf{w}^{\mathrm{M}} = a_w \cdot \mathbf{w} \quad \text{where } a_w \in \mathbb{R}_+^* \tag{6.3}$$

$$\vec{\mathbf{x}}^{\mathrm{M}} = a_x \cdot \vec{\mathbf{x}} \quad \text{where } a_x \in \mathbb{R}_+^* \tag{6.4}$$

$$\vec{\mathbf{y}}^{\mathrm{M}} = a_y \cdot \vec{\mathbf{y}} \quad \text{where } a_y \in \mathbb{R}_+^* \tag{6.5}$$

Since the delta rule is a supervised learning algorithm, a desired output vector is associated to each prototype. It is obviously scaled identically to the output vector:

$$\vec{\mathbf{d}}^{\mathrm{M}} = a_y \cdot \vec{\mathbf{d}} \tag{6.6}$$

The first task is to determine how to initialize the sigma unit. For this purpose, the scaling of the potentials should be derived:

$$\vec{\mathbf{p}}^{\mathrm{M}} = \mathbf{w}^{\mathrm{M}} \cdot \vec{\mathbf{x}}^{\mathrm{M}} = a_x \cdot a_w \cdot \mathbf{w} \cdot \vec{\mathbf{x}} = a_x \cdot a_w \cdot \vec{\mathbf{p}} \tag{6.7}$$

Equation (6.8) shows how the output vector $\vec{\mathbf{y}}^{\mathrm{M}}$ is computed on the machine, while equation (6.9) describes how this vector is related to the corresponding real value:

$$\vec{\mathbf{y}}^{\mathrm{M}} = \sigma^{\mathrm{M}}\left(\vec{\mathbf{p}}^{\mathrm{M}}\right) \tag{6.8}$$

$$\vec{\mathbf{y}}^{\mathrm{M}} = a_y \cdot \vec{\mathbf{y}} = a_y \cdot \sigma\left(\vec{\mathbf{p}}\right) = a_y \cdot \sigma\left(\frac{\vec{\mathbf{p}}^{\mathrm{M}}}{a_x \cdot a_w}\right) \tag{6.9}$$

The function σ^{M} to be implemented by the sigma unit can be derived by unifying equations (6.8) and (6.9):

$$\sigma^{\mathrm{M}}(v) = a_y \cdot \sigma\left(\frac{v}{a_x \cdot a_w}\right) \tag{6.10}$$

The same task should also be carried out for the function-of-Y unit. Again, equation (6.11) shows how the weight matrix is updated on the machine, and equation (6.12) how it is related to the problem's weight matrix:

$$\Delta\mathbf{w}^{\mathrm{M}} = \frac{\left(\left(\vec{\mathbf{d}}^{\mathrm{M}} - \vec{\mathbf{y}}^{\mathrm{M}}\right) \circ f^{\mathrm{M}}\left(\vec{\mathbf{y}}^{\mathrm{M}}\right)\right) \cdot \vec{\mathbf{x}}^{\mathrm{M}\mathrm{T}}}{2^{N_{\mathrm{W}}^{\mathrm{frc}}}} \tag{6.11}$$

$$\begin{aligned}
\Delta\mathbf{w}^{\mathrm{M}} &= a_w \cdot \Delta\mathbf{w} = a_w \cdot \alpha \cdot \left(\left(\vec{\mathbf{d}} - \vec{\mathbf{y}}\right) \circ \sigma'\left(\vec{\mathbf{p}}\right)\right) \cdot \vec{\mathbf{x}}^{\mathrm{T}} \\
&= a_w \cdot \alpha \cdot \left(\left(\frac{\vec{\mathbf{d}}^{\mathrm{M}}}{a_y} - \frac{\vec{\mathbf{y}}^{\mathrm{M}}}{a_y}\right) \circ \sigma'\left(\sigma^{-1}\left(\frac{\vec{\mathbf{y}}^{\mathrm{M}}}{a_y}\right)\right)\right) \cdot \left(\frac{\vec{\mathbf{x}}^{\mathrm{M}}}{a_x}\right)^{\mathrm{T}} \\
&= \frac{a_w}{a_x \cdot a_y} \cdot \alpha \cdot \left(\left(\vec{\mathbf{d}}^{\mathrm{M}} - \vec{\mathbf{y}}^{\mathrm{M}}\right) \circ \sigma'\left(\sigma^{-1}\left(\frac{\vec{\mathbf{y}}^{\mathrm{M}}}{a_y}\right)\right)\right) \cdot \vec{\mathbf{x}}^{\mathrm{M}\mathrm{T}}
\end{aligned} \tag{6.12}$$

where the symbol $\circ$ represents the *Hadamard product* or term-to-term matrix multiplication, and $N_{\mathrm{W}}^{\mathrm{frc}} = N_{\mathrm{W}}^{\mathrm{lrn}} - N_{\mathrm{W}}^{\mathrm{eval}} = 16$ is the number of bits of the fractional part of the weight registers. The

function f^{M} to be loaded in the function-of-Y table can be derived from equations (6.11) and (6.12):

$$f^{\mathrm{M}}(v) = \frac{a_w}{a_x \cdot a_y} \cdot 2^{N_{\mathrm{W}}^{\mathrm{frc}}} \cdot \alpha \cdot \sigma'\left(\sigma^{-1}\left(\frac{v}{a_y}\right)\right) \tag{6.13}$$

To summarize these results, the three coefficients a_x, a_y, and a_w may be considered as degrees of freedom in the problem of scaling the data of a given application to implement it on the MANTRA I machine. Once these factors have been chosen, equations (6.10) and (6.13) describe how to configure the sigma and function-of-Y units of the machine. However, the scaling factors may not be freely chosen, since the finite precision of the hardware imposes some constraints.

The first two conditions are that the inputs and desired outputs must fit on $N_{\mathrm{X}} = 16$ bits and $N_{\mathrm{DES}} = 16$ bits respectively:

$$a_x \leq \frac{2^{N_{\mathrm{X}}-1} - 1}{x_{\max}} \tag{6.14}$$

$$a_y \leq \frac{2^{N_{\mathrm{DES}}-1} - 1}{d_{\max}} \tag{6.15}$$

where $x_{\max}$ and $d_{\max}$ are defined as the largest absolute values of the corresponding quantities:

$$x_{\max} = \max_{s,j}\left(|x_j(s)|\right) \quad \begin{array}{l}\text{for } s = 1,2,\ldots,S \\ \text{for } j = 1,2,\ldots,n\end{array} \tag{6.16}$$

$$d_{\max} = \max_{s,i}\left(|d_i(s)|\right) \quad \begin{array}{l}\text{for } s = 1,2,\ldots,S \\ \text{for } i = 1,2,\ldots,m\end{array} \tag{6.17}$$

A minor optimization is to determine whether $x_{\max}$ comes from a positive or negative value and, in the latter case, to use the largest magnitude of negative numbers $2^{N_{\mathrm{X}}-1}$ instead of the positive one $2^{N_{\mathrm{X}}-1} - 1$. In this document, this type of optimization is systematically disregarded. However, sometimes the largest positive value is used and sometimes the largest negative one, in order to derive always the most conservative formulae.

Constraining the weight scaling factor a_w is less easy, because there is no *a priori* knowledge of the weight space, and only indirect conditions can be found. There is no guarantee that the weights will not overflow. Only an *a posteriori* analysis of the overflow bits can yield this information.

The next constraint is that the outputs of the neurons must fit on $N_{\mathrm{Y}} = 16$ bits, or more exactly, that the output of the activation function σ^{M} must have this precision. On the machine, the magnitude of the potentials can be bounded by:

$$|p_i(t,s)| \leq p_{\max}^{\mathrm{M}} = \min\left(n \cdot 2^{N_{\mathrm{W}}^{\mathrm{eval}}+N_{\mathrm{D}}^{\mathrm{eval}}-2}, 2^{N_{\mathrm{PS}}^{\mathrm{eval}}-1}\right) \quad \begin{array}{l}\text{for } s = 1,2,\ldots,S \\ \text{for } i = 1,2,\ldots,m\end{array} \tag{6.18}$$

where $N_{\mathrm{W}}^{\mathrm{eval}} = 16$ bits, $N_{\mathrm{D}}^{\mathrm{eval}} = 16$ bits, and $N_{\mathrm{PS}}^{\mathrm{eval}} = 39$ bits. Therefore, the largest possible output of the activation function is:

$$\begin{aligned}\sigma_{\max}^{\mathrm{M}} &= \max_{v\in[-p_{\max}^{\mathrm{M}},\, p_{\max}^{\mathrm{M}}]}\left(\left|\sigma^{\mathrm{M}}(v)\right|\right) \\ &= \max_{v\in[-p_{\max}^{\mathrm{M}},\, p_{\max}^{\mathrm{M}}]}\left(\left|a_y \cdot \sigma\left(\frac{v}{a_x \cdot a_w}\right)\right|\right) \\ &= a_y \cdot \max_{v\in[-\frac{p_{\max}^{\mathrm{M}}}{a_x\cdot a_w},\, \frac{p_{\max}^{\mathrm{M}}}{a_x\cdot a_w}]}\left(|\sigma(v)|\right)\end{aligned} \tag{6.19}$$

Since the product $a_x \cdot a_w$ can hardly be removed from the max function, an upper bound for this expression is $\sigma^{\mathrm{M}}_{\max} \leq a_y \cdot \sigma_{\max}$, where:

$$\sigma_{\max} = \max_{v \in [-\infty, +\infty]} (|\sigma(v)|) \tag{6.20}$$

Since the activation function σ is usually bounded by a pair of left and right horizontal asymptotes, this limit exists in most practical cases. Should this function be unbounded, the user must truncate it outside the "useful range." The next condition — that is, that the output of the activation function must fit on $N_{\mathrm{Y}} = 16$ bits — can be written as:

$$a_y \leq \frac{2^{N_{\mathrm{Y}}-1} - 1}{\sigma_{\max}} \tag{6.21}$$

A similar condition can be drawn for the function-of-Y unit. The following expression can be derived from equations (6.12) and (6.13):

$$f^{\mathrm{M}}(\vec{\mathrm{y}}^{\mathrm{M}}) = \frac{a_w}{a_x \cdot a_y} \cdot 2^{N^{\mathrm{frc}}_{\mathrm{W}}} \cdot \alpha \cdot \sigma'(\vec{\mathrm{p}}) \tag{6.22}$$

The output of this unit is bounded by:

$$\begin{aligned} f^{\mathrm{M}}_{\max} &= \frac{a_w}{a_x \cdot a_y} \cdot 2^{N^{\mathrm{frc}}_{\mathrm{W}}} \cdot \alpha_{\max} \cdot \max_{v \in [-\frac{p^{\mathrm{M}}_{\max}}{a_x \cdot a_w}, \frac{p^{\mathrm{M}}_{\max}}{a_x \cdot a_w}]} (|\sigma'(v)|) \\ &\leq \frac{a_w}{a_x \cdot a_y} \cdot 2^{N^{\mathrm{frc}}_{\mathrm{W}}} \cdot \alpha_{\max} \cdot \sigma'_{\max} \end{aligned} \tag{6.23}$$

where $\sigma'_{\max}$ is the largest value of the activation function's derivative:

$$\sigma'_{\max} = \max_{v \in [-\infty, +\infty]} (|\sigma'(v)|) \tag{6.24}$$

Since the output of the function-of-Y table must fit on $N_{\mathrm{FY}} = 16$ bits, this condition becomes:

$$\frac{a_w}{a_x \cdot a_y} \leq \frac{2^{N_{\mathrm{FY}}-1} - 1}{2^{N^{\mathrm{frc}}_{\mathrm{W}}} \cdot \alpha_{\max} \cdot \sigma'_{\max}} \tag{6.25}$$

The relations (6.14), (6.15), (6.21), and (6.25) are strong constraints, because if they would be violated, some data (i.e., the inputs, the desired outputs, the activation function, or its derivative) could not be loaded on the machine. Condition (6.21) is more restrictive than condition (6.15) when $d_{\max} \leq \sigma_{\max}$. Since this condition is true in any mathematically sound problem (i.e., the output range of the activation function should contain all desired outputs), the relation (6.15) can be ignored.

The next two conditions prevent some prototypes from systematically overflowing the weights. These are somewhat weaker constraints, because they do not prevent the algorithm from being executed when they are violated. It is possible that an application converges even if they are only satisfied after a given number of iterations $t_{\text{V-lim}}$. This can be expressed by choosing a value $\alpha_{\text{V-lim}}$ of the learning coefficient, so that $\alpha(t) \leq \alpha_{\text{V-lim}}$ for $t \geq t_{\text{V-lim}}$. Since the learning coefficient is usually a decreasing function, this can be reduced to $\alpha_{\text{V-lim}} = \alpha(t_{\text{V-lim}})$. The convergence analysis under this hypothesis is beyond the scope of this thesis. Hence, the conservative approach is to take $\alpha_{\text{V-lim}} = \alpha_{\max}$.

Data conversion	$\mathbf{w}^{\mathrm{M}} = a_w \cdot \mathbf{w}$ $\vec{\mathbf{x}}^{\mathrm{M}} = a_x \cdot \vec{\mathbf{x}}$ $\vec{\mathbf{y}}^{\mathrm{M}} = a_y \cdot \vec{\mathbf{y}}$ $\vec{\mathbf{d}}^{\mathrm{M}} = a_y \cdot \vec{\mathbf{d}}$
Look-up tables	$\sigma^{\mathrm{M}}(v) = a_y \cdot \sigma\left(\frac{v}{a_x \cdot a_w}\right)$ $f^{\mathrm{M}}(v) = \frac{a_w}{a_x \cdot a_y} \cdot 2^{N_{\mathrm{W}}^{\mathrm{frc}}} \cdot \alpha \cdot \sigma'\left(\sigma^{-1}\left(\frac{v}{a_y}\right)\right)$
Constraints on the scaling factors	$a_x \leq \frac{2^{N_{\mathrm{X}}-1}-1}{x_{\max}}$ $a_y \leq \frac{2^{N_{\mathrm{Y}}-1}-1}{\sigma_{\max}}$ $\frac{a_w}{a_x \cdot a_y} \leq \frac{2^{N_{\mathrm{FY}}-1}-1}{2^{N_{\mathrm{W}}^{\mathrm{frc}}} \cdot \alpha_{\max} \cdot \sigma'_{\max}}$ $\frac{a_w}{a_x} \leq \frac{2^{N_{\mathrm{mul}}-1}-1}{2^{N_{\mathrm{W}}^{\mathrm{frc}}} \cdot \alpha_{\text{V-lim}} \cdot (d_{\max}+\sigma_{\max}) \cdot \sigma'_{\max}}$

Table 6.1: *Integer implementation of the delta rule on the MANTRA I machine.*

tions (6.14) and (6.21) give a maximum for a_x and a_y respectively. This delimits a rectangular area on the plane (a_x, a_y). Relation (6.25) corresponds to a hyperbolic paraboloid (a "horse saddle" function) whose center point is at the origin, and relation (6.27) to a plane containing the a_y axis. The former surface is the lowest of the two for:

$$a_y \leq \frac{\left(2^{N_{\mathrm{mul}}-1}-1\right) \cdot \alpha_{\max}}{\left(2^{N_{\mathrm{FY}}-1}-1\right) \cdot \alpha_{\text{V-lim}} \cdot (d_{\max}+\sigma_{\max})} \tag{6.31}$$

For the sake of clarity, figure 6.1 has been plotted for a ratio $\frac{\alpha_{\max}}{\alpha_{\text{V-lim}}}$ equal to 10000. In most real problems, this ratio and the relative size of the hyperbolic paraboloid are expected to be much smaller. The paraboloid becomes predominant when:

$$\frac{\alpha_{\max}}{\alpha_{\text{V-lim}}} \geq \frac{\left(2^{N_{\mathrm{Y}}-1}-1\right) \cdot \left(2^{N_{\mathrm{FY}}-1}-1\right) \cdot (d_{\max}+\sigma_{\max})}{\left(2^{N_{\mathrm{mul}}-1}-1\right) \cdot \sigma_{\max}} \tag{6.32}$$

Table 6.1 summarizes how the basic algorithms 6.1 and 6.2, implementing the delta rule, should be modified to account for the integer nature of the MANTRA I machine. The first part of this table defines the scaling factors for the different data used by this ANN model. The second part shows how the look-up tables of the sigma and function-of-Y unit should be initialized. Finally, the third part gives the minimum set of strong and weak constraints on the scaling factors.

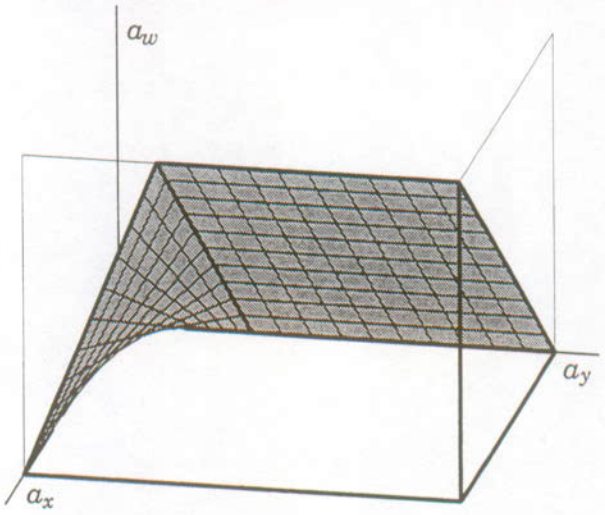


Figure 6.1: *Constraints on the scaling factors for the delta rule.*

The first weak condition is that the output of the delta unit must fit on the GENES IV multiplier's size $N_{\text{mul}} = 17$ bits. This quantity can be derived from equation (6.22):

$$\left(\vec{\mathbf{d}}^{\mathrm{M}} - \vec{\mathbf{y}}^{\mathrm{M}}\right) \circ f^{\mathrm{M}}\left(\vec{\mathbf{y}}^{\mathrm{M}}\right) \;=\; \frac{a_w}{a_x} \cdot 2^{N_{\mathrm{W}}^{\mathrm{frc}}} \cdot \alpha \cdot \left(\vec{\mathbf{d}} - \vec{\mathbf{y}}\right) \circ \sigma'\left(\vec{\mathbf{p}}\right) \tag{6.26}$$

and the condition be written as:

$$\frac{a_w}{a_x} \;\leq\; \frac{2^{N_{\text{mul}}-1} - 1}{2^{N_{\mathrm{W}}^{\mathrm{frc}}} \cdot \alpha_{\text{V-lim}} \cdot \left(d_{\max} + \sigma_{\max}\right) \cdot \sigma'_{\max}} \tag{6.27}$$

The second weak condition is that the weight-update amount—that is, the result of the multiplication in the Hebbian learning operation—must fit on $N_{\mathrm{W}}^{\mathrm{lrn}} = 32$ bits. This amount can be derived from equation (6.26):

$$\left(\left(\vec{\mathbf{d}}^{\mathrm{M}} - \vec{\mathbf{y}}^{\mathrm{M}}\right) \circ f^{\mathrm{M}}\left(\vec{\mathbf{y}}^{\mathrm{M}}\right)\right) \cdot \vec{\mathbf{x}}^{\mathrm{M}^{\mathrm{T}}} \;=\; a_w \cdot 2^{N_{\mathrm{W}}^{\mathrm{frc}}} \cdot \alpha \cdot \left(\left(\vec{\mathbf{d}} - \vec{\mathbf{y}}\right) \circ \sigma'\left(\vec{\mathbf{p}}\right)\right) \cdot \vec{\mathbf{x}}^{\mathrm{T}} \tag{6.28}$$

Hence, the second weak condition can be written as:

$$a_w \;\leq\; \frac{2^{N_{\mathrm{W}}^{\mathrm{lrn}}-1} - 1}{2^{N_{\mathrm{W}}^{\mathrm{frc}}} \cdot \alpha_{\text{V-lim}} \cdot x_{\max} \cdot \left(d_{\max} + \sigma_{\max}\right) \cdot \sigma'_{\max}} \tag{6.29}$$

This constraint is less restrictive than relation (6.27) over the allowed range for a_x. This can easily be seen, since condition (6.29) is predominant for:

$$a_x \;\geq\; \frac{2^{N_{\mathrm{W}}^{\mathrm{lrn}}-1} - 1}{\left(2^{N_{\text{mul}}-1} - 1\right) \cdot x_{\max}} \;\approx\; \frac{32768.5}{x_{\max}} \tag{6.30}$$

Since this relation is incompatible with constraint (6.14): $a_x \leq \frac{32767}{x_{\max}}$, condition (6.29) can be ignored.

Figure 6.1 displays the constraints on the scaling factors in the space (a_x, a_y, a_w). Condi-

Heuristic choice of the scaling factors

The choice of an optimal point in the region shown in figure 6.1 is usually problem dependent, and the user may be required to perform some experimental tuning. Alternatively, it is also possible to define a heuristic. For instance, this routine would perform a few learning iterations with given values for the scaling factors, analyze the weights and the outputs, and finally define a new set of scaling factors according to some predefined rules. This process can be iteratively repeated until some criteria are met.

The most obvious rules are to reduce the weight scaling factor a_w if the learning is hindered by a large number of weight saturations, and to reduce both the input and weight scaling factors a_w and a_x if the convergence is impeded by a frequent saturation on the potential. There are other relations that are less easy to find. For instance, reducing the input or output scaling factors a_x and a_y increases the resolution of the function-of-Y table. Therefore, decreasing one of these factors may help convergence, because the activation function's derivative is better represented.

The start-point of such a heuristic is another degree of freedom. A possible choice is the point defined by the maxima of the three factors. This optimizes the data representation of the weights, inputs, and outputs. However, it can be shown that, in this case, the effective size of the function-of-Y table is one bit (i.e., every entry is either zero or one). Experimental results, shown in chapter 7, exhibit an optimal behavior when five to six bits are sacrificed from the representation of inputs and outputs, while a weight scaling factor of 80 % to 90 % of its maximal value has been found optimal. Therefore, an efficient heuristic would not start with the largest possible values of the scaling factors, but rather with some predefined fractions of the maxima. It should be stressed that these results do not imply an over-design of the hardware, since they are strictly algorithm dependent. For instance, the full input range can be used for the Kohonen model.

6.2.2 The back-propagation rule

Since the back-propagation rule (see section 1.8) is an extension of the delta rule to multi-layer feed-forward networks, most of the results derived in section 6.2.1 have similar counterparts here. Since the network is multi-layer, a different scaling factor is defined for each layer. The present analysis shows that these factors can be reduced back to three degrees of freedom. Equation (6.4) is still valid, but equations (6.3) and (6.5) become:

$$\mathbf{w}^{\mathrm{M}[k]} = a_w^{[k]} \cdot \mathbf{w}^{[k]} \quad \text{where } a_w^{[k]} \in \mathbb{R}_+^* \quad \text{for } k = 1, 2, \ldots, L \tag{6.33}$$

$$\vec{\mathbf{y}}^{\mathrm{M}[k]} = a_y^{[k]} \cdot \vec{\mathbf{y}}^{[k]} \quad \text{where } a_y^{[k]} \in \mathbb{R}_+^* \quad \text{for } k = 1, 2, \ldots, L \tag{6.34}$$

As shown in algorithm 6.4, the sigma unit is used in phases $\mathrm{A}^{[k]}$ and $\mathrm{B}^{[k]}$, and both look-up tables must be initialized. A similar calculation as for equation (6.10) gives the function to be implemented during phases $\mathrm{A}^{[k]}$:

$$\sigma_{\mathrm{A}}^{\mathrm{M}}(v) = a_y^{[k]} \cdot \sigma\left(\frac{v}{a_y^{[k-1]} \cdot a_w^{[k]}}\right) \tag{6.35}$$

with $a_y^{[0]} = a_x$. Since the same table should be used for all layers, the following constraints on the

scaling factors can be drawn:

$$a_w^{[1]} = \frac{a_y}{a_x} \cdot a_w \tag{6.36}$$

$$a_w^{[k]} = a_w \qquad \text{for } k = 2,\ldots,L \tag{6.37}$$

$$a_y^{[k]} = a_y \qquad \text{for } k = 1,2,\ldots,L \tag{6.38}$$

Since the output vector $\vec{\mathbf{y}}^{[k]}$ of a layer is the input of the next one, at first sight taking $a_y = a_x$ (and thus $a_w^{[1]} = a_w$) may seem to be a good idea. However, since the input vector $\vec{\mathbf{x}}$ is usually a physical quantity different from the output vector $\vec{\mathbf{y}}^{[L]}$, this would often lead to a poor utilization of the available data dynamic.

During phases $\mathrm{B}^{[k]}$ the sigma unit is used to output the error vector $\vec{\varepsilon}^{[k-1]}$ (see algorithm 6.4). Although this corresponds conceptually to an identity function, the unit converts a 40-bit value into a 16-bit one and hence performs data re-scaling:

$$\sigma_{\mathrm{B}}^{\mathrm{M}}(v) = \left\lfloor \frac{v}{\Gamma} \right\rfloor \tag{6.39}$$

The constant Γ may be freely chosen and represent a fourth degree of freedom. Although it can be any number, due to the architecture of the sigma unit (see section 5.1.2), full-precision functions can be computed only for $\Gamma = 2^c$ with $c = 0,1,\ldots,7$ (implemented by the fine-grain table) or $c = 16,\ldots,23$ (implemented by the coarse-grain table). However, the latter series leads to constants usually far too large.

The same task should now be carried out for the function-of-Y unit. Algorithm 6.4 shows that two different functions should be implemented for the phases $\mathrm{B}^{[k]}$ and $\mathrm{C}^{[k]}$. Moreover, since the scaling factors are different for the first layer than for the others, a third function should be used for phase $\mathrm{C}^{[1]}$. Since only four tables are available, the same tables should be used for all phases $\mathrm{B}^{[k]}$ and for the phases $\mathrm{C}^{[L]}$ to $\mathrm{C}^{[2]}$. Since there is no *a priori* knowledge on the scaling of the error vectors $\vec{\varepsilon}^{[k]}$, new scaling factors are defined:

$$\varepsilon^{\mathrm{M}[k]} = a_\varepsilon^{[k]} \cdot \varepsilon^{[k]} \qquad \text{where } a_\varepsilon^{[k]} \in \mathbb{R}_+^* \qquad \text{for } k = 1,2,\ldots,L \tag{6.40}$$

Equation (6.41) shows how the error vector $\vec{\varepsilon}^{[L-1]}$ is computed during phase $\mathrm{B}^{[L]}$ (the floor function, which is not relevant to this analysis, has been suppressed), and equation (6.42) how this quantity is related to the problem's values:

$$\vec{\varepsilon}^{\mathrm{M}[L-1]} = \frac{\mathbf{w}^{\mathrm{M}[L]^{\mathrm{T}}} \cdot \left(\left(\vec{\mathbf{d}}^{\mathrm{M}} - \vec{\mathbf{y}}^{\mathrm{M}[L]} \right) \circ f_{\mathrm{B}}^{\mathrm{M}} \left(\vec{\mathbf{y}}^{\mathrm{M}[L]} \right) \right)}{\Gamma} \tag{6.41}$$

$$\begin{aligned} \vec{\varepsilon}^{\mathrm{M}[L-1]} &= a_\varepsilon^{[L-1]} \cdot \vec{\varepsilon}^{[L-1]} = a_\varepsilon^{[L-1]} \cdot \mathbf{w}^{[L]^{\mathrm{T}}} \cdot \left(\left(\vec{\mathbf{d}} - \vec{\mathbf{y}}^{[L]} \right) \circ \sigma' \left(\vec{\mathbf{p}}^{[L]} \right) \right) \\ &= \frac{a_\varepsilon^{[L-1]}}{a_y \cdot a_w} \cdot \mathbf{w}^{\mathrm{M}[L]^{\mathrm{T}}} \cdot \left(\left(\vec{\mathbf{d}}^{\mathrm{M}} - \vec{\mathbf{y}}^{\mathrm{M}[L]} \right) \circ \sigma' \left(\sigma^{-1} \left(\frac{\vec{\mathbf{y}}^{\mathrm{M}[L]}}{a_y} \right) \right) \right) \end{aligned} \tag{6.42}$$

By unifying these equations, the function to be computed by the function-of-Y unit, during phase $\mathrm{B}^{[L]}$, can be found:

$$f_{\mathrm{B}}^{\mathrm{M}}(v) = \frac{a_\varepsilon^{[L-1]}}{a_y \cdot a_w} \cdot \Gamma \cdot \sigma' \left(\sigma^{-1} \left(\frac{v}{a_y} \right) \right) \tag{6.43}$$

This analysis can be repeated for the phases $\mathrm{B}^{[L-1]}$ to $\mathrm{B}^{[2]}$:

$$\vec{\varepsilon}^{\,\mathrm{M}[k-1]} = \frac{\mathbf{w}^{\mathrm{M}[k]\mathrm{T}} \cdot \left(\vec{\varepsilon}^{\,\mathrm{M}[k]} \circ f_{\mathrm{B}}^{\mathrm{M}}\left(\vec{\mathbf{y}}^{\mathrm{M}[k]}\right)\right)}{\Gamma} \qquad \text{for } k = 2,\ldots,L-1 \quad (6.44)$$

$$\begin{aligned}\vec{\varepsilon}^{\,\mathrm{M}[k-1]} &= a_{\varepsilon}^{[k-1]} \cdot \vec{\varepsilon}^{\,[k-1]} = a_{\varepsilon}^{[k-1]} \cdot \mathbf{w}^{[k]\mathrm{T}} \cdot \left(\vec{\varepsilon}^{\,[k]} \circ \sigma'\left(\vec{\mathbf{p}}^{\,[k]}\right)\right) \\ &= \frac{a_{\varepsilon}^{[k-1]}}{a_w \cdot a_{\varepsilon}^{[k]}} \cdot \mathbf{w}^{\mathrm{M}[k]\mathrm{T}} \cdot \left(\vec{\varepsilon}^{\,\mathrm{M}[k]} \circ \sigma'\left(\sigma^{-1}\left(\frac{\vec{\mathbf{y}}^{\mathrm{M}[k]}}{a_y}\right)\right)\right) \qquad \text{for } k = 2,\ldots,L-1 \quad (6.45)\end{aligned}$$

yielding:

$$f_{\mathrm{B}}^{\mathrm{M}}(v) = \frac{a_{\varepsilon}^{[k-1]}}{a_w \cdot a_{\varepsilon}^{[k]}} \cdot \Gamma \cdot \sigma'\left(\sigma^{-1}\left(\frac{v}{a_y}\right)\right) \qquad \text{for } k = 2,\ldots,L-1 \quad (6.46)$$

The functions implemented by the function-of-Y unit during phase $\mathrm{C}^{[L]}$ and phases $\mathrm{C}^{[L-1]}$ to $\mathrm{C}^{[2]}$ can be derived in a similar way as equation (6.13):

$$f_{\mathrm{C}^{[L..2]}}^{\mathrm{M}}(v) = \frac{a_w}{a_y^2} \cdot 2^{N_{\mathrm{W}}^{\mathrm{frc}}} \cdot \alpha \cdot \sigma'\left(\sigma^{-1}\left(\frac{v}{a_y}\right)\right) \quad (6.47)$$

$$f_{\mathrm{C}^{[L..2]}}^{\mathrm{M}}(v) = \frac{a_w}{a_y \cdot a_{\varepsilon}^{[k]}} \cdot 2^{N_{\mathrm{W}}^{\mathrm{frc}}} \cdot \alpha \cdot \sigma'\left(\sigma^{-1}\left(\frac{v}{a_y}\right)\right) \qquad \text{for } k = 2,\ldots,L-1 \quad (6.48)$$

Since the same function $f_{\mathrm{C}^{[L..2]}}^{\mathrm{M}}$ should be used for all layers 2 to L, it can be derived from equations (6.47) and (6.48) that $a_{\varepsilon}^{[k]} = a_y$ for $k = 2,\ldots,L-1$. Using a similar consideration for the function $f_{\mathrm{B}}^{\mathrm{M}}$, this equality can be extended to $k = 1$. When $L > 3$, equation (6.46) is enough since it implies that the ratio $a_{\varepsilon}^{[k-1]} \,/\, a_{\varepsilon}^{[k]}$ is constant for $k = 2,\ldots,L-1$. When $L = 3$, the combination of equations (6.43) and (6.46) gives the desired result. It should be noticed that, for $L = 2$, each phase uses a different look-up table. Therefore, there is no constraint on the unique $a_{\varepsilon}^{[1]} = a_{\varepsilon}$, which is another degree of freedom. Since 3-layer networks are the most common topologies, and since this section aims at providing general results, 2-layer networks are not treated separately. The results derived here are still applicable to ANNs with two layers, but may lead to a non-optimal use of the available dynamic. The user interested in such an optimal implementation is invited to modify this analysis to take the scaling factor a_{ε} into account.

Finally a formula similar to equation (6.48), combined with equation (6.36), gives the function to be implemented by this unit during phase $\mathrm{C}^{[1]}$:

$$f_{\mathrm{C}^{[1]}}^{\mathrm{M}}(v) = \frac{a_w}{a_x^2} \cdot 2^{N_{\mathrm{W}}^{\mathrm{frc}}} \cdot \alpha \cdot \sigma'\left(\sigma^{-1}\left(\frac{v}{a_y}\right)\right) \quad (6.49)$$

Now that the initialization of the look-up tables has been defined, the constraints on the three degrees of freedom a_x, a_y, and a_w should be derived. Following the same approach as in section 6.2.1 strong and weak constraints are distinguished. The strong constraints (6.14), (6.15), and (6.21) hold, with the relation (6.21) being still more restrictive than condition (6.15).

The next conditions force the output of the three function-of-Y tables $f_{\mathrm{B}}^{\mathrm{M}}$, $f_{\mathrm{C}^{[1]}}^{\mathrm{M}}$, and $f_{\mathrm{C}^{[L..2]}}^{\mathrm{M}}$ to fit on $N_{\mathrm{FY}} = 16$ bits. They are derived in a similar way as condition (6.25):

$$a_w \geq \frac{\Gamma \cdot \sigma'_{\max}}{2^{N_{\mathrm{FY}}-1} - 1} \quad (6.50)$$

$$\frac{a_w}{a_x^2} \leq \frac{2^{N_{\text{FY}}-1}-1}{2^{N_{\text{W}}^{\text{frc}}} \cdot \alpha_{\max} \cdot \sigma'_{\max}} \tag{6.51}$$

$$\frac{a_w}{a_y^2} \leq \frac{2^{N_{\text{FY}}-1}-1}{2^{N_{\text{W}}^{\text{frc}}} \cdot \alpha_{\max} \cdot \sigma'_{\max}} \tag{6.52}$$

Some conditions should now be found to ensure that the output of the delta unit fits on the multiplier's size $N_{\text{mul}} = 17$ bits. This requires to find a maximum magnitude for the elements of the error vectors $\vec{\varepsilon}^{[k]}$. On the last layer, this vector is equal to $\vec{\varepsilon}^{[L]} = \vec{\mathbf{d}} - \vec{\mathbf{y}}^{[L]}$ and its elements are bounded in magnitude by $d_{\max} + \sigma_{\max}$. This expression is in turn bounded by $2\,\sigma_{\max}$, which may be considered as the largest possible error. By analogy, the same maximum error can be taken for internal layers. This requires to saturate the sigma table $\sigma_{\text{B}}^{\text{M}}$ at $\pm 2 \cdot a_y \cdot \sigma_{\max}$.

The following three constraints are similar to condition (6.27) and derived for the three different function-of-Y tables. While in the last two conditions (corresponding to phase $\text{C}^{[1]}$ and to the phases $\text{C}^{[L]}$ to $\text{C}^{[2]}$) this data is used for a weight update, in the first condition (phases $\text{B}^{[L]}$ to $\text{B}^{[2]}$) it is used for a matrix-vector multiplication in transpose mode. Since, for this operation, the vector's components are not checked to be 17-bit values, condition (6.53) is a strong constraint, while conditions (6.54) and (6.55) are weak:

$$\frac{a_w}{a_y} \geq \frac{2 \cdot \Gamma \cdot \sigma_{\max} \cdot \sigma'_{\max}}{2^{N_{\text{mul}}-1}-1} \tag{6.53}$$

$$\frac{a_y \cdot a_w}{a_x^2} \leq \frac{2^{N_{\text{mul}}-1}-1}{2^{N_{\text{W}}^{\text{frc}}+1} \cdot \alpha_{\text{V-lim}} \cdot \sigma_{\max} \cdot \sigma'_{\max}} \tag{6.54}$$

$$\frac{a_w}{a_y} \leq \frac{2^{N_{\text{mul}}-1}-1}{2^{N_{\text{W}}^{\text{frc}}+1} \cdot \alpha_{\text{V-lim}} \cdot \sigma_{\max} \cdot \sigma'_{\max}} \tag{6.55}$$

Finally, the last two constraints prevent the weight-update amounts to be outside the weight range. They are derived similarly to condition (6.29):

$$\frac{a_y \cdot a_w}{a_x} \leq \frac{2^{N_{\text{W}}^{\text{lm}}-1}-1}{2^{N_{\text{W}}^{\text{frc}}+1} \cdot \alpha_{\text{V-lim}} \cdot x_{\max} \cdot \sigma_{\max} \cdot \sigma'_{\max}} \tag{6.56}$$

$$a_w \leq \frac{2^{N_{\text{W}}^{\text{lm}}-1}-1}{2^{N_{\text{W}}^{\text{frc}}+1} \cdot \alpha_{\text{V-lim}} \cdot \sigma_{\max}^2 \cdot \sigma'_{\max}} \tag{6.57}$$

These two relations are predominant over conditions (6.54) and (6.55), respectively, when:

$$a_x \geq \frac{2^{N_{\text{W}}^{\text{lm}}-1}-1}{\left(2^{N_{\text{mul}}-1}-1\right) \cdot x_{\max}} \approx \frac{32768.5}{x_{\max}} \tag{6.58}$$

$$a_y \geq \frac{2^{N_{\text{W}}^{\text{lm}}-1}-1}{\left(2^{N_{\text{mul}}-1}-1\right) \cdot \sigma_{\max}} \approx \frac{32768.5}{\sigma_{\max}} \tag{6.59}$$

Since these constraints are incompatible with condition (6.14): $a_x \leq \frac{32767}{x_{\max}}$ and condition (6.21): $a_y \leq \frac{32767}{\sigma_{\max}}$, relations (6.56) and (6.57) can be ignored.

Figure 6.2(a) shows the constraints on the scaling factors in the space (a_x, a_y, a_w). The four relations (6.51), (6.52), (6.54), and (6.55) define an upper-bound surface for a_w. The first two

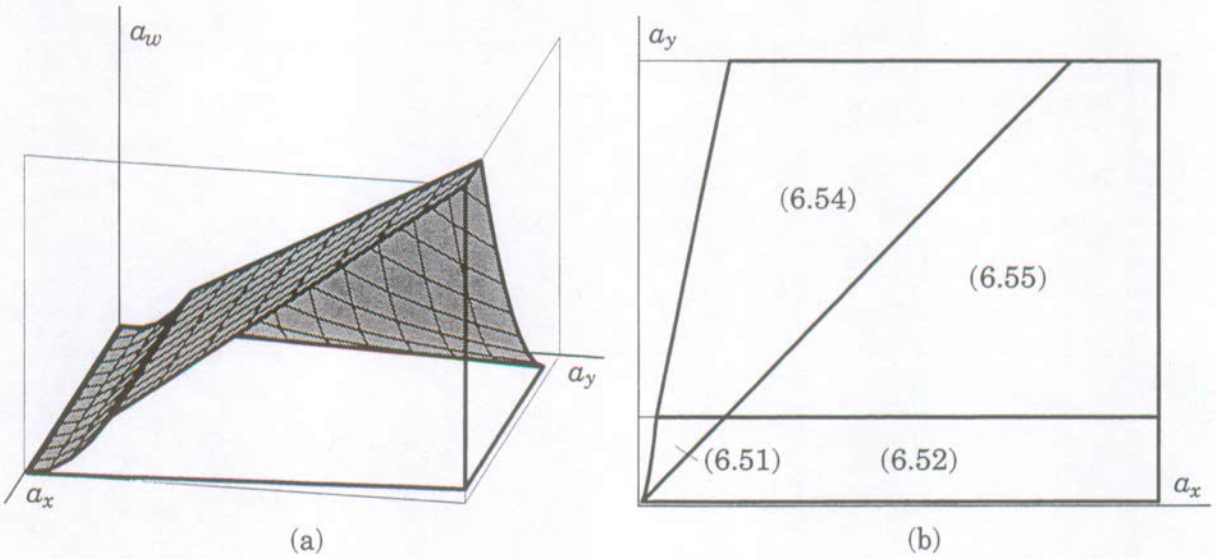


Figure 6.2: *Constraints on the scaling factors for the back-propagation rule.* (a) *Allowed volume in the space* (a_x, a_y, a_w). (b) *Predominance areas of the upper-bound equations in the plane* (a_x, a_y).

surfaces so defined are translational paraboloids,[5] while the last one is a plane. Condition (6.51) is predominant over condition (6.54) and condition (6.52) over condition (6.55) when:

$$a_y < \frac{\left(2^{N_{\text{mul}}-1} - 1\right) \cdot \alpha_{\max}}{2 \cdot \left(2^{N_{\text{FY}}-1} - 1\right) \cdot \alpha_{\text{V-lim}} \cdot \sigma_{\max}} \tag{6.60}$$

Similarly, conditions (6.51) and (6.54) are more restrictive than conditions (6.52) and (6.55) respectively, when $a_x < a_y$ (in the positive quadrant). Figure 6.2(b) shows the respective predominance areas of these constraints in the plane (a_x, a_y).

Conditions (6.50) and (6.53) give a lower bound on a_w, the former relation being more restrictive than the latter for:

$$a_y < \frac{2^{N_{\text{mul}}-1} - 1}{2 \cdot \left(2^{N_{\text{FY}}-1} - 1\right) \cdot \sigma_{\max}} \tag{6.61}$$

Since this quantity is approximately 0.003 % of the maximum value of a_y given by relation (6.21), condition (6.50) can be ignored in most practical cases. This condition does not appear in figure 6.2(a), since the lower bound it imposes is larger than the upper bound given by constraints (6.51) and (6.52) on all its predominance surface. However, this is not always the case, and condition (6.50) may not be removed from the set of constraints, although often useless.

For the sake of visibility, figure 6.2(a) has been plotted with a ratio $\frac{\alpha_{\max}}{\alpha_{\text{V-lim}}} \approx 6550$. In most application, this expression is expected to be much smaller, and the boundary given by relation (6.60) much closer to the a_x axis. Similarly, the constant Γ has been taken equal to 2^{15}, to emphasize the slope of the lower-bound plane given by condition (6.53), which would be much flatter in most applications. These results are summarized in table 6.2.

[5] A translational paraboloid is a surface generated by the translation of a parabola along the normal direction to its plane. It is a degenerated paraboloid.

Data conversion	$\mathbf{w}^{\mathrm{M}[1]} = \frac{a_y}{a_x} \cdot a_w \cdot \mathbf{w}^{[1]}$ $\mathbf{w}^{\mathrm{M}[k]} = a_w \cdot \mathbf{w}^{[k]}$ for $k = 2, \ldots, L$ $\vec{\mathbf{x}}^{\mathrm{M}} = a_x \cdot \vec{\mathbf{x}}$ $\vec{\mathbf{y}}^{\mathrm{M}[k]} = a_y \cdot \vec{\mathbf{y}}^{[k]}$ for $k = 1, 2, \ldots, L$ $\vec{\mathbf{d}}^{\mathrm{M}} = a_y \cdot \vec{\mathbf{d}}$
Look-up tables	$\sigma_{\mathrm{A}}^{\mathrm{M}}(v) = a_y \cdot \sigma\left(\frac{v}{a_y \cdot a_w}\right)$ $\sigma_{\mathrm{B}}^{\mathrm{M}}(v) = \min\left(\max\left(\left\lfloor \frac{v}{\Gamma} \right\rfloor, -2 \cdot a_y \cdot \sigma_{\max}\right), +2 \cdot a_y \cdot \sigma_{\max}\right)$ $f_{\mathrm{B}}^{\mathrm{M}}(v) = \frac{\Gamma}{a_w} \cdot \sigma'\left(\sigma^{-1}\left(\frac{v}{a_y}\right)\right)$ $f_{\mathrm{C}[1]}^{\mathrm{M}}(v) = \frac{a_w}{a_x^2} \cdot 2^{N_{\mathrm{W}}^{\mathrm{frc}}} \cdot \alpha \cdot \sigma'\left(\sigma^{-1}\left(\frac{v}{a_y}\right)\right)$ $f_{\mathrm{C}[L..2]}^{\mathrm{M}}(v) = \frac{a_w}{a_y^2} \cdot 2^{N_{\mathrm{W}}^{\mathrm{frc}}} \cdot \alpha \cdot \sigma'\left(\sigma^{-1}\left(\frac{v}{a_y}\right)\right)$
Constraints on the scaling factors	$a_x \leq \frac{2^{N_{\mathrm{X}}-1} - 1}{x_{\max}}$ $a_y \leq \frac{2^{N_{\mathrm{Y}}-1} - 1}{\sigma_{\max}}$ $a_w \geq \frac{\Gamma \cdot \sigma'_{\max}}{2^{N_{\mathrm{FY}}-1} - 1}$ $\frac{a_w}{a_y} \geq \frac{2 \cdot \Gamma \cdot \sigma_{\max} \cdot \sigma'_{\max}}{2^{N_{\mathrm{mul}}-1} - 1}$ $\frac{a_w}{a_x^2} \leq \frac{2^{N_{\mathrm{FY}}-1} - 1}{2^{N_{\mathrm{W}}^{\mathrm{frc}}} \cdot \alpha_{\max} \cdot \sigma'_{\max}}$ $\frac{a_w}{a_y^2} \leq \frac{2^{N_{\mathrm{FY}}-1} - 1}{2^{N_{\mathrm{W}}^{\mathrm{frc}}} \cdot \alpha_{\max} \cdot \sigma'_{\max}}$ $\frac{a_y \cdot a_w}{a_x^2} \leq \frac{2^{N_{\mathrm{mul}}-1} - 1}{2^{N_{\mathrm{W}}^{\mathrm{frc}}+1} \cdot \alpha_{\text{V-lim}} \cdot \sigma_{\max} \cdot \sigma'_{\max}}$ $\frac{a_w}{a_y} \leq \frac{2^{N_{\mathrm{mul}}-1} - 1}{2^{N_{\mathrm{W}}^{\mathrm{frc}}+1} \cdot \alpha_{\text{V-lim}} \cdot \sigma_{\max} \cdot \sigma'_{\max}}$

Table 6.2: *Integer implementation of the back-propagation rule on the MANTRA I machine.*

6.2.3 The Kohonen model with minimum

The scaling problem is much simpler for the Kohonen model with minimum (see section 1.10.1) than for the delta and back-propagation rules. This is because there is only one degree of freedom instead of three, and because there is no activation function.

Equations (6.3) to (6.5) are still valid. However, since the transposed rows $\mathbf{w}_i^T$ of the weight matrix belong to the same space as the input vectors $\vec{\mathbf{x}}$, their scaling factors are equal: $a_w = a_x$. Moreover, since each output y_i is the distance between the corresponding weight vector and the input vector, the scaling factor a_y is also equal to a_x. In the MANTRA I machine, since the computed potential p_i is the squared Euclidean distance, its scaling factor is a_x^2. Although this data is re-scaled when output from the GENES IV array through the sigma unit, this operation has no influence on the algorithm as long as the smallest distance remains so. Therefore, the only remaining constraint is that the inputs should fit on $N_X = 16$ bits, given by condition (6.14).

References

[COR94] T. Cornu, P. Ienne, D. Niebur, and M. A. Viredaz. *A Systolic Accelerator for Power System Security Assessment.* In *Proceedings of the International Conference on Intelligent System Application to Power Systems*, Montpellier (F), September 1994. To appear.

[IEN94A] P. Ienne and M. A. Viredaz. *Implementation of Kohonen's Self-Organizing Maps on MANTRA I.* In *Proceedings of the 4th International Conference on Microelectronics for Neural Networks and Fuzzy Systems*, Turin (I), September 1994. To appear.

[IEN94B] P. Ienne. *Microcode Scheduling for Systolic Arrays.* Internal report, LAMI, EPFL, Lausanne (CH), 1994. To appear.

[VIR93] M. A. Viredaz. *The MANTRA I Prototype Machine: Hardware Description.* MANTRA internal report no. 93/2, EPFL, Lausanne (CH), September 1993.

There is no strong performance without a little fanaticism in the performer.
Ralph W. Emerson, *Journals*, 1959

7

Performance Analysis

Computer performance evaluation is at the same time one of the most important and one of the most delicate field of computer science. If the quantities to measure or the conditions under which they are measured are not carefully chosen, the analysis may lead (sometimes purposely) to a totally erroneous conclusion. In his book [JAI91], R. Jain states:

> *Unfortunately, the types of applications of computers are so numerous, that it is not possible to have a standard measure of performance, a standard measurement environment (application), or a standard technique for all cases. The first step in performance evaluation is to select the right measures of performance, the right measurement environments, and the right techniques.*

Due to the algorithmic restrictions imposed by the MANTRA I machine (integer computation, epoch updating, etc.), the raw performance—that is, the number of operations per time unit—should not be used alone to characterize the machine. It would not be fair to compare systems on the basis of their raw performance, if the restrictions, imposed by an accelerator like the MANTRA I machine, slow down the convergence speed. So, after a discussion of the relevant metrics and of peak performance, the machine is evaluated by first measuring its raw performance and then studying the impact of hardware characteristics on the convergence of algorithms. In this way, the former results can be weighted by the latter.

7.1 Metrics

Two types of metrics are used. The first one describes how well the hardware is used (utilization rate and idle time) and is mainly of interest for the computer architect, while the second type characterizes the performance on ANN applications and is chiefly intended for the end-user.

7.1.1 Hardware utilization rate

The hardware *utilization rate* U is defined as the proportion of actively used PEs averaged over time. For a multi-processor machine with n_{PE} units, this ratio is defined as:

$$U = \frac{\sum_{t_{\mathrm{clk}}=1}^{t_{\mathrm{clk,\,prgm}}} n_{\mathrm{PE,\,used}}\left(t_{\mathrm{clk}}\right)}{t_{\mathrm{clk,\,prgm}} \cdot n_{\mathrm{PE}}} \tag{7.1}$$

where $t_{\mathrm{clk,\,prgm}}$ is the execution time of a given program measured in clock cycles, and $n_{\mathrm{PE,\,used}}\left(t_{\mathrm{clk}}\right)$ is the number of PEs used during cycle t_{clk}. Since there is no parallelization overhead at the PE level in the MANTRA I machine, the utilization rate is equal to the parallelization efficiency.[1]

As mentioned in section 5.2.3, performance degradation can be of *static* or *dynamic* nature. The static utilization rate U_{stat} is given by equation (7.1) under the assumption that the SIMD module of the machine is kept busy, while the dynamic utilization rate U_{dyn} is defined as the ratio between the amount of clock cycles that the SIMD module is used and the total number of cycles. Knowing an algorithm and its parameters, the former metric can be calculated while the latter is measured by a performance monitor (see section 5.2.3). The effective utilization rate is equal to the product of its two components: $U = U_{\mathrm{stat}} \cdot U_{\mathrm{dyn}}$.

Although the penalty of filling and emptying the GENES IV array (pipeline), at the beginning and end of the computation, affects the static utilization rate, this influence is usually negligible. Hence, this rate is mainly limited by network topologies that are not mapped well on the array (i.e., when the number of neurons and inputs are not multiples of the number of PEs per edge) and by epoch lengths smaller than the required minimum.

7.1.2 Microprocessor's idle time

The *idle time* t_{idle} is defined as the amount of clock cycles that the control microprocessor is waiting for results from the SIMD module or to gain access to a shared resource (e.g., the FIFOs when empty/full or the sigma and function-of-Y units). This figure has little significance, unless the total execution time t_{prgm} of a program is known. In this chapter, only the *relative idle time*, defined as the ratio $\frac{t_{\mathrm{idle}}}{t_{\mathrm{prgm}}}$, is considered. This metric corresponds to the utilization rate of the microprocessor. However, since this quantity has only a temporal but no spatial nature, the term "relative idle time" is preferred to "utilization rate."

7.1.3 Performance of ANN algorithms

Two metrics are traditionally used to evaluate the performance of ANN algorithms and computers: the number of *connections per second* or *CPS* for the evaluation phase and the number of *connection updates per second* or *CUPS* for the learning phase. These metrics suffer from the same problems that affect most units in computer performance evaluation and, above all, the number of FLOPS. The main drawback is that these metrics tend to be used in the absence of a clear and unambiguous definition. In their book [HEN90], J. Hennessy and D. Patterson write:

[1] The parallelization efficiency is defined as the ratio between the speed-up of a parallel algorithm and the number of processors.

> *Another perceived problem is that the MFLOPS rating changes [...] on the mixture of fast and slow floating-point operations. For example, a program with 100 % floating-point adds will have a higher rating than a program with 100 % floating-point divides.*

While these authors decided to avoid the use of FLOPS altogether, CPS and CUPS are too popular to be simply rejected. To avoid irrelevant figures in this thesis, these metrics are always associated with a given model, because different models imply different types of computation. When the relative amount of computation also depends on the network's topology (number of layers, number of neurons per layer, etc.), these characteristics should also accompany the CPS or CUPS rating.

The sustained number of CPS or CUPS on a given problem is defined as the number of connections C that should be evaluated (i.e., C^{eval}) or updated (i.e., C^{lrn}), divided by the total execution time t_{prgm} of this problem: $P = \frac{C}{t_{\text{prgm}}}$. When the network's output is evaluated on a set of prototypes, the performance P^{eval} is expressed in CPS. On the other hand, the performance P^{lrn} is expressed in CUPS when the problem involves learning. Therefore, these metrics account not only for the actual computation or update time of a connection, but also for a proportional amount of the activation function's evaluation time and of any other overhead.

This definition of CUPS also takes into account the evaluation phase required to update the weights. Hence, for the same network and the same set of inputs, the CUPS rating is always lower than the CPS rating. This is the most common definition, but the reader should be aware that some authors consider only the extra computation necessary to update the weights, after the evaluation phase has been completed.

The number of connections C depends on the underlying model. Equations (7.2), (7.3), (7.4), and (7.5) give this quantity for an m-neuron n-input single-layer network, type I and II recurrent networks with the same parameters, and an L-layer feed-forward network, respectively:

$$C_{\text{sgl}} = m \cdot n^* \cdot S \tag{7.2}$$

$$C_{\text{rec I}} = m^* \cdot n^* \cdot S + m^{*2} \cdot \sum_{s=1}^{S} \tau_{\text{stable}}(s) \tag{7.3}$$

$$C_{\text{rec II}} = \left(m \cdot n^* + m^2\right) \cdot \sum_{s=1}^{S} \tau_{\text{stable}}(s) \tag{7.4}$$

$$C_{\text{mlt}} = S \cdot \sum_{k=1}^{L} m_k^* \cdot m_{k-1}^* \tag{7.5}$$

where S is the number of prototypes presented to the network. If the same vectors are repeatedly used, each presentation counts as a new prototype. In recurrent networks, $\tau_{\text{stable}}(s)$ is the number of iteration to reach convergence with prototype s. All networks are considered as fully-connected. For single-layer networks, this means that all inputs are connected to all neurons, and that all neurons are interconnected if the network is recurrent. In multi-layer networks, connections link all neurons of a layer to all those of the next one. Missing connections of partially-connected networks should be removed from equations (7.2) to (7.5). However, the MANTRA I machine supports only fully-connected networks, and non-existent connections are implemented by null weights with a corresponding performance penalty.

If thresholds are treated as weights (see section 1.4), additional connections due to constant inputs and pseudo-neurons are also counted. This explains the $*$-notation in equations (7.2) to (7.5).

While all connections are relevant to calculate the CPS rating, only plastic weights should be taken into account for the CUPS rating. For all models presented in chapter 1, except the Kohonen algorithm with recurrent network (see section 1.10.2), the learning process affects all weights, and the number of connections is identical in recall and learning modes: $C^{\text{eval}} = C^{\text{lrn}}$. For the Kohonen model with recurrent network, the number of synapses is given by equation (7.4) in evaluation mode: $C^{\text{eval}}_{\text{khn, rec II}} = C_{\text{rec II}}$ (or by equation (7.3), if a type I recurrent network is used: $C^{\text{eval}}_{\text{khn, rec I}} = C_{\text{rec I}}$) and by equation (7.2) when the network is trained: $C^{\text{lrn}}_{\text{khn, rec}} = C_{\text{sgl}}$.

Finally, it should be emphasized that — under the proposed definition — all weights are relevant to the CUPS rating of the Kohonen model with minimum, and not only the weights belonging to the winner's neighborhood. This definition is consistent with the Kohonen algorithm with recurrent network, where the neighborhood is variable and data-dependent. However, some authors take into account only the connections that are actually updated.

7.2 Peak performance

The performance of the MANTRA I machine can be expressed by simple relations. As far as sustained performance is concerned, this is of little use since they are given in terms of the utilization rate presented in section 7.1.1. However, these formulae are used to discuss peak performance. For all networks featuring a constant amount of computation per connection (i.e., single-layer or recurrent networks), the expression for the performance P_{cst} is similar to equations (3.15) and (3.21):

$$P_{\text{cst}} = \frac{N^2 \cdot f}{N_{\text{PS}} \cdot n_{\text{op}}} \cdot U \tag{7.6}$$

where N^2 is the number of PEs, f the clock frequency, $N_{\text{PS}} = 40$ the number of clock cycles per macro-cycle, and n_{op} the number of operations to compute or update a connection. The parameter n_{op} is, for instance, equal to 1 for single-layer or Hopfield networks in evaluation mode (see sections 1.4.1 and 1.9), to 2 for the Perceptron and delta learning rules (see sections 1.6 and 1.7) or for the Kohonen model with minimum in recall phase (see section 1.10.1), and to 4 for the same Kohonen network being trained.

For multi-layer networks or combinations of direct and recurrent networks, this simple formula is only valid when the factor n_{op} is constant for all synapses. This property is satisfied, for instance, by multi-layer feed-forward networks in evaluation phase (see section 1.4.3), where $n_{\text{op}} = 1$. When this is not the case, the number of operations required to compute or update a connection should be weighted by the relative amount of computation:

$$P = \frac{N^2 \cdot f \cdot \overbrace{C}^{C^{\text{eval}}}}{N_{\text{PS}} \cdot \sum_{c=1} n_{\text{op}}(c)} \cdot U \tag{7.7}$$

where C is the number of evaluated or updated connections (C^{eval} or C^{lrn}).

Among the ANN algorithms presented in chapter 1, only the back-propagation rule and the Kohonen model with recurrent network (see sections 1.8 and 1.10.2) are incompatible with equation (7.6). Since the former model is based on a multi-layer feed-forward network, equation (7.7)

becomes:

$$P_{\text{mlt}} = \frac{N^2 \cdot f \cdot \sum_{k=1}^{L} m_k^* \cdot m_{k-1}^*}{N_{\text{PS}} \cdot \sum_{k=1}^{L} m_k^* \cdot m_{k-1}^* \cdot n_{\text{op}}^{[k]}} \cdot U \tag{7.8}$$

where $n_{\text{op}}^{[k]}$ is the number of operations per connection on layer k. In learning mode, it is equal to $n_{\text{op}}^{[1]} = 2$ for the first layer, and to $n_{\text{op}}^{[k]} = 3$ on the following layers $k = 2, 3, \ldots, L$.

Since only the direct connections of the Kohonen model with recurrent network are plastic, different formulae are required to calculate the CPS and CUPS ratings. In evaluation mode, the performance is given by equation (7.9) for type I recurrent networks and by equation (7.10) for type II:

$$P_{\text{khn, rec I}}^{\text{eval}} = \frac{N^2 \cdot f \cdot \left(n \cdot S + m \cdot \sum_{s=1}^{S} \tau_{\text{stable}}(s) \right)}{N_{\text{PS}} \cdot \left(n \cdot S \cdot n_{\text{op, sgl}}^{\text{eval}} + m \cdot \left(\sum_{s=1}^{S} \tau_{\text{stable}}(s) \right) \cdot n_{\text{op, rec I}}^{\text{eval}} \right)} \cdot U \tag{7.9}$$

$$P_{\text{khn, rec II}}^{\text{eval}} = \frac{N^2 \cdot f \cdot (n + m)}{N_{\text{PS}} \cdot \left(n \cdot n_{\text{op, sgl}}^{\text{eval}} + m \cdot n_{\text{op, rec II}}^{\text{eval}} \right)} \cdot U \tag{7.10}$$

where $n_{\text{op, sgl}}^{\text{eval}} = 1$, $n_{\text{op, rec I}}^{\text{eval}} = 1$, and $n_{\text{op, rec II}}^{\text{eval}} = 1$. In learning mode, the performance of type I and II networks is respectively:

$$P_{\text{khn, rec I}}^{\text{lrn}} = \frac{N^2 \cdot f \cdot n \cdot S}{N_{\text{PS}} \cdot \left(n \cdot S \cdot n_{\text{op, sgl}} + m \cdot \left(\sum_{s=1}^{S} \tau_{\text{stable}}(s) \right) \cdot n_{\text{op, rec I}}^{\text{eval}} \right)} \cdot U \tag{7.11}$$

$$P_{\text{khn, rec II}}^{\text{lrn}} = \frac{N^2 \cdot f \cdot n \cdot S}{N_{\text{PS}} \cdot \left(\left(\sum_{s=1}^{S} \tau_{\text{stable}}(s) \right) \cdot \left(n \cdot n_{\text{op, sgl}}^{\text{eval}} + m \cdot n_{\text{op, rec II}}^{\text{eval}} \right) + n \cdot n_{\text{op, sgl}}^{\text{lrn}} \cdot S \right)} \cdot U \tag{7.12}$$

where $n_{\text{op, sgl}}^{\text{lrn}} = 1$ and $n_{\text{op, sgl}} = n_{\text{op, sgl}}^{\text{eval}} + n_{\text{op, sgl}}^{\text{lrn}} = 2$. When the recurrent network is iterated until convergence is reached, the parameter τ_{stable} is data-dependent and equations (7.9), (7.11), and (7.12) are of little practical use. However, if the convergence process is approximated by a fixed number of iterations τ_{max}, these formulae can be simplified to:

$$P_{\text{khn, rec I}}^{\text{eval}} = \frac{N^2 \cdot f \cdot (n + m \cdot \tau_{\text{max}})}{N_{\text{PS}} \cdot \left(n \cdot n_{\text{op, sgl}}^{\text{eval}} + m \cdot \tau_{\text{max}} \cdot n_{\text{op, rec I}}^{\text{eval}} \right)} \cdot U \tag{7.13}$$

$$P_{\text{khn, rec I}}^{\text{lrn}} = \frac{N^2 \cdot f \cdot n}{N_{\text{PS}} \cdot \left(n \cdot n_{\text{op, sgl}} + m \cdot \tau_{\text{max}} \cdot n_{\text{op, rec I}}^{\text{eval}} \right)} \cdot U \tag{7.14}$$

$$P_{\text{khn, rec II}}^{\text{lrn}} = \frac{N^2 \cdot f \cdot n}{N_{\text{PS}} \cdot \left(\tau_{\text{max}} \cdot \left(n \cdot n_{\text{op, sgl}}^{\text{eval}} + m \cdot n_{\text{op, rec II}}^{\text{eval}} \right) + n \cdot n_{\text{op, sgl}}^{\text{lrn}} \right)} \cdot U \tag{7.15}$$

Model	Evaluation mode		Learning mode	
	n_{op}	MCPS	n_{op}	MCUPS
Perceptron and delta rules	1	80 / 400	2	40 / 200
Back-propagation rule (lower bound)	1	80 / 400	3	26 / 133
Kohonen model with minimum	2	40 / 200	4	20 / 100

Table 7.1: *Peak performance of the MANTRA I machine for the 400-PE prototype machine running at 8 MHz and for the 1600-PE full-scale machine running at 10 MHz.*

The peak performance ($U = 100\,\%$) of the machine on a given algorithm can be directly derived from above formulae. This figure should be discussed very carefully. To the end-user, it means nothing, the only valuable characteristic being the sustained performance. On the other hand, to the computer architect or the software designer, peak performance provides a good idea of the underlying technology and the intrinsic possibilities of a machine. Moreover, the only way to judge the adequacy of an architecture to solve a given problem or to tune a program is to study performance losses, what requires the knowledge of peak performance.

Table 7.1 gives the peak performance of the machine for the Perceptron and delta rules (see algorithms 6.1 and 6.2), for the back-propagation rule (see algorithms 6.4 and B.1), and for the Kohonen model with minimum (see algorithms 6.6 and B.2). The two values appearing for each algorithm as CPS and CUPS ratings correspond to the peak performance of the current and full-scale configurations (see section 5.4). The former has a GENES IV array of 20×20 PEs running at $f = 8\,\text{MHz}$, while the latter features 40×40 PEs and runs at $f = 10\,\text{MHz}$. As shown by equation (7.8) the learning performance of the back-propagation rule depends on the network topology. It can easily be seen that equation (7.6) with $n_{op} = 3$ provides a lower bound for the peak performance. For instance, for a two-layer network with the same number of inputs and of neurons on each layer, this figure becomes 32 MCUPS and 160 MCUPS depending on the configuration. For a three-layer network with the same characteristics, it decreases to 30 MCUPS and 150 MCUPS.

7.3 Sustained performance for the delta rule

The aim of this section is to study how the different parameters of a model influence the performance. This analysis is based on measurements made with the delta rule, which is the first algorithm ported on the MANTRA I machine. The first version suffers from a certain number of restrictions:

1. Virtual networks are not implemented (this version corresponds to algorithm 6.1). Hence, the number of neurons and of inputs is limited by the size of the machine (i.e., maximum 20 neurons and 20 inputs on the current prototype).
2. All data are stored in the XY, auxiliary Y, and desired output memories (see section 5.1.4). This restricts the number of prototypes to 1638 for a 20-neuron and 20-input network.
3. All data corresponding to an epoch must fit in half of the X and desired output FIFOs, what limits the epoch length to 102 for the same network.

While points 1 and 2 are important restrictions that should be removed in future versions of the software, point 3 does not represent a heavy drawback since short epochs are usually favored.[2] The last requirement is enforced as an easy way to ensure a good overlap between the computation of the microprocessor and of the SIMD module. Points 1 and 2 affect not significantly the results of this experiment, since the goal is to point out which mechanisms cause performance losses and not to benchmark a real application.

The measurements are started with the input data in the microprocessor's memory and stopped when the results are stored in the same place. This means that only the communication overhead between the control and SIMD modules is accounted for. Since the raw performance (i.e., not accounting the convergence speed) is data-independent, inputs are randomly generated. The effect of convergence is studied in section 7.4.

All measurements involve a network with $m = 20$ neurons and $n = 20$ inputs. Figure 7.1 shows the performance of the machine as a function of the epoch length, for 100, 400, and 1600 prototypes. For each of these values, a first curve (the lowest) has been measured for one presentation of the whole training set, while a second one (the highest) shows the learning performance for 100 successive presentations of the same training set. In both cases the results are unloaded at the end. Figures 7.1(a), (b), (c), and (d) present respectively the calculated static utilization rate U_{stat}, the measured dynamic utilization rate U_{dyn}, the microprocessor's relative idle time $\frac{t_{\text{idle}}}{t_{\text{prgm}}}$, and the learning performance $P^{\text{lrn}}_{\text{delta}}$ expressed in MCUPS. The total utilization rate U, that is, the product of the static and dynamic rates, has not been plotted, since it is directly proportional to the learning performance shown in figure 7.1(d). For all the experiments presented in this section, the difference between these two figures (i.e., U and $P^{\text{lrn}}_{\text{delta}}$) after normalization has been found smaller than 9 ppm. This happens because the dynamic utilization rate is measured by the performance monitor (see section 5.2.3) while the execution time is recorded by the microprocessor's timer, two units that are driven by different oscillators.

Figures 7.2(a), (b), (c), and (d) present the same information as a function of the number of prototypes, for 1, 10, 100, 1000, and 10000 presentations. The epoch length has been chosen equal to 50, and only multiples of this value have been used as number of prototypes, to avoid the saw-tooth side-effect of figure 7.1 (see section 7.3.2).

The same metrics are plotted in figure 7.3 as a function of the check-point interval, for three different epochs of 50, 75, and 100 prototypes. The smallest common multiple of these values (i.e., 300) has been chosen as the number of prototypes. This avoids also the saw-tooth side-effect. The number of presentations is 10000. In this experiment, check-points are emulated by simply unloading the output vectors of the whole learning set. In figure 7.3 the check-point interval is measured in number of presentations.

7.3.1 Influence of the number of neurons and inputs

The influence of the number of neurons and inputs on performance does not appear in figures 7.1 to 7.3. The reason is that these parameters affect mainly the static utilization rate, and hence the outcome is highly predictable. For the tested algorithm (i.e., without virtual matrices), the static

[2] With the second version of the control board (see section 5.4), which features blocking FIFO accesses, this limit will easily be extended to the full size of the FIFOs. On the current implementation, this would require the additional overhead of testing the FIFO full flag before each access.

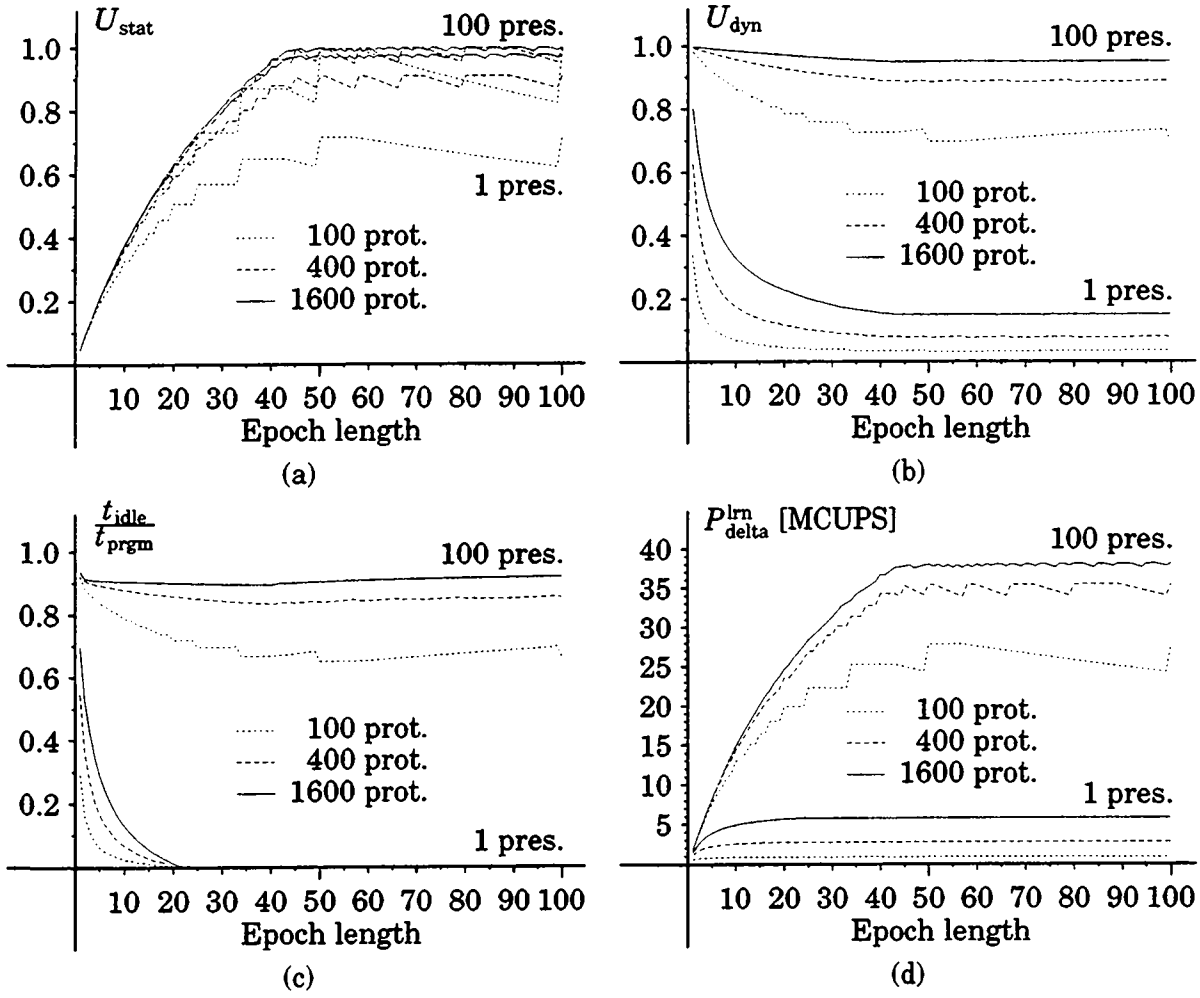


Figure 7.1: *Performance as a function of the epoch length (400-PE MANTRA I machine running at 8 MHz).* (a) *Static utilization rate.* (b) *Dynamic utilization rate.* (c) *Microprocessor's relative idle time.* (d) *Learning performance.*

utilization rate should be multiplied by a factor $\frac{m \cdot n}{N^2}$. If a virtual weight matrix is used, this factor is given with respect to the next larger network that fits exactly on the array:

$$\frac{m \cdot n}{\left\lceil \frac{m}{N} \right\rceil \cdot \left\lceil \frac{n}{N} \right\rceil \cdot N^2}$$

7.3.2 Influence of the epoch length

The main influence of the epoch length on the static utilization rate can be seen in figure 7.1 (a). This rate increases monotonically as long as the epoch size is smaller than the pipeline depth (i.e., $2N + 3 = 43$). The curves have a saw-tooth shape due to the way the algorithm is implemented, that is, when the number of prototypes is not a multiple of the epoch length, the last epoch is accordingly smaller, as shown by equation (1.25). When the amount of prototypes is increased, this effect becomes less noticeable. Another consequence of this implementation choice is that the maximum of the static utilization rate is reached for the first divisor of the number of prototypes

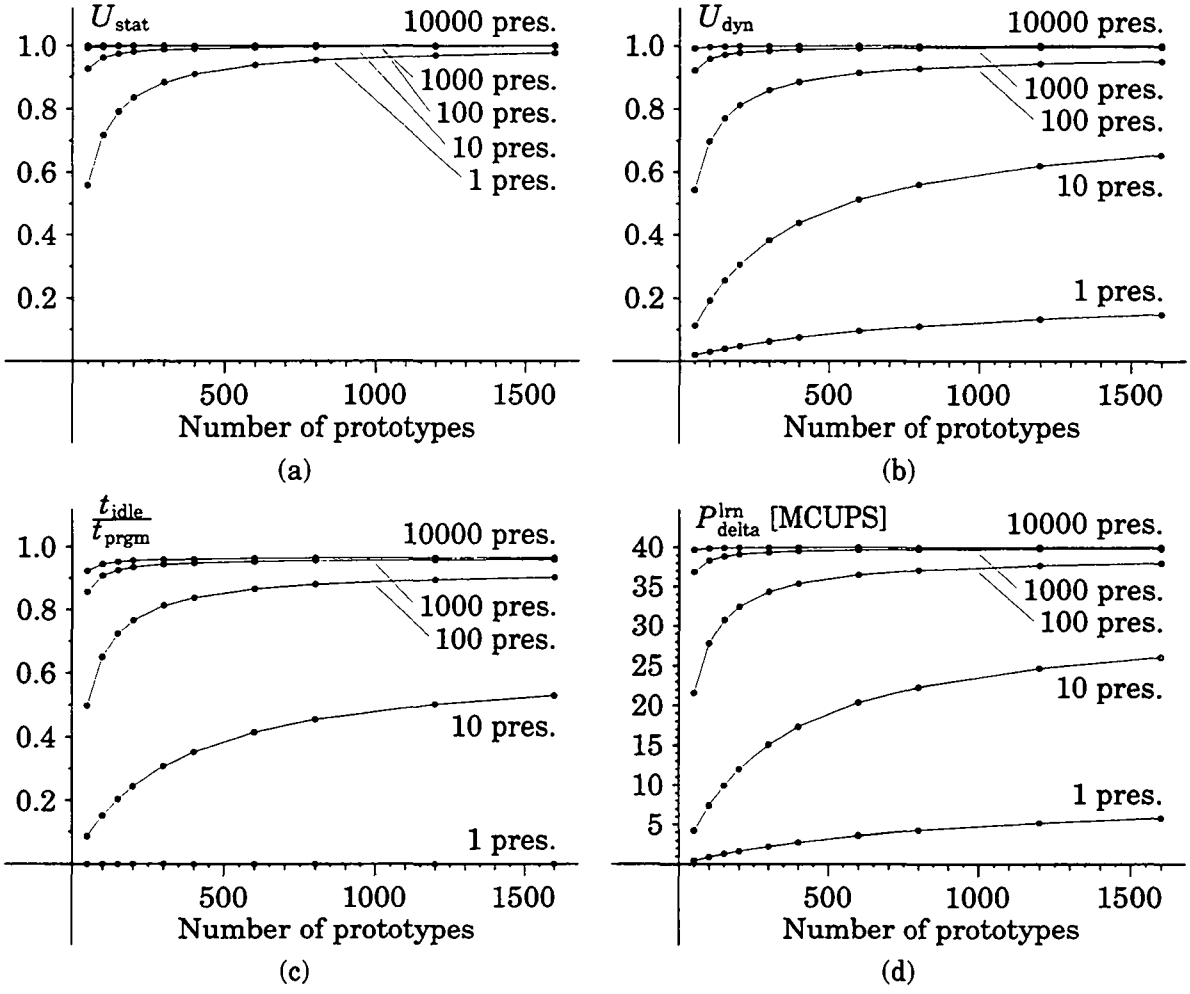


Figure 7.2: *Performance as a function of the number of prototypes (400-PE MANTRA I machine running at 8 MHz).* (a) *Static utilization rate.* (b) *Dynamic utilization rate.* (c) *Microprocessor's relative idle time.* (d) *Learning performance.*

that is higher than $2N + 3 = 43$. In figure 7.1 (a), this value is equal to 50.

The influence of the epoch length on the dynamic utilization rate is shown in figure 7.1 (b). To a much lesser extent, the saw-tooth effect is also observed. For long epochs, the loading and unloading operations do not overlap well with the SIMD module's computation, resulting in a poorer utilization. Since these operations take place only during the first and the last presentation, respectively, this phenomenon is less sensitive to large numbers of presentations. These observations are consistent with the idle-time curves shown in figure 7.1 (c).

Besides the saw-tooth effect, the use of epochs larger than the pipeline depth has a negligible influence, as shown by figure 7.3.

7.3.3 Influence of the number of prototypes

As mentioned in section 7.1.1, the static utilization rate is influenced by loading and unloading the weight matrix, as well as by filling and emptying the pipeline. However, this effect is only noticeable when the number of prototypes and the number of presentations are both very small.

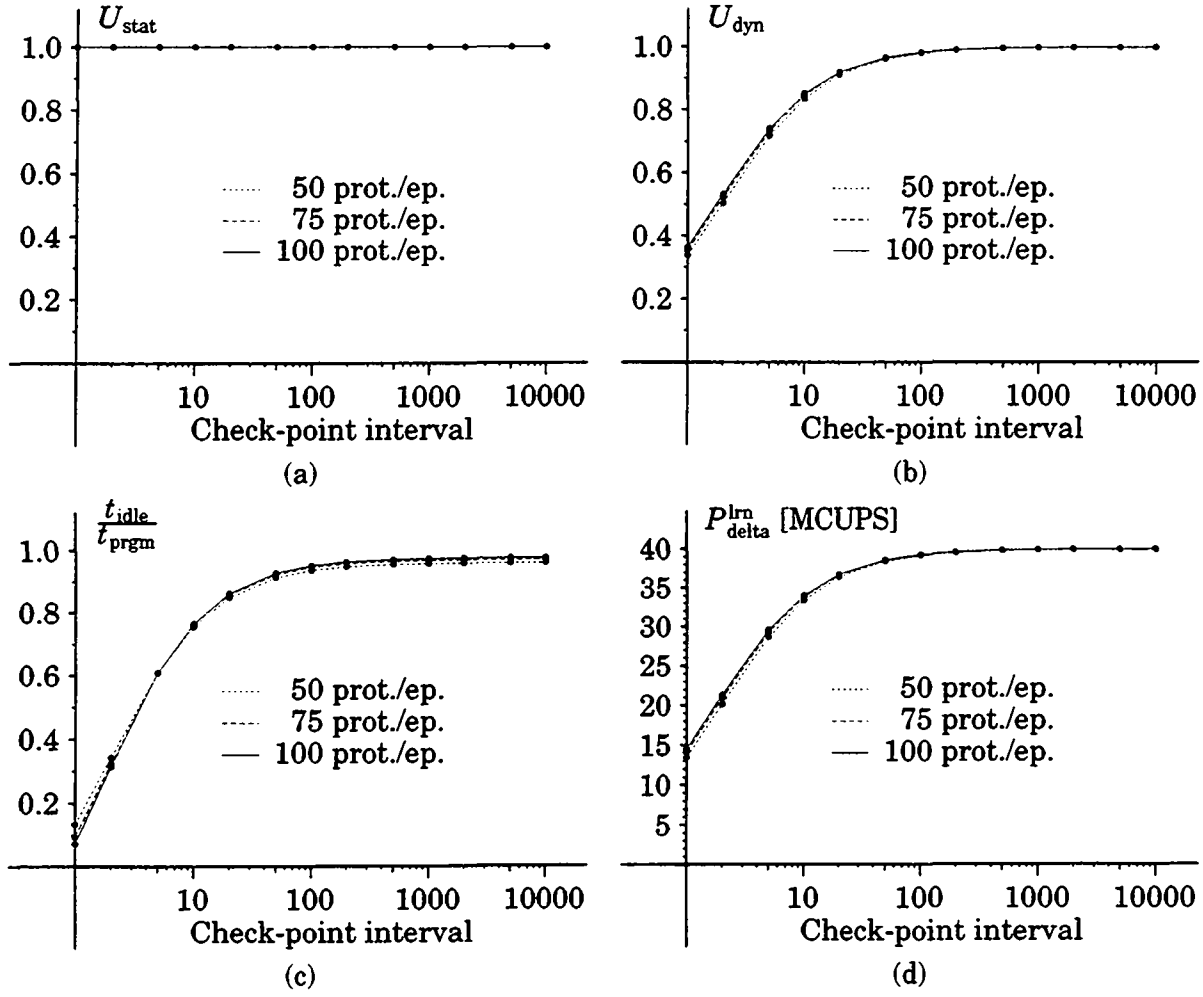


Figure 7.3: *Performance as a function of the number of presentations between check-points (400-PE MANTRA I machine running at 8MHz).* (a) *Static utilization rate.* (b) *Dynamic utilization rate.* (c) *Microprocessor's relative idle time.* (d) *Learning performance.*

This can be clearly seen in figure 7.2(a). Otherwise, the number of prototypes has no influence on the static utilization rate.

The number of prototypes also affects the dynamic utilization rate, as shown in figure 7.2(b). This phenomenon is less obvious to understand because the epoch length is kept constant. It is similar to what happens with the static utilization rate since it is due to the initial loading and final unloading of the weight matrix. These operations do not only consist in transferring the matrix to or from the weight FIFOs (see section 5.1.4), but also in converting the data from parallel to serial form or vice-versa. These are rather long operations that are serialized with the computation of the SIMD module. Figures 7.4(a) and (b) show the dynamic utilization rate, and the learning performance under the same conditions as figure 7.2, but without weight-matrix loading and unloading. As expected, the curves of the dynamic utilization rate are almost flat.

An important factor is neglected in the measurements shown in figure 7.2. As explained at the beginning of this section, the first version of the software requires that the whole training set fits in the memories of the SIMD module. For larger problems, the control microprocessor would have to store a part of the prototypes in its memory and supply them to the array through

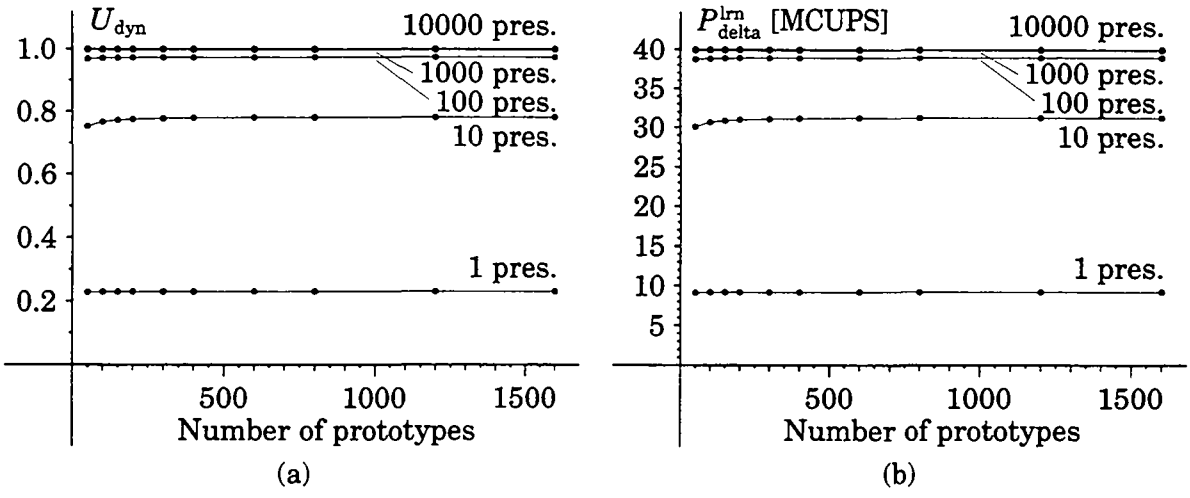


Figure 7.4: *Performance as a function of the number of prototypes, without weight-matrix loading and unloading (400-PE MANTRA I machine running at 8 MHz).* (a) *Dynamic utilization rate.* (b) *Learning performance.*

the FIFOs. The performance of a non-optimized routine would then be close to the curves for one presentation shown in figure 7.2. Advanced software techniques should improve these poor results. This involves the use of DMA channels to load or unload FIFOs and the interleaving of data accesses in the microprocessor and array's memories.

7.3.4 Influence of the number of presentations

Figures 7.1 and 7.2 clearly show the influence of the number of presentations on both utilization rates and on the microprocessor's idle time. Since most of the overhead occurs either at beginning or at the end of the computation, this influence decreases the more the training set is presented. However, this is only true when there are no check-points, as it is the case in figures 7.1 and 7.2.

7.3.5 Influence of the check-point interval

As mentioned above, in most application it is desirable to analyze the state of the network at regular intervals. In figure 7.3, check-points have been modeled by down-loading the output of the evaluation phase on the whole training set. In many cases it would be preferable to perform this task on a distinct and smaller test set. However, as far as the communication overhead is concerned, this is essentially the same. The purpose of such an operation could be to compute some statistics on the error (mean, mean square, minimum, maximum, standard deviation, etc.). No computation (other than storing the output matrix in memory) has been performed to measure the curves of figure 7.3.

As expected, the check-point interval has no influence on the static utilization rate plotted in figure 7.3(a). The influence on the dynamic utilization rate, shown in figure 7.3(b), is also very intuitive. Each check-point represents some overhead because the computation of the microprocessor is not totally overlapping that of the SIMD module. When these check-points get spread further apart, the overhead becomes less perceptible. A similar phenomenon occurs for the micro-

processor's idle time, presented in figure 7.3 (c). The more check-points are spread apart, the more the microprocessor is idle.

7.4 Convergence analysis of the delta rule

ANN algorithms are affected in three ways when implemented on the MANTRA I machine:

1. The learning coefficient α, does not assume a new value for each prototype, but only a restricted set of values can be used.
2. Epoch updating is used.
3. All data are represented as integers.

When studying their impact on convergence, the first two points are of a very different nature than the last one. The reason is that the first two restrictions are imposed only by performance considerations, while the third one is an intrinsic characteristic of the machine. At the price of a very slow execution, it is possible to run benchmarks in on-line mode (i.e., with an epoch size of 1), and/or to change the learning coefficient after each epoch (or prototype in on-line mode). On the other hand, the consequences of integer representation should be analyzed by comparison with the same problem run on a conventional floating-point microprocessor. To distinguish the impact of point 1 or 2 from that of point 3, the same experiments have been run on the MANTRA I machine and on the floating-point microprocessor.

The present analysis is based on the delta rule (see section 1.7). The results of this study are expected to be directly applicable to the back-propagation rule. The same extrapolation is less obvious for the Kohonen model. However, since this model was in the process of being ported to the machine at the time of writing, this analysis had to be delayed.

Since the delta rule is a linear separator, the chosen benchmark is a classification problem with linearly separable regions, to which some noise has been added. This slight deviation from a purely linear problem has been introduced to exercise the robustness of ANNs. A network with $m = 20$ neurons and $n^* = 100$ inputs has been chosen. A precise description of this benchmark is given in appendix C.2.

All results presented here are in the form of an average quadratic error:

$$\xi_{\text{avr}}(t) = \frac{1}{S} \cdot \frac{1}{m} \cdot \sum_{s=1}^{S} \sum_{i=1}^{m} \varepsilon_i^2(t,s) = \frac{1}{S} \cdot \frac{1}{m} \cdot \sum_{s=1}^{S} \sum_{i=1}^{m} (d_i(s) - y_i(t,s))^2 \tag{7.16}$$

where the number of prototypes S is either equal to $S_{\text{lrn}} = 10000$, for the training set, or to $S_{\text{tst}} = 1000$, for the testing set. This error is plotted as a function of the number of presentations ρ of the whole learning set. This quantity is related to the update time t by the relation $t(\rho) = \lceil \frac{S_{\text{lrn}}}{e} \rceil \cdot \rho$, where e is the epoch length. A logarithmic scale has been chosen to display the whole curve and, at the same time, to emphasize the differences between the asymptotic values.

7.4.1 Influence of discrete values of the learning coefficient

The first constraint imposed by the MANTRA I machine is the use of a small set of values for the learning coefficients. As shown by equation (6.13), the learning coefficient α is combined

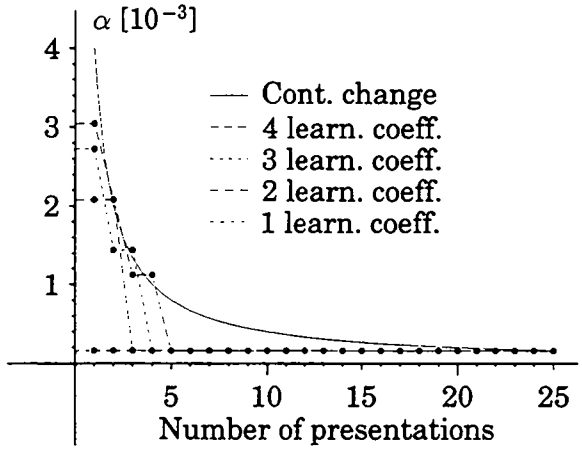


Figure 7.5: *Learning-coefficient functions.*

with the derivative of the activation function σ' in the function-of-Y table. Loading this table requires stopping the SIMD module. This is an expensive process that should be avoided whenever possible. In the case of the delta rule, four tables can be prepared in advance and swapped during computation.

In most applications, the learning coefficient α follows a time-decreasing function, such as:

$$\alpha(t) = \frac{\alpha_0}{1 + \frac{t-1}{t_0}} \tag{7.17}$$

where α_0 is the value of the coefficient for the first update, and t_0 a parameter controlling the steepness of the function. For $t_0 = 1$, equation (7.17) is reduced to $\alpha(t) = \frac{\alpha_0}{t}$. To simplify the algorithm, it has been considered that changing the coefficient after each presentation ρ (rather than each epoch t) is a fair approximation of a continuous function. A few simulations have shown that the function $\alpha(\rho) = \frac{0.004}{\rho}$ performs well on the target benchmark. This function can then be approximated by one to four values as shown in figure 7.5. The continuous function is considered as an approximative upper bound of the discrete ones.

Figure 7.6 shows the results of the benchmark in on-line mode (i.e., $e = 1$) for the different learning-coefficient functions. Figures 7.6(a) and (c) show the quadratic error ξ_{avr} on the training set, while figures 7.6(b) and (d) display it for the testing set. Figures 7.6(a) and (b) correspond to the floating-point version of the program, and figures 7.6(c) and (d) to the MANTRA I implementation. Although the use of a single coefficient affects the convergence, it can be seen that there is no significant difference with the continuous case, as long as at least two values are used.

The 4-coefficient version of the algorithm is used for the benchmarks presented in sections 7.4.2 and 7.4.3. In figure 7.5, it can be observed that the first value of the 3 and 4-coefficient versions is used only during the first presentation. This period is not as short as it may seem, since it corresponds to $\lceil \frac{S_{\text{lm}}}{e} \rceil$ epochs (i.e., 10000 in on-line mode, or 125 for an epoch length of $e = 80$).

7.4.2 Influence of epoch updating

The next restriction, due to the MANTRA I machine, is the use of epoch updating. Since this limitation is also imposed for performance reasons, benchmarks with different epoch lengths,

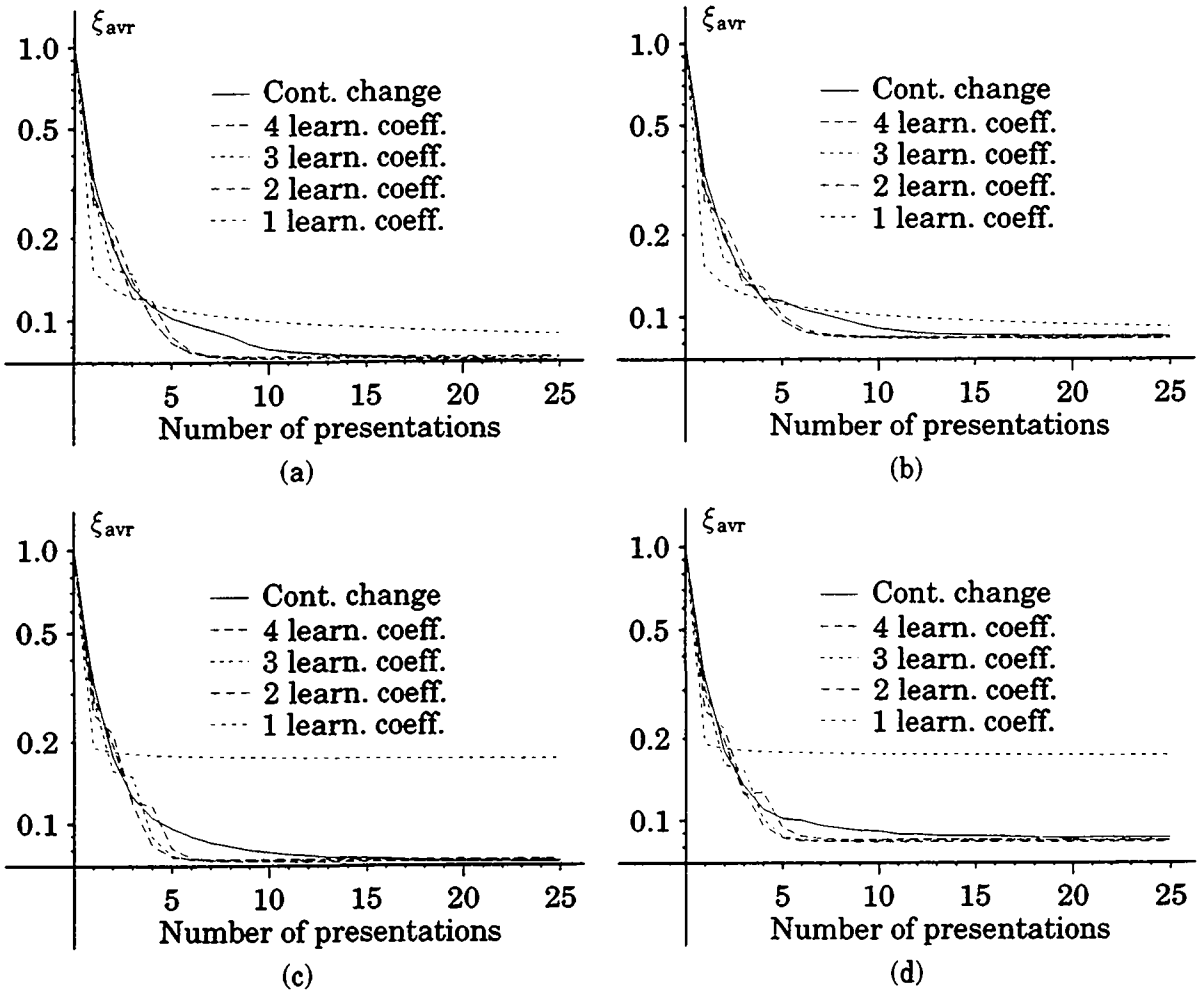


Figure 7.6: *Average quadratic error for different learning-coefficient functions (delta rule, $a_w = 0.8 \cdot a_{w,max}$, $a_x = \frac{a_{x,max}}{64}$, and $a_y = \frac{a_{y,max}}{64}$).* (a) *Training set (floating-point simulator).* (b) *Testing set (floating-point simulator).* (c) *Training set (MANTRA I machine).* (d) *Testing set (MANTRA I machine).*

including the on-line mode (i.e., $e = 1$), can be run. Here again, to distinguish this effect from that of using integers, the same problems are executed on a floating-point simulator, as well as on the MANTRA I machine.

The results are given in figure 7.7. These experiments show a very high robustness to reasonably short epoch lengths (i.e., $e \leq 100$). Section 7.3.2 shows that the machine is optimally used when the epoch size is larger than the pipeline depth $2N + 3$. If virtual matrices are used, the number of prototypes per epoch should be a multiple of $2N$ (see section 6.1 and appendix B). Since the number of PEs per edge is equal to $N = 20$ on the current prototype and to $N = 40$ on the full-scale machine, figure 7.7 shows that epoch updating does not significantly decrease the speed of convergence of the delta rule.

Again, these results are expected to be directly applicable to the back-propagation rule. Their validity for the Kohonen algorithm is less obvious. However, if the epoch length is small compared to the number of neurons, each of them will statistically be affected only by a few prototypes, and epoch updating should not have an important influence on the convergence (see section 1.10.1).

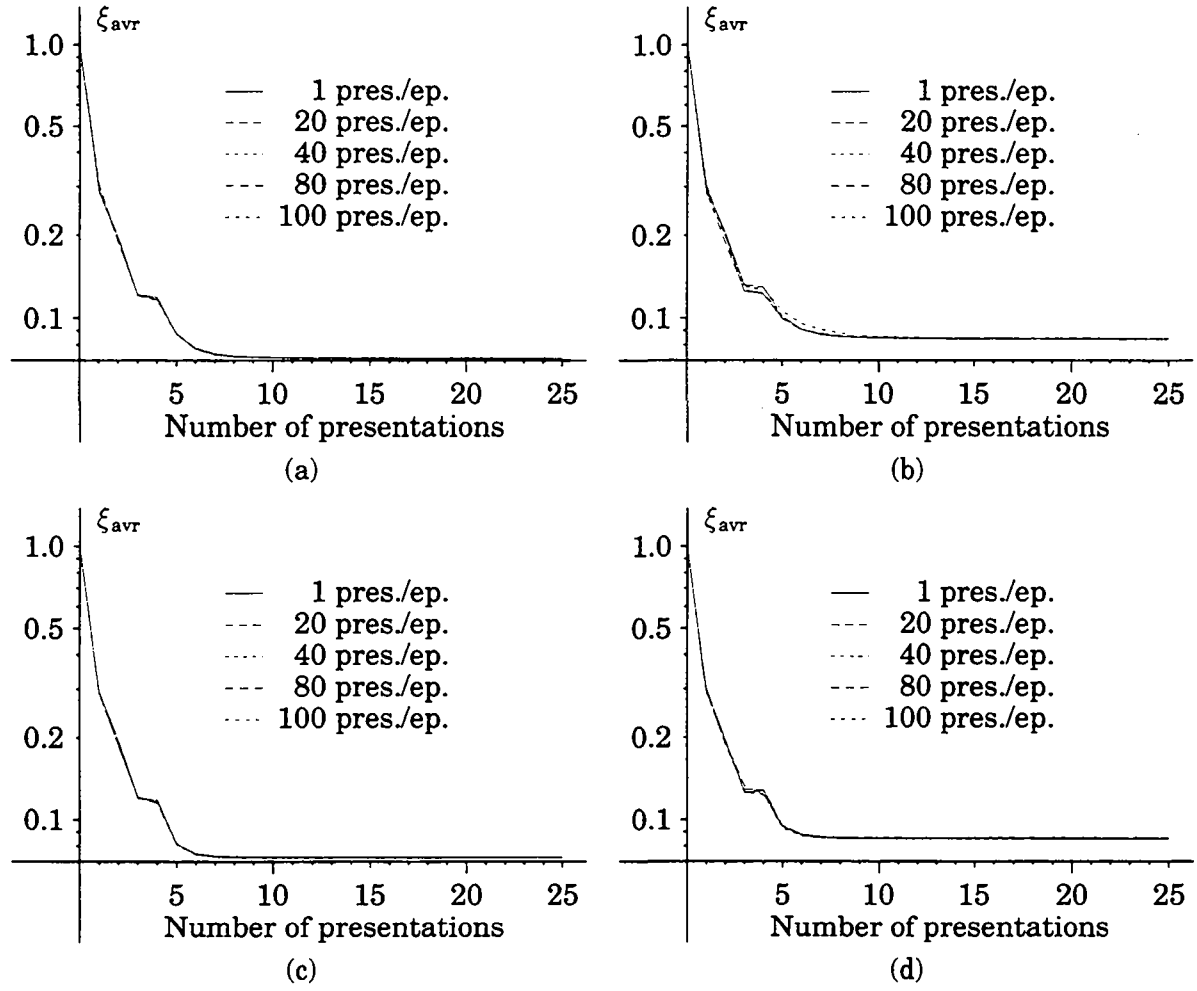


Figure 7.7: *Average quadratic error for different epoch lengths (delta rule, $a_w = 0.8 \cdot a_{w,\max}$, $a_x = \frac{a_{x,\max}}{64}$, and $a_y = \frac{a_{y,\max}}{64}$).* (a) *Training set (floating-point simulator).* (b) *Testing set (floating-point simulator).* (c) *Training set (MANTRA I machine).* (d) *Testing set (MANTRA I machine).*

7.4.3 Influence of integer data representation

The goal of this series of benchmarks is to study the influence of integer data representation, as well as the sensitivity to the scaling factors a_w, a_x, and a_y (see section 6.2). Figures 7.8(a), (c), and (e) shows the quadratic error on the training set, for the floating-point and MANTRA I versions of the algorithm, with an epoch length of $e = 80$. Figures 7.8(b), (d), and (f) show the same results for the testing set.

7.5 Performance comparison

Comparing the performance of different systems is a very delicate task. The only way to make a fair evaluation is to define a suite of benchmarks and to run it on different computers under the same conditions. Even so, the benchmarks should be very carefully chosen, not to bias the results by a poor choice of programs. For instance, systems with a small amount of very fast on-chip memory would be favored by small problems.

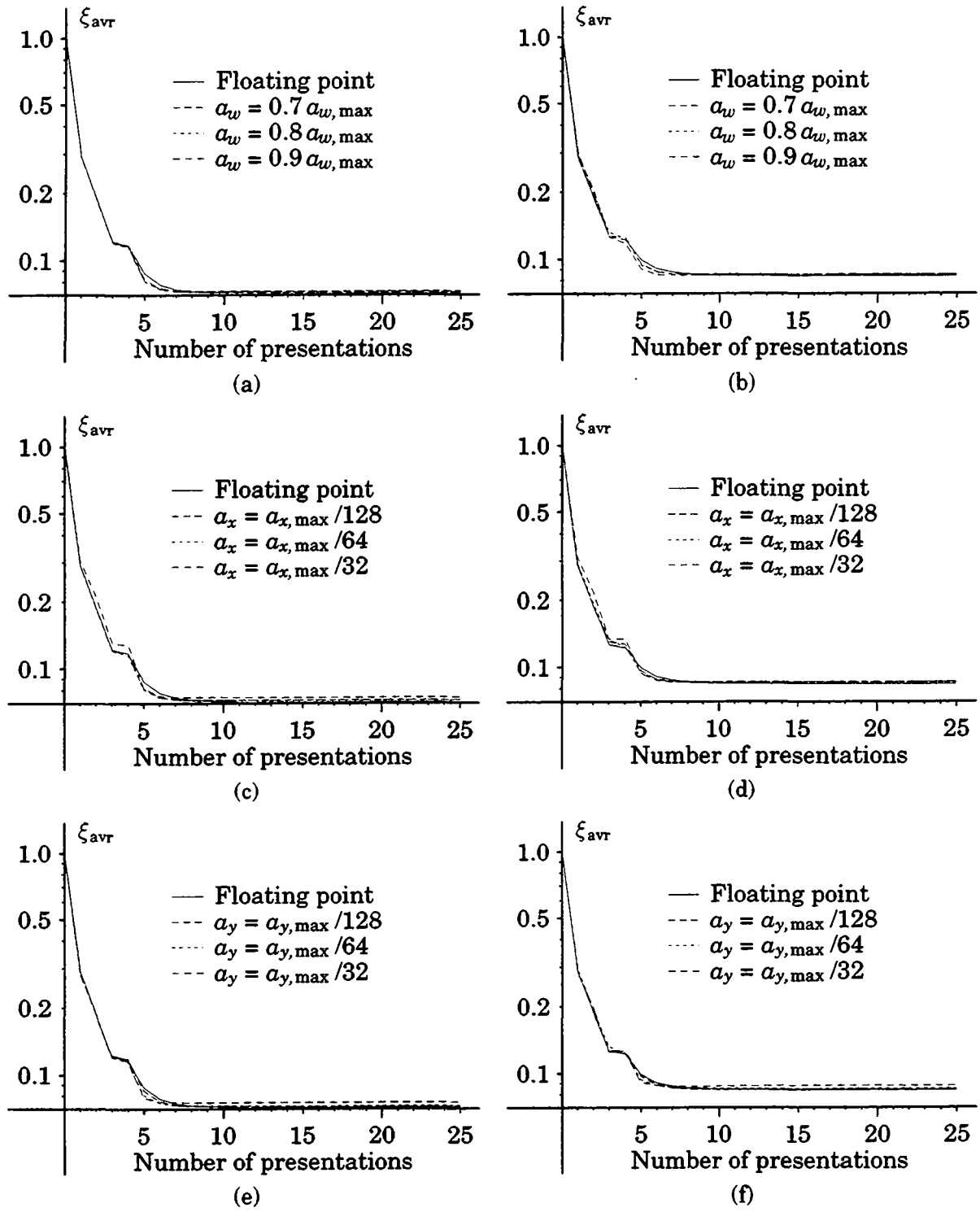


Figure 7.8: *Average quadratic error for floating-point and integer data representations (delta rule,* $a_w = 0.8 \cdot a_{w,\max}$, $a_x = \frac{a_{x,\max}}{64}$, *and* $a_y = \frac{a_{y,\max}}{64}$, *unless otherwise specified).* (a) *Training set.* (b) *Testing set.* (c) *Training set.* (d) *Testing set.* (e) *Training set.* (f) *Testing set.*

System	No. of PEs	Sustained perf. MCPS	Sustained perf. MCUPS	Peak perf. MCPS	Peak perf. MCUPS	Source
PC (i486, 50 MHz)	1	1.1	0.47	—	—	[MUL93]
SUN SPARCstation 10 (Model 30)	1	3.0	1.1	—	—	[MUL93]
Alpha APX station (DEC 3000 Model 500)	1	8.3	3.2	—	—	[MUL93]
Megaframe Hypercluster (IMS T800)	64	27	9.9	—	—	[MUH89]
Warp	10	—	17	—	—	[POM88]
CM-2	64000	180	40	—	—	[ZHA90]
Cray Y-MP C90	1	220	66	—	—	[MUL93]
RAP (TMS320C30, 16 MHz)	40	560	100	—	—	[MOR92]
NEC SX-3	1	—	130	—	—	[MUL93]
MANTRA I	1600	—	—	400	130	
MUSIC-20 (MC96002, 40 MHz)	60	500	250	—	—	[MUL93]
GF11	356	—	900	—	—	[WIT90]
SYNAPSE-1	32	—	—	5100	—	[RAM92]
CNAPS Server II	512	—	—	5800	1900	[ADA94]

Table 7.2: *Performance comparison for the back-propagation rule.*

Such an evaluation being beyond the scope of an architectural thesis, the present comparison is based on figures reported in the literature. Table 7.2 presents the performance of different machines for the back-propagation rule. This algorithm has been chosen because, for many systems, it is the only model for which values have been reported. The numbers shown in table 7.2 should be compared very carefully, since they have been measured with different problems (number of iterations, size, etc.), different versions of the algorithm (floating-point numbers vs. integers, on-line vs. epoch updating, momentum, etc.), and different degrees of optimization (high-level language, assembly, etc.). For these reasons, the values presented in table 7.2 are the highest ones, found in the corresponding reference, rounded to two significant digits.

It appears from table 7.2 that the performance of the MANTRA I machine on ANN algorithms is of the same order of magnitude that the one of super-computers. This observation deserves further discussion. A first difference is that the algorithms programmed on the Cray Y-MP C90 and NEC SX-3 use floating-point numbers, while the MANTRA I version is an integer implementation. However, the results found in the literature and those presented in section 7.4.3 show that this difference has little impact on the convergence. A second remark is that the performance reported by U. Müller [MUL93] for these two super-computers[3] may be sub-optimal.

It should be stressed that the goal of the MANTRA I machine is less to provide a very fast accelerator than to test architectural concepts. The reader should realize that the technology (available to a university) used for the GENES IV circuit (i.e., 1 μm CMOS standard cells, $6.3 \times 6.1\,\text{mm}^2$,

[3] Although U. Müller measured himself the performance of all other cited machines with fairly well optimized benchmarks, the values for these two computers come from unpublished references.

10 MHz) is not comparable with that of a high-end DSP (e.g., TMS320C30 or MC96002) or that of dedicated chips like the MA16 (see the description of the SYNAPSE machine in section 2.2.4) or the CNAPS-1064 (see section 2.2.2).

Finally, it can be mentioned that the performance of the MANTRA I simulator for the delta rule[4] running on a 16 MHz TMS320C40, is approximately 108 KCUPS on all benchmarks presented in section 7.3. The floating-point delta rule used for the simulations of section 7.4 (a decently optimized C routine running on a 20 MHz TMS320C40) sustains a performance of 500 to 600 KCUPS, depending on the parameters. An optimized assembly implementation would probably be two or three times faster.

References

[ADA94] Adaptive Solutions, Beaverton, OR (USA). *CNAPS®Server II*, 1994. Data sheet.

[HEN90] J. L. Hennessy and D. A. Patterson. *Computer Architecture: A Quantitative Approach.* Morgan Kaufmann Publishers, San Mateo, CA (USA), 1990.

[JAI91] R. Jain. *The Art of Computer Systems Performance Analysis: Techniques for Experimental Design, Measurement, Simulation, and Modeling.* Wiley Professional Computing. John Wiley & Sons, New York, NY (USA), 1991.

[MOR92] N. Morgan, J. Beck, P. Kohn, J. Bilmes, E. Allman, and J. Beer. *The Ring Array Processor: A Multiprocessing Peripheral for Connectionist Applications. Journal of Parallel and Distributed Computing*, 14(3):248–259, March 1992. Special issue on neural computing on massively parallel processing.

[MUH89] H. Mühlenbein and K. Wolf. *Neural Network Simulation on Parallel Computers.* In D. J. Evans, G. R. Joubert, and F. J. Peters (eds.), *Proceedings of the International Conference on Parallel Computing*, vol. 2 of *Advances in Parallel Computing*, pp. 365–374, Leiden (NL), August & September 1989. North-Holland.

[MUL93] U. A. Müller. *Simulations of Neural Networks on Parallel Computers*, vol. 23 of *Series in Microelectronics.* Hartung-Gorre, Konstanz (D), 1993. Ph.D. thesis no. 10188, ETH-Zurich.

[POM88] D. A. Pomerleau, G. L. Gusciora, D. S. Touretzky, and H. T. Kung. *Neural Network Simulation at Warp Speed: How We Got 17 Million Connections per Second.* In *Proceedings of the IEEE International Conference on Neural Networks*, vol. II, pp. 143–150, San Diego, CA (USA), July 1988. IEEE.

[RAM92] U. Ramacher. *SYNAPSE—A Neurocomputer That Synthesizes Neural Algorithms on a Parallel Systolic Engine. Journal of Parallel and Distributed Computing*, 14(3):306–318, March 1992. Special issue on neural computing on massively parallel processing.

[WIT90] M. Witbrock and M. Zagha. *An Implementation of Backpropagation Learning on GF11, a Large SIMD Parallel Computer. Parallel Computing*, 14(3):329–346, July 1990.

[4] This software package simulates the MANTRA I machine at the library-function level. The precision of the machine is exactly emulated, yielding a bit-compatible result.

[ZHA90] X. Zhang, M. Mckenna, J. P. Mesirov, and D. L. Waltz. *An Efficient Implementation of the Back-Propagation Algorithm on the Connection Machine CM-2.* In D. S. Touretzky (ed.), *Advances in Neural Information Processing Systems 2*, pp. 801–809. Morgan Kaufmann Publishers, San Mateo, CA (USA), 1990. Proceedings of NIPS-89.

Si enim fallor, sum. Nam qui non est, utique nec falli potest; ac per hoc sum, si fallor. Quia ergo sum si fallor, quo modo esse me fallor, quando certum est me esse, si fallor?[1]

Saint Augustine, *De civitate Dei*, XI, XXVI

8

Future Systolic Architectures for Neural Computation

In this chapter, the use of systolic arrays for ANN computation is discussed in the light of the experience acquired with the GENES IV array and the MANTRA I machine. The discussion has been split in three parts, from most specific to most general. Section 8.1 presents the limitations of the machine. They have already been mentioned in section 6.1 with possible work-around solutions. This section describes how they could be avoided in a new version of the hardware. Section 8.2 proposes several enhancements for the machine. These improvements would greatly simplify the programmer's task, and would be welcome in a re-engineering of the machine. Finally section 8.3 discusses the future of systolic arrays for ANNs in a more general way. This section may be considered as a reflection start-point for the design of neural accelerators.

8.1 Architectural limitations of the MANTRA I machine

This section describes the two problems that hinder a straightforward implementation of the back-propagation rule and the Kohonen model. For each, a simple modification implementable in a new version of the control board (see section 5.4) is presented first. Other solutions, adequate for a re-engineering of the machine are then discussed.

8.1.1 Error back-propagation

In section 6.1.2, the mapping of the back-propagation rule on the machine has pointed out some problems. The first version, shown in algorithm 6.3, is implementable without modifications but presents a few drawbacks, the most important being that it is not extensible to virtual matrices. The second version, shown in algorithm 6.4, requires a small hardware modification, that is, the

[1] *Well, if I am mistaken, I exist. For a man who does not exist can surely not be mistaken either, and if I am mistaken, therefore I exist. So, since I am if I am mistaken, how can I be mistaken in believing that I am when it is certain that if I am mistaken I am.* (Translation by David S. Wiesen.)

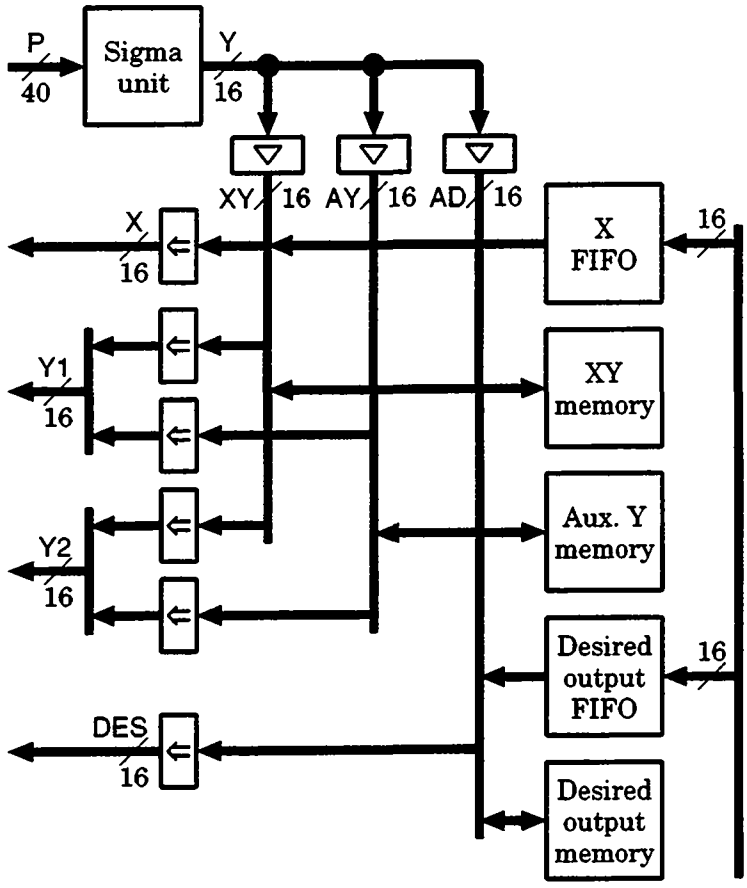


Figure 8.1: *The data units with an additional data path from the* $Y_{15..0}$ *bus to the desired output memory.*

possibility of imposing a null value on the $Y2_{15..0}$ bus. A potential bottleneck for this algorithm is the control microprocessor, which must transfer the error vectors $\vec{\varepsilon}^{[k]}$ from the Y FIFO to the desired output FIFO.

Many solutions can be found to remove this flaw in a re-design of the control board. The first possible modification is to implement this transfer in hardware. This means that a data path should be added to make it possible to write the result $Y_{15..0}$ of a computation into the desired output memory, as shown in figure 8.1. This scheme still requires the ability to impose a zero value on the $Y2_{15..0}$ bus.

While the implementation of the back-propagation rule is possible with this solution, it does not increase the generality of the machine, and a slightly different algorithm could fail to be implementable. So, if some new features are added to the GENES array, as discussed in section 8.3, a more general solution should be found. Since there are four different inputs (i.e., $X_{15..0}$, $Y1_{15..0}$, $Y2_{15..0}$, and $DES_{15..0}$), four different data should be available each cycle. The most straightforward solution is shown in figure 8.2. It involves a dedicated memory bank for each of the four inputs, but only one *input FIFO*. As far as storage is concerned, this implementation is far from being optimal, since it requires duplication of data when the same value should be injected in two or more inputs (possibly at different times).

To avoid this, the memory banks could be considered as shared among the four busses, as shown

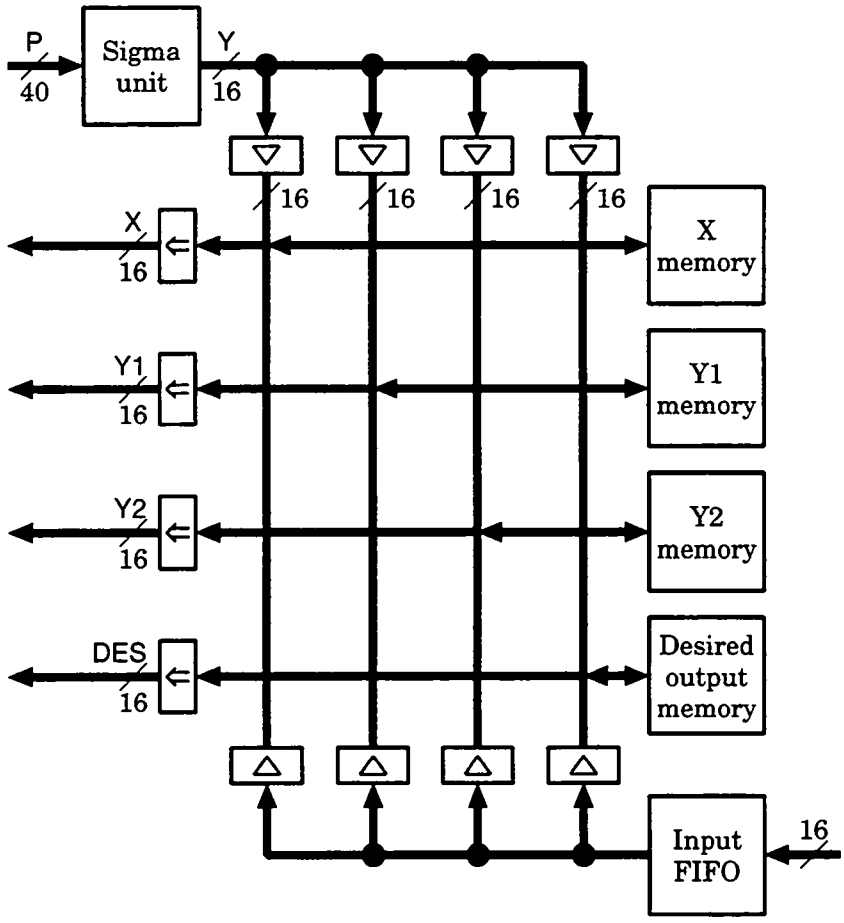


Figure 8.2: *The data units with a dedicated memory bank per input.*

in figure 8.3. It should be noticed that, to ease the representation, multiplexers (with registered output) have been drawn instead of registers with tri-state outputs connected to a common bus. Both approaches being functionally equivalent, the designer should choose one according to criteria such as available devices, achievable speed, etc.

Finally, many variants of this solution are also possible. For instance, the number of memory banks could be reduced to three. This would prevent some algorithms from being run, but would be general enough to run any model presented in chapter 1. Alternatively, the number of memory banks could be increased. It can easily be seen that four banks are enough to implement any input scheme, but sometimes data duplication may not be avoided. A larger number of banks would result in a better utilization of the available storage. A similar consideration can be made for the input FIFO. With a single FIFO, only one value can be injected from the microprocessor per clock cycle. Duplication of the FIFO (and buffers) would increase the generality of this solution. More than four FIFOs would provide no further gain.

8.1.2 Minimum/maximum operations

As mentioned in section 6.1.3, there is a bug in the machine that limits the use of the minimum and maximum operations to numbers that are respectively all negative or all positive, unless the

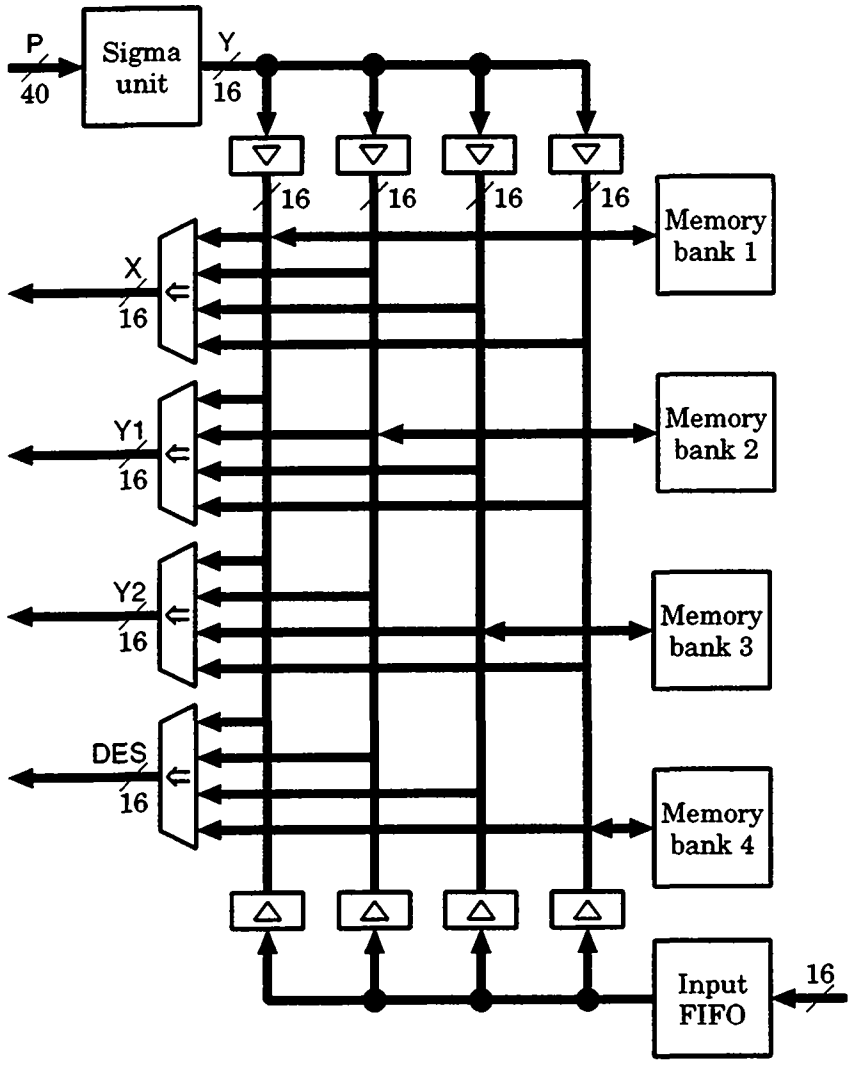


Figure 8.3: *The data units with shared memory banks.*

vector length is a multiple of the number of PEs per array's edge. When only a part of the array is used, extra rows and columns should be deactivated. For all operations but these two, this can be done by loading initial null weights and injecting zeroes in extra rows and columns. For the minimum and maximum operations, the largest representable number N_{max} or the smallest one N_{min}, respectively, should be injected instead. In the MANTRA I machine, two fields of the input/output counter register (see section 5.2.2) are used to define how many "useful" values (and thus how many zeroes) should be sent to the rows and columns of the array.

A first possible way to correct this problem would be to add a new field in the configuration register (see section 5.2.2) to define whether the extra values sent to the array are zeroes or the largest/smallest numbers. This would require only a few additional buffers and could easily be introduced in a new version of the control board. The main advantage of this solution is that it is compatible with the GENES IV circuit.

Should the modification be associated with a new version of the GENES circuit, a more elegant solution would be to implement two vectors of enable bits $\vec{\mathbf{E}}^h$ and $\vec{\mathbf{E}}^v$ propagating systolically

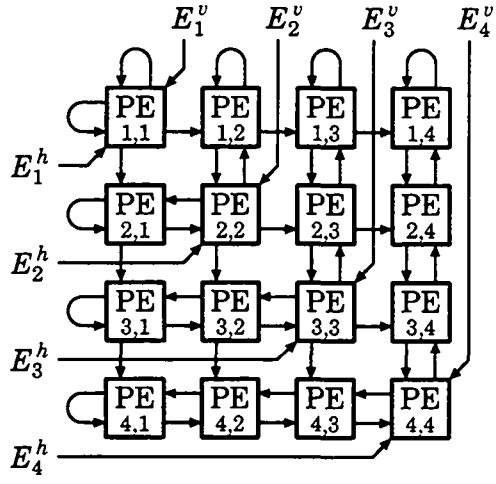


Figure 8.4: *Systolic flow of enable bits (4 × 4 PEs).*

along rows and columns, much like the selectors of ISAs (see section 2.1.3). These bits should obviously be entered on the diagonal and propagate in the counter-flow direction first, to be correctly "diagonalized," as shown in figure 8.4. This scheme would remove the burden of generating special values with external logic. A more important advantage is that this is a general mechanism compatible with any possible new operation, while the special-value method is only valid for some particular operations and the set of special values could grow rather large.

8.2 Enhancements for the MANTRA I machine

This section proposes a few enhancements that could be integrated in a re-engineering of the MANTRA I machine. These issues address hardware units that have been purposely kept simple on the first prototype. An improved version of these units would result in an architecture that is more general and easier to use.

8.2.1 The Kohonen learning rule

As mentioned in section 6.1.3, the MANTRA I machine implements the Kohonen model with semi epoch updating (see section 1.10.1). Although less robust, there could be applications requiring the true epoch updating version. Providing the possibility to implement this algorithm goes towards a generalization of the machine.

This problem can only be corrected by a re-design of the GENES circuit. The simplest way to implement the real Kohonen algorithm with epoch updating, is to add a new register, in which to accumulate the weight updates $\Delta\mathbf{w}$, and a new instruction to add this value to one of the weight registers. This technique is also compatible with a generalization of the PEs, involving a general-purpose register file (see section 8.3.3).

8.2.2 The sigma unit

The double look-up table scheme of the sigma unit, presented in section 5.1.2, gives a rather good precision for a design easily implementable with off-the-shelf components. However, it is not very flexible and cumbersome to initialize. A better solution would be to convert the 40-bit potentials p_i into a floating-point value first and then feed it to a look-up table to compute the integer output y_i. A hardware scheme merging the floating-point and serial-to-parallel conversions is presented below, after a discussion of the floating-point format.

There are many different floating-point formats. All involve a sign bit, a mantissa and an exponent. Only formats, whose exponent is interpreted as a power of two, are considered here. Sometimes the sign bit and the mantissa are concatenated in a two's complement value, while sometimes they form the two entities of a sign and magnitude format. Similarly, the exponent can be a two's complement value or an unsigned number associated with a bias value. From the information point of view (i.e., the relation between the size of each field and the amount of representable values), these techniques are almost equivalent.

For the sigma unit, no fractional numbers need to be represented. Hence, the binary point can be placed on the right of the mantissa's LSB. Therefore, the exponent is always positive and can be represented by an unsigned integer without bias. For a look-up table with N_A address lines, the following relation should be met:

$$N_A = N_{\text{mant}} + \lceil \log_2 (N_{PS}^{\text{eval}} - N_{\text{mant}}) \rceil + 1 \tag{8.1}$$

where N_{mant} is the size of the mantissa, and N_{PS}^{eval} is the effective size of the PS register for evaluation operations. For RAMs with 18 address lines (the same as in the current design), the mantissa would be of 12 bits and the exponent of 5 bits. It can be noticed that some exponents are unused (in this case 5 out of 32). Advantage could be taken from this fact to treat overflowed potentials in a special way. For instance, the largest exponent could be reserved for these values.

With this scheme, it can easily be seen that the MSB of the mantissa is redundant for all exponents different from zero. In two's complement format this bit is always equal to the inverted sign bit, while it is always equal to one in sign and magnitude format. Such values are called normalized. Most floating-point formats accept only normalized mantissas to spare the explicit coding of its MSB. Normalizing the mantissas in the sigma unit would significantly increase the range of possible values of the exponent, since negative exponents would be required. The gain of a bit in the mantissa could then be lost by an additional bit required for the exponent. An alternate scheme is to place the binary point on the left of the mantissa's LSB. The range of values for the exponent becomes then $[1, N_{PS}^{\text{eval}} - N_{\text{mant}}]$, instead of $[0, N_{PS}^{\text{eval}} - N_{\text{mant}} - 1]$. A zero exponent may then have the special meaning that the mantissa is de-normalized and should be taken as an integer. The relation (8.1) between the different field sizes must then be replaced by:

$$N_A = N_{\text{mant}} + \lceil \log_2 (N_{PS}^{\text{eval}} - N_{\text{mant}} + 1) \rceil \tag{8.2}$$

For the same 18-address-bit RAMs, the mantissa is now of 13 bits (with only 12 bits that are actually represented), while the exponent remains of 5 bits.

Since the floating-point conversion mainly involves counting how many MSBs are identical, the bit-serial LSB-first format of the potential is ill-suited. A first possible scheme is to buffer the potential in a shift register (serial-to-parallel converter), and perform the floating-point conversion

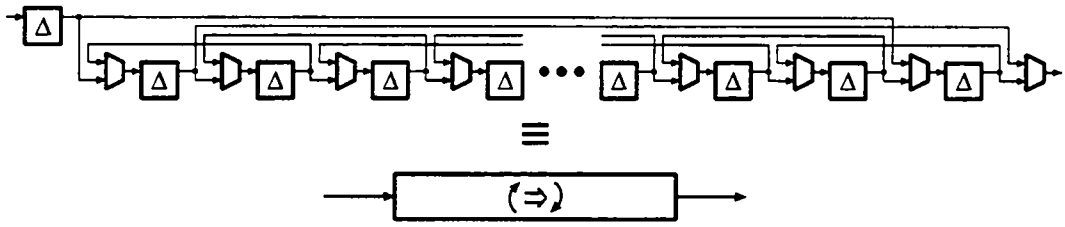


Figure 8.5: *Bi-directional shift register and its symbol.*

in parallel during the last clock cycle of each macro-cycle. This requires a parallel unit to compute the exponent and a barrel shifter.

Another possibility, less expensive in hardware, is to convert the bit-serial potential into MSB-first format. This can be done, by adding a pipeline stage in the sigma unit, with the bi-directional shift register shown in figure 8.5. Changing the shifting direction each macro-cycle transforms the LSB-first input stream into an MSB-first output stream. The special position of the overflow flip-flop (MSB) ensures that the overflow bit is always read last. Hence, the floating-point converter can be built as shown in figure 8.6. During the first cycle of each macro-cycle, the shift register is initialized with the sign bit, and the down counter is set to $N_{\mathrm{PS}}^{\mathrm{eval}} - N_{\mathrm{mant}}$. Shifting is then enabled as long as the first two MSBs are equal. The signal CNTEN is active from the $(N_{\mathrm{mant}} + 2)^{\mathrm{th}}$ to the $(N_{\mathrm{PS}}^{\mathrm{eval}})^{\mathrm{th}}$ cycle. This scheme does not specify how overflows are handled. If the overflow bit is to be ignored, the circuit shown in figure 8.6 can be directly used. The transfer of the shift register and the counter to the parallel registers is then performed during the last clock cycle. A better scheme would be, for instance, to load the exponent register with the largest representable value $2^{N_{\mathrm{A}}-N_{\mathrm{mant}}} - 1$ in case of overflow.

The schematics of figure 8.6 require $N_{\mathrm{PS}}^{\mathrm{eval}} + 2N_{\mathrm{A}} + 1$ flip-flops (an additional flip-flop is necessary if overflows should be treated). This is a large number for an implementation with EPLDs or PGAs, since no more than four or five converters would fit in the largest available ones (the combinatorial logic should not be a problem). However, if the GENES circuit is also re-designed, the U and L registers (see section 4.2.2) could be implemented as bi-directional shift registers. Global signals could then control when the shifting direction should be toggled. This modification

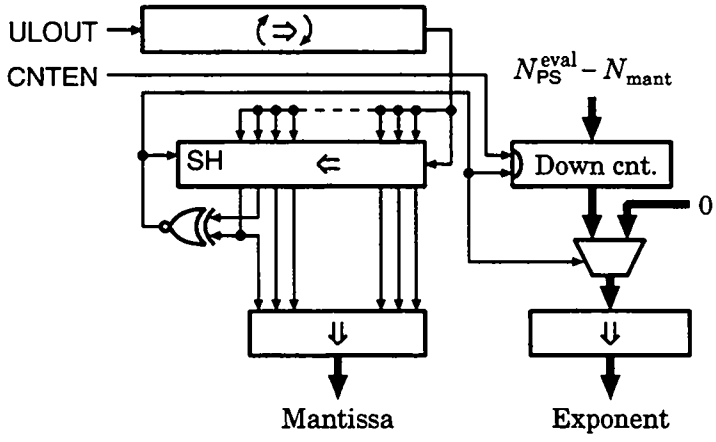


Figure 8.6: *Floating-point converter.*

would also require to place the signals UIN, UOUT, LIN, and LOUT after the U and L registers respectively (see figure 4.4). This would reduce the floating-point converter to $2N_A + 1$ flip-flops. Hence, eight to ten units could be implemented per EPLD or PGA. Concerning the GENES circuit, this modification would not significantly increase the amount of hardware, since standard-cell libraries offer well-optimized cells combining a flip-flop and a multiplexer. A pipeline stage would also be saved.

Many variations and/or improvements can be considered for the floating-point converter. For instance, the size of the mantissa N_{mant} could be configurable. Therefore, for a given RAM size, either a single and precise look-up table or multiple tables with a coarser precision could be implemented.

8.2.3 The SIMD controller and the instruction FIFO

The main drawback of the MANTRA I machine is its complex programmer's model. As mentioned in section 5.2.2, the machine follows the VLIW philosophy, and resource parallelism is managed by software. What makes programming complex is that the fields of the VLIW control different stages of the pipeline during the same macro-cycle. In the $2N$ stages formed by the GENES IV array, the relevant part of the instruction is propagated from PE to PE, following the corresponding data (see section 4.1.3). However, this is not true for the preceding and following stages.

A great simplification of the programmer's model would be to have the instructions follow the data along the whole pipeline. This could be achieved with a reasonable amount of hardware. An analysis of the instruction and auxiliary registers shows that the stages that are directly accessed are the first, second, third, $(2N+2)^{th}$, and $(2N+3)^{th}$. Delaying the activation of some fields by one or two macro-cycle presents no problem, and can be implemented by a few flip-flops in the controller. Delaying them by $2N+1$ and $2N+2$ cycles is less obvious, because the number of processors N^2 should be configurable. A possible implementation is to use an elastic FIFO. A special location should be defined to let the microprocessor configure the FIFO depth. If $2N+1$ write accesses are executed at boot time, and then a read access is always balanced by a write access, the FIFO depth will remain constant and hence fulfill the requirement.

8.3 Future trends

ANN systems based on systolic architectures will probably evolve in three main directions: performance, generality, and ease of use. While the first two aspects concern chiefly the hardware, the third one should be solved by a combination of hardware and software.

8.3.1 Floating-point numbers

As mentioned in section 4.2.5, papers in the literature show that 16-bit integers are enough for most algorithms to converge (the precision is usually increased during the learning phase as in the GENES IV array). This fact is confirmed by the experimental results presented in section 7.4.3. However, section 6.2 shows that achieving an optimally scaled conversion of floating-point values to integers may be a tough process. With the MANTRA I machine, this is a small problem, because this system can run only a small number of algorithms, and the constraints imposed by integer arithmetic could be analyzed once for all. However, as ANN computers become more general and

each user can run his own model, this task must be repeated for each implementation. Although integers are sufficient as far as performance is concerned, floating-point computation may become a crucial criterion if generality and ease of use are taken into account.

Since most problems can be solved with 16-bit integers, it can be assumed that the precision of conventional 32-bit floating-point formats is not required. However, some compatibility with the IEEE 754-1985 standard [IEE85] may be a great advantage. A possible solution is to adopt a similar format with a reduced mantissa, so that format conversions become simple truncations or extensions with zeroes.

8.3.2 Serial versus parallel computation and communication

As mentioned in section 2.1.2, equal computation and communication rates is one of the key characteristics of systolic architectures. Fully parallel communication with 16-bit or larger data would probably be not be feasible for medium-cost fine-grain or massively parallel systems. On the other hand, bit-serial computation is an important performance penalty. A compromise is likely to be adopted. For instance, 4 or 8 bits could be transmitted in parallel.

If floating-point numbers are used, additions and subtractions may not be started before the exponents, the sign bits, and—in the worst case—the full mantissas are known. Therefore, the communication should either be fully parallel, or both input and output data should be buffered, doubling the pipeline depth.

8.3.3 General systolic processing element

To keep the hardware of the GENES IV circuit simple, only the minimum number of operations required to implement the ANN models of chapter 1 have been integrated. In a truly general neural computer, the PEs should be much more general, to let users run modified or optimized versions of a model or even new algorithms. For instance, the MANTRA I machine implements only the Kohonen model with semi epoch updating (see section 8.2.1). Similarly, the momentum for the delta or back-propagation rule can only be approximated in a rather inefficient way (see section 6.1.1).

PEs can be made more general by implementing a register file and a larger instruction set. The idea of a register file is a generalization of the double weight register implemented in the GENES IV PEs (see section 4.2.2). These registers could be used to store both weights and intermediate results. Instructions should code which registers are used as operands and which one is used for input/output through the dedicated path WGTIN–WGTOUT.

The fractional part of the weight registers used during learning operations (see section 4.2.5) could be implemented by cascading two registers. This added flexibility would let the programmer make use of the available storage capacity in an optimal way. This feature would probably be of limited use if floating-point numbers are used. Four floating-point registers or eight integer registers would probably be enough for most applications.

A larger instruction set is necessary to offer more flexibility to the programmer. In the GENES IV circuit, the arithmetic unit is composed of an adder, a subtracter and a multiplier. A possible orthogonalization of the instruction set would be to implement the operation $A + B \cdot (C - D)$, as shown in figure 8.7. Any of the operands could come from one of the input ports NIN and WIN or from the register file. A dummy *neutral register*, which would take the value of zero if placed in position A, C, or D and the value of one if placed in position B, could

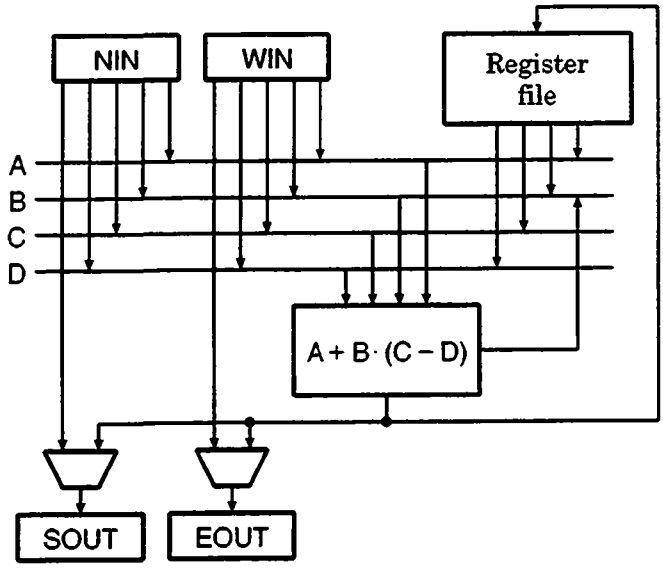


Figure 8.7: *Possible general systolic processing element (PE).*

be implemented. It should also be possible to re-inject C – D on the B bus to implement the square operation. The result could be either stored in the register file or sent to one of the output ports SOUT and EOUT. Figure 8.7 shows neither the dedicated path WGTIN–WGTOUT, nor the counter-flow paths SIN–NOUT and EIN–WOUT (if implemented, see section 8.3.4). The input and output ports NIN, WIN, SOUT, and EOUT may be either combinatorial or registered, depending on the implementation (see section 8.3.2).

For this architecture, no operation code is required in the instruction word, operations being implicitly coded by their operands. The transposition mechanism (see section 4.2.3) of GENES IV PEs is also generalized by the choice of the operands and destination. With n_{reg} registers (excluding the neutral register), $\lceil \log_2 (n_{\text{reg}} + 3) \rceil$ bits are required for the operands A, C, and D, $\lceil \log_2 (n_{\text{reg}} + 4) \rceil$ bits for the operand B, and $\lceil \log_2 (n_{\text{reg}} + 2) \rceil$ bits for the result.[2] For example, with $n_{\text{reg}} = 4$ registers, instructions would be 15 bits wide. Some combinations are useless, and the coding could be optimized. However, this would not comply to the VLIW approach, and would make the control logic more complex. Some information is also required to control the saturation mechanism, used to implement the minimum and maximum operations.

With such a PE, the Kohonen model with epoch updating and the delta or back-propagation rules with momentum can be efficiently implemented. However, it is questionable if this instruction set is complete enough. In particular, many algorithms involve multiplications by values smaller than one or divisions by constants. With integers, a shifter could be useful for these operations. However, it is not necessary, since it can be emulated by keeping only the most-significant half of the result, possibly preceded by a multiplication by a power of 2 (equivalent to a left-shift operation). This raises also the question, whether to choose the most-significant or least-significant half of the result when registers are not cascaded. The most general approach is to define different fields in the instruction to choose the destination of both halves. In this case, writing into the neutral register would discard the result. Many other compromises could also be found, which would require fewer bits for the instruction word.

[2] Some additional information is required if (integer) registers should be cascadable.

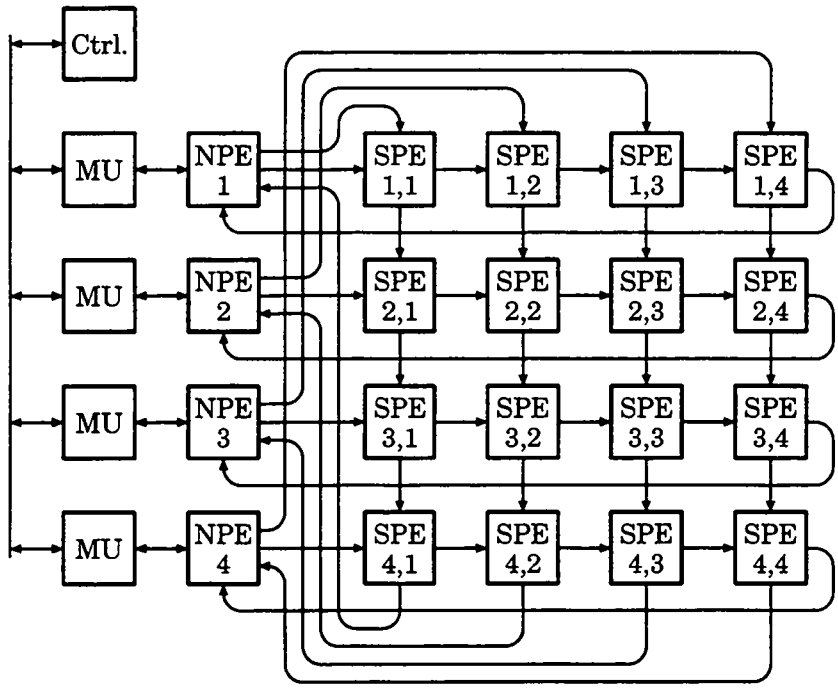


Figure 8.8: *General linear/bi-dimensional systolic architecture.*

To summarize, the choice of the PE architecture—mainly the instruction set and the size of the register file—should reflect a trade-off between generality, efficiency, cost in silicon area, and implementation (integer vs. floating-point, serial vs. parallel). Arithmetic precision remains also an open question.

8.3.4 General systolic architecture

In the MANTRA I machine, the bi-dimensional GENES IV array is used for $O(m \cdot n)$ and $O(m^2)$ operations (see section 4.1), where m is the number of neurons and n is the number of inputs. On the other hand, dedicated units are used for $O(m)$ operations (see section 5.1). These units are either serial or in the form of an extra column of the array.

Trying to fit all operations on a linear array is possible and has been extensively done in the literature (see section 2.2). However, the parallelism grain would be significantly coarsened. On the other hand, trying to fit the whole computation on a bi-dimensional system would lead to a poor utilization rate. For maximal performance, the natural decomposition of the MANTRA I machine should be kept.

In a fully parallel system, serial units should be avoided. The architecture can be divided into a bi-dimensional part that scales with the number of synapses $m \cdot n$ and a linear part that scales with the number of neurons m. The PEs of the bi-dimensional array are hereafter referred to as *synaptic processing elements (SPEs)*. In a similar way, linear units are grouped into entities called *neural processing elements (NPEs)*. Figure 8.8 shows the general interconnection topology between the SPE array and the NPEs. This figure shows neither counter-flow paths nor input/output ports on the diagonal. It is debatable whether these features of the GENES IV array should be kept, since the "diagonalization" can also be achieved by an appropriate pattern of memory accesses.

Since the NPEs scale with the number of neurons and the SPEs with the number of synapses, it could be more advantageous to increase the complexity of the NPEs rather than waste the silicon area of the SPEs and increase the pipeline depth. This design choice—conditioned by criteria like implementation technology (off-the-shelf vs. VLSI components) and parallelism grain—does not affect the following discussion.

Each NPE should be attached to a *memory unit (MU)* whose task is to input and output data and to store intermediate results and auxiliary data such as look-up tables. Its implementation offers another degree of freedom to the architecture. It is likely to be composed of a mixture of local RAMs, shared RAMs, dual-port RAMs, and/or FIFOs (see sections 5.1.4 and 8.1.1). Since neural computers are meant to implements very large ANNs, it is important that the NPEs fetch data in dynamic RAMs or video RAMs (if dual-port devices are required), as in the SYNAPSE-1 machine (see section 2.2.4). The use of static RAMs for weights and data in the MANTRA I machine, severely restricts the size of applications, or affect the computation speed if paging to and from the microprocessor's dynamic RAM through the FIFOs is required.

The analysis of a typical algorithm such as the delta rule (see section 1.7) shows that, during the instruction execution time of the SPEs, each NPE should be able to:

1. store the result of an evaluation phase (either the potential p_i^* or the output y_i),
2. compute an activation function $\sigma(p_i^*)$,
3. compute an activation function's derivative $\sigma'(p_i^*)$,
4. compute an error signal δ_i, and
5. fetch an input x_j^*.

These operations will likely be performed on different prototypes. Therefore, very fast NPEs are not required, since pipelining can be used. Some stages may be composed of parallel units, so the pipeline depth may be smaller than five. A depth of three, as on the MANTRA I machine, is probably close to optimal. The analysis of the back-propagation rule (see section 1.8) shows similar requirements, since the computation of the last layer is identical and other phases of the algorithm involve either less or the same amount of computation.

This analysis provides a lower bound for the memory bandwidth: operation 1 requires a data write access, operations 2 and 3 a data read access[3] and two look-up table accesses, and operations 4 and 5 an additional data read access each. Altogether, one data write access, three data read accesses, and two look-up table accesses are necessary every SPE instruction cycle. As stated above, data should be stored directly in dynamic RAMs. However, the cycle time of typical dynamic RAMs is much too long for random accesses. Memory interleaving could be used, but may not be necessary if advantage is taken from the fast serial access capabilities offered by dynamic RAMs (i.e., either the page or static column modes of dynamic RAMs, or the serial ports of video RAMs). This requires each NPE to be attached to four interchangeable memory banks.

If the activation function and its derivative are implemented by look-up tables, as assumed above, some more storage would be required. Look-up table accesses being random by nature, these tables should be stored in static RAM. These functions could also be computed by a different technique. In this case, the static memory would not be required, but a dedicated unit should

[3] Depending whether the potential p_i or the output y_i is stored in memory, this data is used by both operations or only by operation 3.

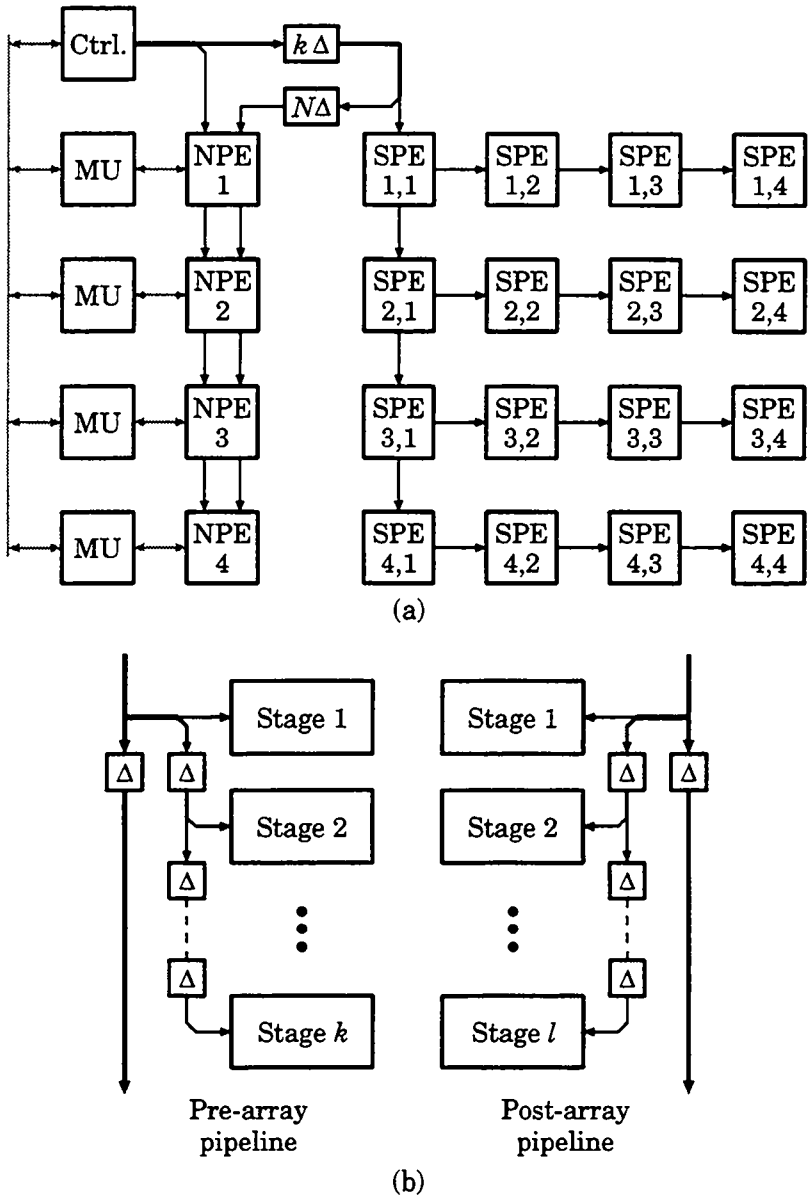


Figure 8.9: *Instruction stream in a general ANN systolic architecture.* (a) *Linear/bi-dimensional systolic array,* (b) *Neural processing element (NPE).*

be able to sustain a computation rate of one evaluation per cycle. This would probably call for pipelined units, and would also restrict the type of implementable functions. The increase of pipeline depth would probably be negligible compared to the $2N$ stages formed by the SPE array.

The great regularity of ANN models and the very small data dependence of their control flow makes the SIMD approach (see section 2.1.1) very well suited. MIMD architecture would only bring very little advantage. If hardware simplicity is desirable, the VLIW philosophy may be a good solution. Instructions should follow the data trough the pipeline (see section 8.2.3). Figure 8.9(a) shows the instruction stream at the machine level, while figure 8.9(b) details the control of the pipelined NPEs. These units are split in two halves: a *pre-array pipeline* that processes the inputs of the SPE array and a *post-array pipeline* that processes its outputs. The fields of the

VLIW controlling the latter units are delayed by $N + k$ cycles, where k is the pre-array pipeline depth. Similarly, the part of the VLIW destined to the SPEs is delayed by k cycles.

8.3.5 Sparse matrices

Systolic arrays are usually good at processing dense matrices, while they perform poorly on sparse matrices. The machine should be designed carefully enough so that sparse matrices are not used with bi-dimensional systolic arrays. In the MANTRA I machine, this poor behavior is mainly observed for the search of the winner and the multiplication by the sparse neighborhood matrix λ. In the linear/bi-dimensional architecture proposed in figure 8.8, these operations could advantageously be computed by the NPEs. This would require an interconnection network between these elements. A ring would probably best fit their topology.

References

[IEE85] ANSI, IEEE, New York, NY (USA). *IEEE Standard for Binary Floating-Point Arithmetic*, 1985. Std 754-1985.

Vanité du mot expérience. L'expérience n'est pas expérimentale. On ne la provoque pas. On la subit. [...]
Tout pratique : au sortir de l'expérience, on n'est pas savant, on est expert. Mais en quoi ?[1]

Albert Camus, *Carnets,* 1935 - 1937

Conclusion

The aim of this thesis was to study the design of dedicated computers for artificial neural networks. The need for ANN accelerators has clearly been identified. At the time the GENES project started, there were no such machines. Nowadays, although several research and commercial systems have been running for some time, the computing needs in the domain keep increasing. Therefore, research in this field is becoming increasingly important.

ANN models have first been analyzed to identify the required type of computation. The common formalism adopted to describe ANNs has proved very helpful. Other types of ANN computers have been studied. In particular, the work of F. Blayo and C. Lehmann, on the GENES bi-dimensional systolic arrays, has been analyzed in depth. Compared to other machines, these systems have the particularity of targeting a much finer grain of parallelism. However, they are of a very restricted use, the first version being dedicated to the sole Hopfield network and the following to a few algorithms only. Moreover, besides a very attractive general architecture, they hide specific flaws that make their efficient use rather difficult.

This work has been extended with the GENES IV chip, co-designed by P. Ienne and the author. The key directions that led this development were generality of use and efficiency. Concerning the latter, the instruction set of the previous versions has been augmented to implement the delta rule (as well as the Perceptron and Adaline rules), the back-propagation rule, the Hopfield model, and the Kohonen model. Such a system may be called a multi-model ANN computer. A transpose mode (exchange of the row and column's roles) has been implemented to make efficient processing of transpose matrices possible.

Efficiency has been addressed by the implementation of a matrix-swapping mechanism that makes it possible to exchange sub-matrices in background, while performing some other useful computation. In this way, a user can efficiently implement networks whose weight matrices are larger than the systolic array. Operations have been designed in such a way that cascading two

[1] *Futility of the term experience. Experience is not experimental. You cannot create it. You must undergo it.* [...] *After experience, you are not learned, you are expert. But, what in?*

sub-matrix iterations does not result in any overhead. A systolic flow of instructions has also been implemented, so that each instruction follows the corresponding data.

These two techniques—virtual matrices and systolic instruction flow—ensure an optimal static utilization rate, except for incomplete epochs or incomplete sub-matrices. Optimal use of the hardware is achieved for epoch lengths larger than $2N + 3$ (if the array is not time-shared) or multiple of $2N$ (if virtual matrices are used), where N is the number of PEs per array's edge. Epoch updating is the drawback of using the huge computing power of systolic arrays or any other pipelined system. Hence, it is the user's responsibility to ensure that all epochs are of the required size or not significantly smaller.

The issue of incomplete sub-matrices is of a different nature, because an application's size is usually conditioned by factors different from the implementation. Loss of performance occurs when matrices have an aspect ratio that is poorly mapped on the square array. This problem could have been solved with a configurable interconnection network. This solution has been discarded because the resulting overhead is usually so big that all advantages of systolic arrays are lost. However, this problem vanishes when the size of matrices becomes large enough compared to that of the array. Since the array's size is expected to be small[2] compared to the problem size that scientists would like to run, this should not be a severe drawback.

The same quest for efficiency has led the design of the MANTRA I machine. Dedicated units have been designed to perform operations that scale with the number of neurons and would fit poorly on the bi-dimensional GENES IV array. A complex system of FIFOs and memories ensures that the required input/output bandwidth can be sustained, while minimizing the number of data transfers to be performed by the control module. This unit is based on a powerful microprocessor to maximize the dynamic utilization rate of the GENES IV array.

The results of the performance analysis are very close to what was predicted. It has been shown that the control module becomes a bottleneck only for very small problems, where the number of data transfers is very large compared to the amount of computation.

Unfortunately, due to the lack of software support, the effects of two important factors could not be analyzed in detail: the memory size and the integration of the machine in its environment. The first version of the software supports only networks that fit entirely in the memories of the SIMD module. Performance degradation should be expected if some parts of the application reside in the control microprocessor's memory. This points out one of the weaknesses of the machine, that is, the small amount of memory directly attached to the systolic array. Although swapping is possible, it results in a performance drop. This problem is due to the use of static RAMs, chosen to keep the hardware simple. In a re-engineering of the machine, the possibility of directly interfacing dynamic RAMs to the array should be considered.

The second factor that could have a non-negligible impact on performance is the integration of the machine in a heterogeneous-node network. Since users will likely develop their applications on a workstation and access the MANTRA I machine through an input/output bus (e.g., the SBus) and inter-processor links, the communication overhead should be taken into account. This could not be studied, because the required software is still under development.

The impact of the constraints imposed by the machine (i.e., discrete values of the learning coefficient, epoch updating, and integer data representation) on the convergence of algorithms has also

[2] The MANTRA I machine features up to 1600 PEs. However, with a state-of-the-art technology, a few tens of thousands PEs should be implementable.

been explored. Experiments show a negligible influence once the parameters have been correctly tuned. This shows that these constraints — in particular the integer data representation — do not hinder performance. However, the analysis of the floating-point to integer conversion shows that the use of integers significantly complicates the programming of the machine. As accelerators become more user-friendly, floating-point arithmetic will probably be a crucial point.

Another drawback of the machine is the programming complexity. This is due to the VLIW nature of the machine and to the fact that the different stages of the pipeline must be controlled at the same time, instead of having the instructions following the corresponding data. In future ANN computers, this issue should be tackled by a combination of hardware and software techniques.

Most of the weaknesses of the machine come from the fact that the hardware has been kept as simple as possible. The machine has been purposely designed this way, to keep the implementation effort manageable. This state of things is compatible with the aim of this thesis, which is to study the use of systolic array as neural accelerators. The goal of the machine is more to serve as a test-bench for this research, than to be the fastest neural accelerator that could be designed with today's technology. This also explains why a relatively slow CMOS technology (the only available to a university) and standard cells have been employed.

It is interesting to see that most ANN systems share the same weaknesses than the MANTRA I machine. For instance, the CNAPS computer uses integer arithmetic and has rather small on-chip memories. On the other hand, the SYNAPSE machine is attached to a huge amount of memory, but it uses integer too arithmetic and has a very complex programming model. The main advantage of the MANTRA I machine, compared to these systems, is that it targets a finer parallelism (at the synapse level), and thus exploits most of the intrinsic parallelism of ANN models. The only unexplored parallelism is that contained in the "basic operations." For instance, multiply-accumulate operations of typical synaptic connections could be split in two and evaluated by a pipelined PE. Other features of the machine are a very efficient use of the available resources and a good sustainable utilization rate. The former point is illustrated, for instance, by the multiplier of the PEs, which is used by the matrix-vector product, the squared Euclidean distance, and the two learning rules.

A comparison can be drawn between the state of ANN accelerators today and that of super-computers in the late 60's and early 70's. Both offer a very large computing power (compared to the available technology). However, exploiting this power requires a lot of work from the user, because of the restrictions imposed on algorithms (e.g., batch processing) and of a cumbersome programming model. Writing software for these machines is restricted to a few library designers, who have a deep knowledge of the underlying architecture. The average user should restrict himself to the use of these libraries.

The comparison with super-computers does not hold any more when cost is considered; ANN accelerator offering high performance at low cost (i.e., at workstation cost), while super-computers have always been extremely expensive.

ANN computers should now evolve in the direction of increased user-friendliness and ease of programmability, like super-computers that have become accessible to non-specialists by running operating systems like UNIX and offering vectorizing or parallelizing compilers. This goal should be achieved by a combination of hardware and software. However, while super-computers required almost two decades to reach this state of maturity — thank to the advances in the fields of computer architecture, parallel processing, VLSI/WSI, compiler technology, and software engineering — ANN accelerators are destined to become commercially appealing solutions in a much nearer future.

A

Addendum to ANN Models

This appendix discusses some particular aspects of the ANN models presented in chapter 1. Additional demonstrations are also provided. The presentation of these topics completes the description of ANNs, but their knowledge is not required for the understanding of the core of this thesis.

A.1 Activation functions

This section presents a non-exhaustive list of common activation functions. The simplest activation function is the *step function* or *all-or-none response*. There are four basic types, each of them can be expressed in terms of any of the others:

$$\mathrm{u}_{\_\uparrow}(v) = \begin{cases} 0 & \text{if } v \leq 0 \\ 1 & \text{if } v > 0 \end{cases} = 1 - \mathrm{u}_{\_\uparrow}(-v) = 1 - \mathrm{d}_{\uparrow\_}(v) = \mathrm{d}_{\uparrow\_}(-v) \tag{A.1}$$

$$\mathrm{u}_{\_\uparrow}(v) = \begin{cases} 0 & \text{if } v < 0 \\ 1 & \text{if } v \geq 0 \end{cases} = 1 - \mathrm{u}_{\_\uparrow}(-v) = \mathrm{d}_{\uparrow\_}(-v) = 1 - \mathrm{d}_{\uparrow\_}(v) \tag{A.2}$$

$$\mathrm{d}_{\uparrow\_}(v) = \begin{cases} 1 & \text{if } v \leq 0 \\ 0 & \text{if } v > 0 \end{cases} = 1 - \mathrm{u}_{\_\uparrow}(v) = \mathrm{u}_{\_\uparrow}(-v) = 1 - \mathrm{d}_{\uparrow\_}(-v) \tag{A.3}$$

$$\mathrm{d}_{\uparrow\_}(v) = \begin{cases} 1 & \text{if } v < 0 \\ 0 & \text{if } v \geq 0 \end{cases} = \mathrm{u}_{\_\uparrow}(-v) = 1 - \mathrm{u}_{\_\uparrow}(v) = 1 - \mathrm{d}_{\uparrow\_}(-v) \tag{A.4}$$

These functions may be given a threshold θ and an amplitude β: $\beta \cdot \sigma(v - \theta)$. A further generalization is to specify a left asymptote β_{left} and a right one β_{right}: $\beta_{\text{left}} + (\beta_{\text{right}} - \beta_{\text{left}}) \cdot \mathrm{u}(v - \theta)$. This class of functions is referred to as *hard limiters*. One of the most often used is the sign function ($\beta_{\text{left}} = -1$ and $\beta_{\text{right}} = +1$):

$$\operatorname{sign}(v) = -1 + 2\,\mathrm{u}_{\_\uparrow}(v) = \begin{cases} -1 & \text{if } v < 0 \\ +1 & \text{if } v \geq 0 \end{cases} \tag{A.5}$$

A second category of activation functions consists of *saturated linear functions* defined by the two asymptotes β_{left} and β_{right}, and the two discontinuity points v_{left} and v_{right}:

$$\sigma(v) = \begin{cases} \beta_{\text{left}} & \text{if } v \leq v_{\text{left}} \\ \dfrac{\beta_{\text{right}} - \beta_{\text{left}}}{v_{\text{right}} - v_{\text{left}}} \cdot v + \dfrac{\beta_{\text{left}} \cdot v_{\text{right}} - \beta_{\text{right}} \cdot v_{\text{left}}}{v_{\text{right}} - v_{\text{left}}} & \text{if } v_{\text{left}} < v < v_{\text{right}} \\ \beta_{\text{right}} & \text{if } v \geq v_{\text{right}} \end{cases} \tag{A.6}$$

The so-called *sigmoid* function $(1+e^{-v})^{-1}$ is one of the most often used. It may also be given a threshold θ and an amplitude β or a pair of left and right asymptotes β_{left} and β_{right}:

$$\sigma(v) = \frac{\beta}{1+e^{-\frac{v-\theta}{T}}} \tag{A.7}$$

$$\sigma(v) = \beta_{\text{left}} + \frac{\beta_{\text{right}} - \beta_{\text{left}}}{1+e^{-\frac{v-\theta}{T}}} \tag{A.8}$$

An additional parameter T—sometimes called *temperature* by analogy with physics—make it possible to tune the slope $\frac{\beta}{4T}$ or $\frac{\beta_{\text{right}}-\beta_{\text{left}}}{4T}$ of the curve at the threshold θ. Some authors use the hyperbolic tangent instead of the sigmoid. These functions are equivalent, as shown by:

$$\tanh(v) = -1 + \frac{2}{1+e^{-2v}} \tag{A.9}$$

The hyperbolic tangent may lead to more elegant formulae when the activation function is odd and/or the slope at the origin is 1.

A.2 Addendum to the delta rule

The quadratic error for each neuron (see equation (1.29) of section 1.7) is equal to:

$$\begin{aligned} \xi_i(t) &= \frac{1}{2} \cdot \left(d_i(t) - \sigma\left(\mathbf{w}_i^*(t) \cdot \vec{\mathbf{x}}^*(t)\right)\right)^2 \\ &= \frac{1}{2} \cdot d_i^2(t) + \frac{1}{2} \cdot \sigma^2\left(\mathbf{w}_i^*(t) \cdot \vec{\mathbf{x}}^*(t)\right) - d_i(t) \cdot \sigma\left(\mathbf{w}_i^*(t) \cdot \vec{\mathbf{x}}^*(t)\right) \end{aligned} \tag{A.10}$$

Using the gradient descent, this expression is minimized by the following weight updating rule:

$$\begin{aligned} \mathbf{w}_i^*(t+1) &= \mathbf{w}_i^*(t) - \alpha(t) \cdot \left(\text{grad}_{\mathbf{w}_i^*}\left(\xi_i(t)\right)\right)^{\mathrm{T}} \\ &= \mathbf{w}_i^*(t) - \alpha(t) \cdot \left(\sigma\left(\mathbf{w}_i^*(t) \cdot \vec{\mathbf{x}}^*(t)\right) \cdot \sigma'\left(\mathbf{w}_i^*(t) \cdot \vec{\mathbf{x}}^*(t)\right) \cdot \vec{\mathbf{x}}^*(t)\right. \\ &\quad \left. - d_i(t) \cdot \sigma'\left(\mathbf{w}_i^*(t) \cdot \vec{\mathbf{x}}^*(t)\right) \cdot \vec{\mathbf{x}}^*(t)\right)^{\mathrm{T}} \\ &= \mathbf{w}_i^*(t) + \alpha(t) \cdot \left(d_i(t) - \sigma\left(\mathbf{w}_i^*(t) \cdot \vec{\mathbf{x}}^*(t)\right)\right) \cdot \sigma'\left(\mathbf{w}_i^*(t) \cdot \vec{\mathbf{x}}^*(t)\right) \cdot \vec{\mathbf{x}}^{*\mathrm{T}}(t) \\ &= \mathbf{w}_i^*(t) + \alpha(t) \cdot \left(d_i(t) - y_i(t)\right) \cdot \sigma'\left(p_i^*(t)\right) \cdot \vec{\mathbf{x}}^{*\mathrm{T}}(t) \end{aligned} \tag{A.11}$$

For the whole network, equation (A.11) becomes:

$$\mathbf{w}^*(t+1) = \mathbf{w}^*(t) + \alpha(t) \cdot \left(\left(\vec{\mathbf{d}}(t) - \vec{\mathbf{y}}(t)\right) \circ \sigma'\left(\vec{\mathbf{p}}^*(t)\right)\right) \cdot \vec{\mathbf{x}}^{*\mathrm{T}}(t) \tag{A.12}$$

where the symbol $\circ$ represents the *Hadamard product* or term-to-term matrix multiplication.

A.2.1 The Adaline rule in batch mode

A special expression can be derived for the Adaline model (i.e., $\sigma(v) = v$) in batch mode (i.e., $E(t) = [1,\ S]$). The quadratic error to be minimized is given by equation (1.31):

$$\begin{aligned}
\xi_i(t) &= \frac{1}{2} \cdot \sum_{s=1}^{S} \left(d_i(s) - \mathbf{w}_i^*(t) \cdot \vec{\mathbf{x}}^*(s)\right)^2 \\
&= \frac{1}{2} \cdot \sum_{s=1}^{S} \left(d_i^2(s) + \left(\mathbf{w}_i^*(t) \cdot \vec{\mathbf{x}}^*(s)\right)^2 - 2\, d_i(s) \cdot \mathbf{w}_i^*(t) \cdot \vec{\mathbf{x}}^*(s)\right) \\
&= \frac{1}{2} \cdot \sum_{s=1}^{S} d_i^2(s) + \frac{1}{2} \cdot \sum_{s=1}^{S} \mathbf{w}_i^*(t) \cdot \left(\vec{\mathbf{x}}^*(s) \cdot \vec{\mathbf{x}}^{*\mathrm{T}}(s)\right) \cdot \mathbf{w}_i^{*\mathrm{T}}(t) \\
&\quad - \sum_{s=1}^{S} \mathbf{w}_i^*(t) \cdot \left(d_i(s) \cdot \vec{\mathbf{x}}^*(s)\right) \\
&= \frac{1}{2} \cdot \sum_{s=1}^{S} d_i^2(s) + \frac{1}{2} \cdot \mathbf{w}_i^*(t) \cdot \left(\sum_{s=1}^{S} \vec{\mathbf{x}}^*(s) \cdot \vec{\mathbf{x}}^{*\mathrm{T}}(s)\right) \cdot \mathbf{w}_i^{*\mathrm{T}}(t) \\
&\quad - \mathbf{w}_i^*(t) \cdot \sum_{s=1}^{S} d_i(s) \cdot \vec{\mathbf{x}}^*(s)
\end{aligned} \tag{A.13}$$

Introducing the auxiliary matrices $\mathbf{R}$ and $\mathbf{Q}$, defined as:

$$\mathbf{R} = \sum_{s=1}^{S} \vec{\mathbf{x}}^*(s) \cdot \vec{\mathbf{x}}^{*\mathrm{T}}(s) \tag{A.14}$$

$$\mathbf{Q} = \sum_{s=1}^{S} \vec{\mathbf{d}}(s) \cdot \vec{\mathbf{x}}^{*\mathrm{T}}(s) \tag{A.15}$$

the quadratic error, given by equation (A.13) becomes:

$$\xi_i(t) = \frac{1}{2} \cdot \sum_{s=1}^{S} d_i^2(s) + \frac{1}{2} \cdot \mathbf{w}_i^*(t) \cdot \mathbf{R} \cdot \mathbf{w}_i^{*\mathrm{T}}(t) - \mathbf{w}_i^*(t) \cdot \mathbf{Q}_i^{\mathrm{T}} \tag{A.16}$$

The learning rule can then be derived:

$$\begin{aligned}
\mathbf{w}_i^*(t+1) &= \mathbf{w}_i^*(t) - \alpha(t) \cdot \left(\mathrm{grad}_{\mathbf{w}_i^*}\left(\xi_i(t)\right)\right)^{\mathrm{T}} \\
&= \mathbf{w}_i^*(t) - \alpha(t) \cdot \left(\frac{1}{2} \cdot \mathrm{grad}_{\mathbf{w}_i^*}\left(\mathbf{w}_i^*(t) \cdot \mathbf{R} \cdot \mathbf{w}_i^{*\mathrm{T}}(t)\right) - \mathrm{grad}_{\mathbf{w}_i^*}\left(\mathbf{w}_i^*(t) \cdot \mathbf{Q}_i^{\mathrm{T}}\right)\right)^{\mathrm{T}}
\end{aligned} \tag{A.17}$$

Since:

$$\frac{\partial}{\partial w_{i,l}}\left(\mathbf{w}_i^*(t) \cdot \mathbf{R} \cdot \mathbf{w}_i^{*\mathrm{T}}(t)\right) = \frac{\partial}{\partial w_{i,l}}\left(\sum_{k=1}^{n} w_{i,k}(t) \cdot \sum_{j=1}^{n} R_{k,j}(t) \cdot w_{i,j}(t)\right)$$

$$
\begin{aligned}
&= \sum_{\substack{k=1\\k\neq l}}^{n} R_{k,l}(t)\cdot w_{i,k}(t) + \sum_{\substack{j=1\\j\neq l}}^{n} R_{l,j}(t)\cdot w_{i,j}(t) + 2R_{l,l}(t)\cdot w_{i,l}(t)\\
&= \sum_{k=1}^{n} R_{k,l}(t)\cdot w_{i,k}(t) + \sum_{j=1}^{n} R_{l,j}(t)\cdot w_{i,j}(t)\\
&= \sum_{k=1}^{n} R_{l,k}(t)\cdot w_{i,k}(t) + \sum_{j=1}^{n} R_{l,j}(t)\cdot w_{i,j}(t)\\
&= 2\cdot\sum_{j=1}^{n} R_{l,j}(t)\cdot w_{i,j}(t) \qquad \text{(A.18)}
\end{aligned}
$$

$$
\frac{\partial}{\partial w_{i,l}}\left(\mathbf{w}_i^*(t)\cdot\mathbf{Q}_i^{\mathrm{T}}\right) = \frac{\partial}{\partial w_{i,l}}\left(\sum_{j=1}^{n} w_{i,j}(t)\cdot Q_{i,j}\right) = Q_{i,l} \qquad \text{(A.19)}
$$

the learning rule, given by equation A.17, becomes:

$$
\begin{aligned}
\mathbf{w}_i^*(t+1) &= \mathbf{w}_i^*(t) - \alpha(t)\cdot\left(\mathbf{R}\cdot\mathbf{w}_i^{*\mathrm{T}}(t) - \mathbf{Q}_i^{\mathrm{T}}\right)^{\mathrm{T}}\\
&= \mathbf{w}_i^*(t) + \alpha(t)\cdot\left(\mathbf{Q}_i - \mathbf{w}_i^*(t)\cdot\mathbf{R}^{\mathrm{T}}\right)\\
&= \mathbf{w}_i^*(t) + \alpha(t)\cdot\left(\mathbf{Q}_i - \mathbf{w}_i^*(t)\cdot\mathbf{R}\right) \qquad \text{(A.20)}\\
\mathbf{w}^*(t+1) &= \mathbf{w}^*(t) + \alpha(t)\cdot\left(\mathbf{Q} - \mathbf{w}^*(t)\cdot\mathbf{R}\right) \qquad \text{(A.21)}
\end{aligned}
$$

It can easily be shown that this formula is equivalent to equation (1.32) with $\sigma(v) = v$.

A.3 The back-propagation rule with thresholds

When the thresholds of a multi-layer feed-forward network are treated as weights, the thresholds of the first layer are associated with a constant unity input, while those of following layers are connected to pseudo-neurons. The weights of these units are initialized to yield a unity output, as shown by equation (1.19), and must not be updated. For the back-propagation rule, this is achieved by modifying either equation (1.34) or equation (1.35) (both methods can be adapted to epoch updating). In the first case, these equations become:

$$
\begin{aligned}
\delta_i^{\theta[k]}(t) &= \vec{\mathbf{w}}_i^{\theta[k+1]^{\mathrm{T}}}(t)\cdot\vec{\delta}^{\theta[k+1]}(t)\cdot\sigma'\left(p_i^{\theta[k]}(t)\right) && \text{for } i = 1,2,\ldots,m_k \text{ for } k = 1,2,\ldots,L-1 \qquad \text{(A.22)}\\
\delta_{m_k+1}^{\theta[k]}(t) &= 0 && \text{for } k = 1,2,\ldots,L-1 \qquad \text{(A.23)}\\
\mathbf{w}^{\theta[k]}(t+1) &= \mathbf{w}^{\theta[k]}(t) + \alpha(t)\cdot\vec{\delta}^{\theta[k]}(t)\cdot\vec{\mathbf{y}}^{\theta[k-1]^{\mathrm{T}}}(t) && \text{for } k = 1,2,\ldots,L \qquad \text{(A.24)}
\end{aligned}
$$

Using the second technique yields:

$$
\begin{aligned}
\vec{\delta}^{\theta[k]}(t) &= \left(\mathbf{w}^{\theta[k+1]^{\mathrm{T}}}(t)\cdot\vec{\delta}^{\theta[k+1]}(t)\right)\circ\sigma'\left(\vec{\mathbf{p}}^{\theta[k]}(t)\right) && \text{for } k = 1,2,\ldots,L-1 \qquad \text{(A.25)}\\
\mathbf{w}_i^{\theta[k]}(t+1) &= \mathbf{w}_i^{\theta[k]}(t) + \alpha(t)\cdot\delta_i^{\theta[k]}(t)\cdot\vec{\mathbf{y}}^{\theta[k-1]^{\mathrm{T}}}(t) && \text{for } i = 1,2,\ldots,m_k \text{ for } k = 1,2,\ldots,L \qquad \text{(A.26)}\\
\mathbf{w}_{m_k+1}^{\theta[k]}(t+1) &= \mathbf{w}_{m_k+1}^{\theta[k]}(t) && \text{for } k = 1,2,\ldots,L-1 \qquad \text{(A.27)}
\end{aligned}
$$

B

Addendum to Algorithm Mapping

B.1 The back-propagation rule

The implementation of the back-propagation rule on the MANTRA I machine is discussed in section 6.1.2. The second version, algorithm 6.4, can be extended to virtual matrices. Algorithm B.1 (a) presents the mapping of the feed-forward phase. This block should be iterated for each layer $k = 1, 2, \ldots, L$, with $\vec{\mathbf{y}}^{[0]} = \vec{\mathbf{x}}$. Algorithms B.1 (b), (c), and (d) show the learning phase for the last layer L, the intermediate layers $k = L-1, \ldots, 2$, and the first one, respectively.

The maximum epoch size is $2N$, longer epochs being implementable by replication of the different phases. When the epoch length e is smaller than $2N$, phases $\mathrm{A}^{[k]}_{[i,j]}$ and $\mathrm{B}^{[k]}_{[i,j]}$ must be padded with $2N - e$ additional NOP instructions.

B.2 The Kohonen model with minimum

The implementation of the Kohonen model with minimum and Euclidean distance is presented in section 6.1.3. For physical matrices, the mapping of this model on the MANTRA I machine is shown in algorithm 6.6. Algorithm B.2 presents the same scheme modified to accept virtual matrices. Like for all virtual-matrix algorithms, the maximum epoch length is $2N$ unless the different phases are correctly replicated, in which case optimal epochs contain multiples of $2N$ prototypes.

The scheduling of the phase $\mathrm{B}_{[i,j]}$ is somehow unconventional. During each block $\mathrm{B}_{[i,1]} - \mathrm{B}_{[i,q]}$, the phase $\mathrm{B}_{[i,i]}$ is first processed, followed by all others (in any order). This is not required by the GENES IV array—for which any order would be suitable—but saves memory. This scheme being the only one where a unique sub-vector is injected into the array each phase, the potential vectors $\vec{\mathbf{p}}$ can be stored either in the XY memory or in the auxiliary Y memory. With any other scheduling, two different sub-vectors should be entered into the array during the first phase of each block, requiring the potential vectors $\vec{\mathbf{p}}$ to be copied into both memories.

	Function-of-Y unit	Delta unit	GENES IV array	Sigma unit
$A^{[k]}_{[1,1]}$			$\vec{p}^{\,*[k]}_{[1]} := w^{*[k]}_{[1,1]} \cdot \vec{y}^{\,*[k-1]}_{[1]}$	
$A^{[k]}_{[1,2]}$			$\vec{p}^{\,*[k]}_{[1]} := \vec{p}^{\,*[k]}_{[1]} + w^{*[k]}_{[1,2]} \cdot \vec{y}^{\,*[k-1]}_{[2]}$	
⋮			⋮	
$A^{[k]}_{[1,r]}$			$\vec{p}^{\,*[k]}_{[1]} := \vec{p}^{\,*[k]}_{[1]} + w^{*[k]}_{[1,r]} \cdot \vec{y}^{\,*[k-1]}_{[r]}$	$\vec{y}^{\,*[k]}_{[1]} := \sigma(\vec{p}^{\,*[k]}_{[1]})$
⋮				
$A^{[k]}_{[q,1]}$			$\vec{p}^{\,*[k]}_{[q]} := w^{*[k]}_{[q,1]} \cdot \vec{y}^{\,*[k-1]}_{[1]}$	
$A^{[k]}_{[q,2]}$			$\vec{p}^{\,*[k]}_{[q]} := \vec{p}^{\,*[k]}_{[q]} + w^{*[k]}_{[q,2]} \cdot \vec{y}^{\,*[k-1]}_{[2]}$	
⋮			⋮	
$A^{[k]}_{[q,r]}$			$\vec{p}^{\,*[k]}_{[q]} := \vec{p}^{\,*[k]}_{[q]} + w^{*[k]}_{[q,r]} \cdot \vec{y}^{\,*[k-1]}_{[r]}$	$\vec{y}^{\,*[k]}_{[q]} := \sigma(\vec{p}^{\,*[k]}_{[q]})$

(a)

Algorithm B.1: The back-propagation rule using virtual matrices.
(a) Feed-forward phase, iterated for each layer $k = 1,2,\ldots,L$ (where $q = \lceil \frac{m_k}{N} \rceil$ and $r = \lceil \frac{m_{k-1}}{N} \rceil$).

	Function-of-Y unit	Delta unit	GENES IV array	Sigma unit
$\mathrm{B}^{[L]}_{[1,1]}$	$\vec{\mathbf{f}}^{[L]}_{[1]} := \sigma'\left(\sigma^{-1}\left(\vec{\mathbf{y}}^{[L]}_{[1]}\right)\right)$	$\vec{\delta}^{[L]}_{[1]} := \left(\vec{\mathbf{d}}_{[1]} - \vec{\mathbf{y}}^{[L]}_{[1]}\right) \circ \vec{\mathbf{f}}^{[L]}_{[1]}$	$\vec{\varepsilon}^{[L-1]}_{[1]} := \mathbf{w}^{[L]^{\mathrm{T}}}_{[1,1]} \cdot \vec{\delta}^{[L]}_{[1]}$	
$\mathrm{B}^{[L]}_{[2,1]}$	$\vec{\mathbf{f}}^{[L]}_{[2]} := \sigma'\left(\sigma^{-1}\left(\vec{\mathbf{y}}^{[L]}_{[2]}\right)\right)$	$\vec{\delta}^{[L]}_{[2]} := \left(\vec{\mathbf{d}}_{[2]} - \vec{\mathbf{y}}^{[L]}_{[2]}\right) \circ \vec{\mathbf{f}}^{[L]}_{[2]}$	$\vec{\varepsilon}^{[L-1]}_{[1]} := \vec{\varepsilon}^{[L-1]}_{[1]} + \mathbf{w}^{[L]^{\mathrm{T}}}_{[2,1]} \cdot \vec{\delta}^{[L]}_{[2]}$	
$\vdots$	$\vdots$	$\vdots$	$\vdots$	
$\mathrm{B}^{[L]}_{[q,1]}$	$\vec{\mathbf{f}}^{[L]}_{[q]} := \sigma'\left(\sigma^{-1}\left(\vec{\mathbf{y}}^{[L]}_{[q]}\right)\right)$	$\vec{\delta}^{[L]}_{[q]} := \left(\vec{\mathbf{d}}_{[q]} - \vec{\mathbf{y}}^{[L]}_{[q]}\right) \circ \vec{\mathbf{f}}^{[L]}_{[q]}$	$\vec{\varepsilon}^{[L-1]}_{[1]} := \vec{\varepsilon}^{[L-1]}_{[1]} + \mathbf{w}^{[L]^{\mathrm{T}}}_{[q,1]} \cdot \vec{\delta}^{[L]}_{[q]}$	$\vec{\varepsilon}^{[L-1]}_{[1]} := \vec{\varepsilon}^{[L-1]}_{[1]}$
		$\vdots$		
$\mathrm{B}^{[L]}_{[1,r]}$	$\vec{\mathbf{f}}^{[L]}_{[1]} := \sigma'\left(\sigma^{-1}\left(\vec{\mathbf{y}}^{[L]}_{[1]}\right)\right)$	$\vec{\delta}^{[L]}_{[1]} := \left(\vec{\mathbf{d}}_{[1]} - \vec{\mathbf{y}}^{[L]}_{[1]}\right) \circ \vec{\mathbf{f}}^{[L]}_{[1]}$	$\vec{\varepsilon}^{[L-1]}_{[r]} := \mathbf{w}^{[L]^{\mathrm{T}}}_{[1,r]} \cdot \vec{\delta}^{[L]}_{[1]}$	
$\mathrm{B}^{[L]}_{[2,r]}$	$\vec{\mathbf{f}}^{[L]}_{[2]} := \sigma'\left(\sigma^{-1}\left(\vec{\mathbf{y}}^{[L]}_{[2]}\right)\right)$	$\vec{\delta}^{[L]}_{[2]} := \left(\vec{\mathbf{d}}_{[2]} - \vec{\mathbf{y}}^{[L]}_{[2]}\right) \circ \vec{\mathbf{f}}^{[L]}_{[2]}$	$\vec{\varepsilon}^{[L-1]}_{[r]} := \vec{\varepsilon}^{[L-1]}_{[r]} + \mathbf{w}^{[L]^{\mathrm{T}}}_{[2,r]} \cdot \vec{\delta}^{[L]}_{[2]}$	
$\vdots$	$\vdots$	$\vdots$	$\vdots$	
$\mathrm{B}^{[L]}_{[q,r]}$	$\vec{\mathbf{f}}^{[L]}_{[q]} := \sigma'\left(\sigma^{-1}\left(\vec{\mathbf{y}}^{[L]}_{[q]}\right)\right)$	$\vec{\delta}^{[L]}_{[q]} := \left(\vec{\mathbf{d}}_{[q]} - \vec{\mathbf{y}}^{[L]}_{[q]}\right) \circ \vec{\mathbf{f}}^{[L]}_{[q]}$	$\vec{\varepsilon}^{[L-1]}_{[r]} := \vec{\varepsilon}^{[L-1]}_{[r]} + \mathbf{w}^{[L]^{\mathrm{T}}}_{[q,r]} \cdot \vec{\delta}^{[L]}_{[q]}$	$\vec{\varepsilon}^{[L-1]}_{[r]} := \vec{\varepsilon}^{[L-1]}_{[r]}$
$\mathrm{C}^{[L]}_{[1,1]}$	$\vec{\mathbf{f}}^{[L]}_{[1]} := \alpha \cdot \sigma'\left(\sigma^{-1}\left(\vec{\mathbf{y}}^{[L]}_{[1]}\right)\right)$	$\vec{\delta}^{[L]}_{[1]} := \left(\vec{\mathbf{d}}_{[1]} - \vec{\mathbf{y}}^{[L]}_{[1]}\right) \circ \vec{\mathbf{f}}^{[L]}_{[1]}$	$\mathbf{w}^{*[L]}_{[1,1]} := \mathbf{w}^{*[L]}_{[1,1]} + \vec{\delta}^{[L]}_{[1]} \cdot \vec{\mathbf{y}}^{*[L-1]^{\mathrm{T}}}_{[1]}$	
$\mathrm{C}^{[L]}_{[1,2]}$	$\vec{\mathbf{f}}^{[L]}_{[1]} := \alpha \cdot \sigma'\left(\sigma^{-1}\left(\vec{\mathbf{y}}^{[L]}_{[1]}\right)\right)$	$\vec{\delta}^{[L]}_{[1]} := \left(\vec{\mathbf{d}}_{[1]} - \vec{\mathbf{y}}^{[L]}_{[1]}\right) \circ \vec{\mathbf{f}}^{[L]}_{[1]}$	$\mathbf{w}^{*[L]}_{[1,2]} := \mathbf{w}^{*[L]}_{[1,2]} + \vec{\delta}^{[L]}_{[1]} \cdot \vec{\mathbf{y}}^{*[L-1]^{\mathrm{T}}}_{[2]}$	
$\vdots$	$\vdots$	$\vdots$	$\vdots$	
$\mathrm{C}^{[L]}_{[1,r]}$	$\vec{\mathbf{f}}^{[L]}_{[1]} := \alpha \cdot \sigma'\left(\sigma^{-1}\left(\vec{\mathbf{y}}^{[L]}_{[1]}\right)\right)$	$\vec{\delta}^{[L]}_{[1]} := \left(\vec{\mathbf{d}}_{[1]} - \vec{\mathbf{y}}^{[L]}_{[1]}\right) \circ \vec{\mathbf{f}}^{[L]}_{[1]}$	$\mathbf{w}^{*[L]}_{[1,r]} := \mathbf{w}^{*[L]}_{[1,r]} + \vec{\delta}^{[L]}_{[1]} \cdot \vec{\mathbf{y}}^{*[L-1]^{\mathrm{T}}}_{[r]}$	
		$\vdots$		
$\mathrm{C}^{[L]}_{[q,1]}$	$\vec{\mathbf{f}}^{[L]}_{[q]} := \alpha \cdot \sigma'\left(\sigma^{-1}\left(\vec{\mathbf{y}}^{[L]}_{[q]}\right)\right)$	$\vec{\delta}^{[L]}_{[q]} := \left(\vec{\mathbf{d}}_{[q]} - \vec{\mathbf{y}}^{[L]}_{[q]}\right) \circ \vec{\mathbf{f}}^{[L]}_{[q]}$	$\mathbf{w}^{*[L]}_{[q,1]} := \mathbf{w}^{*[L]}_{[q,1]} + \vec{\delta}^{[L]}_{[q]} \cdot \vec{\mathbf{y}}^{*[L-1]^{\mathrm{T}}}_{[1]}$	
$\mathrm{C}^{[L]}_{[q,2]}$	$\vec{\mathbf{f}}^{[L]}_{[q]} := \alpha \cdot \sigma'\left(\sigma^{-1}\left(\vec{\mathbf{y}}^{[L]}_{[q]}\right)\right)$	$\vec{\delta}^{[L]}_{[q]} := \left(\vec{\mathbf{d}}_{[q]} - \vec{\mathbf{y}}^{[L]}_{[q]}\right) \circ \vec{\mathbf{f}}^{[L]}_{[q]}$	$\mathbf{w}^{*[L]}_{[q,2]} := \mathbf{w}^{*[L]}_{[q,2]} + \vec{\delta}^{[L]}_{[q]} \cdot \vec{\mathbf{y}}^{*[L-1]^{\mathrm{T}}}_{[2]}$	
$\vdots$	$\vdots$	$\vdots$	$\vdots$	
$\mathrm{C}^{[L]}_{[q,r]}$	$\vec{\mathbf{f}}^{[L]}_{[q]} := \alpha \cdot \sigma'\left(\sigma^{-1}\left(\vec{\mathbf{y}}^{[L]}_{[q]}\right)\right)$	$\vec{\delta}^{[L]}_{[q]} := \left(\vec{\mathbf{d}}_{[q]} - \vec{\mathbf{y}}^{[L]}_{[q]}\right) \circ \vec{\mathbf{f}}^{[L]}_{[q]}$	$\mathbf{w}^{*[L]}_{[q,r]} := \mathbf{w}^{*[L]}_{[q,r]} + \vec{\delta}^{[L]}_{[q]} \cdot \vec{\mathbf{y}}^{*[L-1]^{\mathrm{T}}}_{[r]}$	

(b)

Algorithm B.1: The back-propagation rule using virtual matrices.
(b) Learning phase, last layer (where $q = \left\lceil \frac{m_L}{N} \right\rceil$ and $r = \left\lceil \frac{m_{L-1}}{N} \right\rceil$).

	Function-of-Y unit	Delta unit	GENES IV array	Sigma unit
$\mathbf{B}^{[k]}_{[1,1]}$	$\vec{\mathbf{f}}^{[k]}_{[1]} := \sigma'\left(\sigma^{-1}\left(\vec{\mathbf{y}}^{[k]}_{[1]}\right)\right)$	$\vec{\delta}^{[k]}_{[1]} := \left(\vec{\varepsilon}^{[k]}_{[1]} - \vec{\mathbf{0}}\right) \circ \vec{\mathbf{f}}^{[k]}_{[1]}$	$\vec{\varepsilon}^{[k-1]}_{[1]} := \mathbf{w}^{[k]^{\mathrm{T}}}_{[1,1]} \cdot \vec{\delta}^{[k]}_{[1]}$	
$\mathbf{B}^{[k]}_{[2,1]}$	$\vec{\mathbf{f}}^{[k]}_{[2]} := \sigma'\left(\sigma^{-1}\left(\vec{\mathbf{y}}^{[k]}_{[2]}\right)\right)$	$\vec{\delta}^{[k]}_{[2]} := \left(\vec{\varepsilon}^{[k]}_{[2]} - \vec{\mathbf{0}}\right) \circ \vec{\mathbf{f}}^{[k]}_{[2]}$	$\vec{\varepsilon}^{[k-1]}_{[1]} := \vec{\varepsilon}^{[k-1]}_{[1]} + \mathbf{w}^{[k]^{\mathrm{T}}}_{[2,1]} \cdot \vec{\delta}^{[k]}_{[2]}$	
$\vdots$	$\vdots$	$\vdots$	$\vdots$	
$\mathbf{B}^{[k]}_{[q,1]}$	$\vec{\mathbf{f}}^{[k]}_{[q]} := \sigma'\left(\sigma^{-1}\left(\vec{\mathbf{y}}^{[k]}_{[q]}\right)\right)$	$\vec{\delta}^{[k]}_{[q]} := \left(\vec{\varepsilon}^{[k]}_{[q]} - \vec{\mathbf{0}}\right) \circ \vec{\mathbf{f}}^{[k]}_{[q]}$	$\vec{\varepsilon}^{[k-1]}_{[1]} := \vec{\varepsilon}^{[k-1]}_{[1]} + \mathbf{w}^{[k]^{\mathrm{T}}}_{[q,1]} \cdot \vec{\delta}^{[k]}_{[q]}$	$\vec{\varepsilon}^{[k-1]}_{[1]} := \vec{\varepsilon}^{[k-1]}_{[1]}$
		$\vdots$		
$\mathbf{B}^{[k]}_{[1,r]}$	$\vec{\mathbf{f}}^{[k]}_{[1]} := \sigma'\left(\sigma^{-1}\left(\vec{\mathbf{y}}^{[k]}_{[1]}\right)\right)$	$\vec{\delta}^{[k]}_{[1]} := \left(\vec{\varepsilon}^{[k]}_{[1]} - \vec{\mathbf{0}}\right) \circ \vec{\mathbf{f}}^{[k]}_{[1]}$	$\vec{\varepsilon}^{[k-1]}_{[r]} := \mathbf{w}^{[k]^{\mathrm{T}}}_{[1,r]} \cdot \vec{\delta}^{[k]}_{[1]}$	
$\mathbf{B}^{[k]}_{[2,r]}$	$\vec{\mathbf{f}}^{[k]}_{[2]} := \sigma'\left(\sigma^{-1}\left(\vec{\mathbf{y}}^{[k]}_{[2]}\right)\right)$	$\vec{\delta}^{[k]}_{[2]} := \left(\vec{\varepsilon}^{[k]}_{[2]} - \vec{\mathbf{0}}\right) \circ \vec{\mathbf{f}}^{[k]}_{[2]}$	$\vec{\varepsilon}^{[k-1]}_{[r]} := \vec{\varepsilon}^{[k-1]}_{[r]} + \mathbf{w}^{[k]^{\mathrm{T}}}_{[2,r]} \cdot \vec{\delta}^{[k]}_{[2]}$	
$\vdots$	$\vdots$	$\vdots$	$\vdots$	
$\mathbf{B}^{[k]}_{[q,r]}$	$\vec{\mathbf{f}}^{[k]}_{[q]} := \sigma'\left(\sigma^{-1}\left(\vec{\mathbf{y}}^{[k]}_{[q]}\right)\right)$	$\vec{\delta}^{[k]}_{[q]} := \left(\vec{\varepsilon}^{[k]}_{[q]} - \vec{\mathbf{0}}\right) \circ \vec{\mathbf{f}}^{[k]}_{[q]}$	$\vec{\varepsilon}^{[k-1]}_{[r]} := \vec{\varepsilon}^{[k-1]}_{[r]} + \mathbf{w}^{[k]^{\mathrm{T}}}_{[q,r]} \cdot \vec{\delta}^{[k]}_{[q]}$	$\vec{\varepsilon}^{[k-1]}_{[r]} := \vec{\varepsilon}^{[k-1]}_{[r]}$
$\mathbf{C}^{[k]}_{[1,1]}$	$\vec{\mathbf{f}}^{[k]}_{[1]} := \alpha \cdot \sigma'\left(\sigma^{-1}\left(\vec{\mathbf{y}}^{[k]}_{[1]}\right)\right)$	$\vec{\delta}^{[k]}_{[1]} := \left(\vec{\varepsilon}^{[k]}_{[1]} - \vec{\mathbf{0}}\right) \circ \vec{\mathbf{f}}^{[k]}_{[1]}$	$\mathbf{w}^{*[k]}_{[1,1]} := \mathbf{w}^{*[k]}_{[1,1]} + \vec{\delta}^{*[k]}_{[1]} \cdot \vec{\mathbf{y}}^{*[k-1]^{\mathrm{T}}}_{[1]}$	
$\mathbf{C}^{[k]}_{[1,2]}$	$\vec{\mathbf{f}}^{[k]}_{[1]} := \alpha \cdot \sigma'\left(\sigma^{-1}\left(\vec{\mathbf{y}}^{[k]}_{[1]}\right)\right)$	$\vec{\delta}^{[k]}_{[1]} := \left(\vec{\varepsilon}^{[k]}_{[1]} - \vec{\mathbf{0}}\right) \circ \vec{\mathbf{f}}^{[k]}_{[1]}$	$\mathbf{w}^{*[k]}_{[1,2]} := \mathbf{w}^{*[k]}_{[1,2]} + \vec{\delta}^{*[k]}_{[1]} \cdot \vec{\mathbf{y}}^{*[k-1]^{\mathrm{T}}}_{[2]}$	
$\vdots$	$\vdots$	$\vdots$	$\vdots$	
$\mathbf{C}^{[k]}_{[1,r]}$	$\vec{\mathbf{f}}^{[k]}_{[1]} := \alpha \cdot \sigma'\left(\sigma^{-1}\left(\vec{\mathbf{y}}^{[k]}_{[1]}\right)\right)$	$\vec{\delta}^{[k]}_{[1]} := \left(\vec{\varepsilon}^{[k]}_{[1]} - \vec{\mathbf{0}}\right) \circ \vec{\mathbf{f}}^{[k]}_{[1]}$	$\mathbf{w}^{*[k]}_{[1,r]} := \mathbf{w}^{*[k]}_{[1,r]} + \vec{\delta}^{*[k]}_{[1]} \cdot \vec{\mathbf{y}}^{*[k-1]^{\mathrm{T}}}_{[r]}$	
		$\vdots$		
$\mathbf{C}^{[k]}_{[q,1]}$	$\vec{\mathbf{f}}^{[k]}_{[q]} := \alpha \cdot \sigma'\left(\sigma^{-1}\left(\vec{\mathbf{y}}^{[k]}_{[q]}\right)\right)$	$\vec{\delta}^{[k]}_{[q]} := \left(\vec{\varepsilon}^{[k]}_{[q]} - \vec{\mathbf{0}}\right) \circ \vec{\mathbf{f}}^{[k]}_{[q]}$	$\mathbf{w}^{*[k]}_{[q,1]} := \mathbf{w}^{*[k]}_{[q,1]} + \vec{\delta}^{*[k]}_{[q]} \cdot \vec{\mathbf{y}}^{*[k-1]^{\mathrm{T}}}_{[1]}$	
$\mathbf{C}^{[k]}_{[q,2]}$	$\vec{\mathbf{f}}^{[k]}_{[q]} := \alpha \cdot \sigma'\left(\sigma^{-1}\left(\vec{\mathbf{y}}^{[k]}_{[q]}\right)\right)$	$\vec{\delta}^{[k]}_{[q]} := \left(\vec{\varepsilon}^{[k]}_{[q]} - \vec{\mathbf{0}}\right) \circ \vec{\mathbf{f}}^{[k]}_{[q]}$	$\mathbf{w}^{*[k]}_{[q,2]} := \mathbf{w}^{*[k]}_{[q,2]} + \vec{\delta}^{*[k]}_{[q]} \cdot \vec{\mathbf{y}}^{*[k-1]^{\mathrm{T}}}_{[2]}$	
$\vdots$	$\vdots$	$\vdots$	$\vdots$	
$\mathbf{C}^{[k]}_{[q,r]}$	$\vec{\mathbf{f}}^{[k]}_{[q]} := \alpha \cdot \sigma'\left(\sigma^{-1}\left(\vec{\mathbf{y}}^{[k]}_{[q]}\right)\right)$	$\vec{\delta}^{[k]}_{[q]} := \left(\vec{\varepsilon}^{[k]}_{[q]} - \vec{\mathbf{0}}\right) \circ \vec{\mathbf{f}}^{[k]}_{[q]}$	$\mathbf{w}^{*[k]}_{[q,r]} := \mathbf{w}^{*[k]}_{[q,r]} + \vec{\delta}^{*[k]}_{[q]} \cdot \vec{\mathbf{y}}^{*[k-1]^{\mathrm{T}}}_{[r]}$	

(c)

Algorithm B.1: The back-propagation rule using virtual matrices.
(c) Learning phase, iterated for each layer $k = L - 1,\ldots,2$ (where $q = \left\lceil \frac{m_k}{N} \right\rceil$ and $r = \left\lceil \frac{m_{k-1}}{N} \right\rceil$).

	Function-of-Y unit	Delta unit	GENES IV array	Sigma unit
$C^{[1]}_{[1,1]}$	$\vec{\mathbf{f}}^{[1]}_{[1]} := \alpha \cdot \sigma'\left(\sigma^{-1}\left(\vec{\mathbf{y}}^{[1]}_{[1]}\right)\right)$	$\vec{\delta}^{[1]}_{[1]} := \left(\vec{\varepsilon}^{[1]}_{[1]} - \vec{\mathbf{0}}\right) \circ \vec{\mathbf{f}}^{[1]}_{[1]}$	$\mathbf{w}^{*[1]}_{[1,1]} := \mathbf{w}^{*[1]}_{[1,1]} + \vec{\delta}^{*[1]}_{[1]} \cdot \vec{\mathbf{x}}^{*\mathrm{T}}_{[1]}$	
$C^{[1]}_{[1,2]}$	$\vec{\mathbf{f}}^{[1]}_{[1]} := \alpha \cdot \sigma'\left(\sigma^{-1}\left(\vec{\mathbf{y}}^{[1]}_{[1]}\right)\right)$	$\vec{\delta}^{[1]}_{[1]} := \left(\vec{\varepsilon}^{[1]}_{[1]} - \vec{\mathbf{0}}\right) \circ \vec{\mathbf{f}}^{[1]}_{[1]}$	$\mathbf{w}^{*[1]}_{[1,2]} := \mathbf{w}^{*[1]}_{[1,2]} + \vec{\delta}^{*[1]}_{[1]} \cdot \vec{\mathbf{x}}^{*\mathrm{T}}_{[2]}$	
$\vdots$	$\vdots$	$\vdots$	$\vdots$	
$C^{[1]}_{[1,r]}$	$\vec{\mathbf{f}}^{[1]}_{[1]} := \alpha \cdot \sigma'\left(\sigma^{-1}\left(\vec{\mathbf{y}}^{[1]}_{[1]}\right)\right)$	$\vec{\delta}^{[1]}_{[1]} := \left(\vec{\varepsilon}^{[1]}_{[1]} - \vec{\mathbf{0}}\right) \circ \vec{\mathbf{f}}^{[1]}_{[1]}$	$\mathbf{w}^{*[1]}_{[1,r]} := \mathbf{w}^{*[1]}_{[1,r]} + \vec{\delta}^{*[1]}_{[1]} \cdot \vec{\mathbf{x}}^{*\mathrm{T}}_{[r]}$	
$\vdots$				
$C^{[1]}_{[q,1]}$	$\vec{\mathbf{f}}^{[1]}_{[q]} := \alpha \cdot \sigma'\left(\sigma^{-1}\left(\vec{\mathbf{y}}^{[1]}_{[q]}\right)\right)$	$\vec{\delta}^{[1]}_{[q]} := \left(\vec{\varepsilon}^{[1]}_{[q]} - \vec{\mathbf{0}}\right) \circ \vec{\mathbf{f}}^{[1]}_{[q]}$	$\mathbf{w}^{*[1]}_{[q,1]} := \mathbf{w}^{*[1]}_{[q,1]} + \vec{\delta}^{*[1]}_{[q]} \cdot \vec{\mathbf{x}}^{*\mathrm{T}}_{[1]}$	
$C^{[1]}_{[q,2]}$	$\vec{\mathbf{f}}^{[1]}_{[q]} := \alpha \cdot \sigma'\left(\sigma^{-1}\left(\vec{\mathbf{y}}^{[1]}_{[q]}\right)\right)$	$\vec{\delta}^{[1]}_{[q]} := \left(\vec{\varepsilon}^{[1]}_{[q]} - \vec{\mathbf{0}}\right) \circ \vec{\mathbf{f}}^{[1]}_{[q]}$	$\mathbf{w}^{*[1]}_{[q,2]} := \mathbf{w}^{*[1]}_{[q,2]} + \vec{\delta}^{*[1]}_{[q]} \cdot \vec{\mathbf{x}}^{*\mathrm{T}}_{[2]}$	
$\vdots$	$\vdots$	$\vdots$	$\vdots$	
$C^{[1]}_{[q,r]}$	$\vec{\mathbf{f}}^{[1]}_{[q]} := \alpha \cdot \sigma'\left(\sigma^{-1}\left(\vec{\mathbf{y}}^{[1]}_{[q]}\right)\right)$	$\vec{\delta}^{[1]}_{[q]} := \left(\vec{\varepsilon}^{[1]}_{[q]} - \vec{\mathbf{0}}\right) \circ \vec{\mathbf{f}}^{[1]}_{[q]}$	$\mathbf{w}^{*[1]}_{[q,r]} := \mathbf{w}^{*[1]}_{[q,r]} + \vec{\delta}^{*[1]}_{[q]} \cdot \vec{\mathbf{x}}^{*\mathrm{T}}_{[r]}$	

(d)

Algorithm B.1: The back-propagation rule using virtual matrices.
(d) Learning phase, first layer (where $q = \left\lceil \frac{m_1}{N} \right\rceil$ and $r = \left\lceil \frac{n}{N} \right\rceil$).

	Function-of-Y unit	GENES IV array	Sigma unit
$A_{[1,1]}$		$p_{[1]_i} := \lvert\vec{\mathbf{x}}_{[1]} - \mathbf{w}^{\mathrm{T}}_{[1,1]_i}\rvert^2$	
$A_{[1,2]}$		$p_{[1]_i} := p_{[1]_i} + \lvert\vec{\mathbf{x}}_{[2]} - \mathbf{w}^{\mathrm{T}}_{[1,2]_i}\rvert^2$	
$\vdots$		$\vdots$	
$A_{[1,r]}$		$p_{[1]_i} := p_{[1]_i} + \lvert\vec{\mathbf{x}}_{[r]} - \mathbf{w}^{\mathrm{T}}_{[1,r]_i}\rvert^2$	$\vec{\mathbf{p}}_{[1]} := \vec{\mathbf{p}}_{[1]}$
$\vdots$			
$A_{[q,1]}$		$p_{[q]_i} := \lvert\vec{\mathbf{x}}_{[1]} - \mathbf{w}^{\mathrm{T}}_{[q,1]_i}\rvert^2$	
$A_{[q,2]}$		$p_{[q]_i} := p_{[q]_i} + \lvert\vec{\mathbf{x}}_{[2]} - \mathbf{w}^{\mathrm{T}}_{[q,2]_i}\rvert^2$	
$\vdots$		$\vdots$	
$A_{[q,r]}$		$p_{[q]_i} := p_{[q]_i} + \lvert\vec{\mathbf{x}}_{[r]} - \mathbf{w}^{\mathrm{T}}_{[q,r]_i}\rvert^2$	$\vec{\mathbf{p}}_{[q]} := \vec{\mathbf{p}}_{[q]}$

(a)

Algorithm B.2: The Kohonen model with minimum and Euclidean distance using virtual matrices.
(a) Computation of the squared Euclidean distance (where $q = \lceil \frac{m}{N} \rceil$ and $r = \lceil \frac{n}{N} \rceil$).

	Function-of-Y unit	GENES IV array	Sigma unit
$B_{[1,1]}$	$p_{[1]_i} := N_{\max} - p_{[1]_i}$	$\mu_{[1]_i} := \begin{cases} p_{[1]_i} & \text{if } p_{[1]_i} = \max(\vec{\mathbf{p}}_{[1]}) \\ N_{\min} & \text{otherwise} \end{cases}$	
$B_{[1,2]}$	$p_{[2]_i} := N_{\max} - p_{[2]_i}$	$\mu_{[1]_i} := \begin{cases} \mu_{[1]_i} & \text{if } \mu_{[1]_i} \geq \max(\vec{\mathbf{p}}_{[2]}) \\ N_{\min} & \text{otherwise} \end{cases}$	
$\vdots$		$\vdots$	
$B_{[1,q]}$	$p_{[q]_i} := N_{\max} - p_{[q]_i}$	$\mu_{[1]_i} := \begin{cases} \mu_{[1]_i} & \text{if } \mu_{[1]_i} \geq \max(\vec{\mathbf{p}}_{[q]}) \\ N_{\min} & \text{otherwise} \end{cases}$	$v_{[1]_i} := \begin{cases} 1 & \text{if } \mu_{[1]_i} \neq N_{\min} \\ 0 & \text{if } \mu_{[1]_i} = N_{\min} \end{cases}$
$\vdots$			
$B_{[q,q]}$	$p_{[q]_i} := N_{\max} - p_{[q]_i}$	$\mu_{[q]_i} := \begin{cases} p_{[q]_i} & \text{if } p_{[q]_i} = \max(\vec{\mathbf{p}}_{[q]}) \\ N_{\min} & \text{otherwise} \end{cases}$	
$B_{[q,1]}$	$p_{[1]_i} := N_{\max} - p_{[1]_i}$	$\mu_{[q]_i} := \begin{cases} \mu_{[q]_i} & \text{if } \mu_{[q]_i} \geq \max(\vec{\mathbf{p}}_{[1]}) \\ N_{\min} & \text{otherwise} \end{cases}$	
$\vdots$		$\vdots$	
$B_{[q,q-1]}$	$p_{[q-1]_i} := N_{\max} - p_{[q-1]_i}$	$\mu_{[q]_i} := \begin{cases} \mu_{[q]_i} & \text{if } \mu_{[q]_i} \geq \max(\vec{\mathbf{p}}_{[q-1]}) \\ N_{\min} & \text{otherwise} \end{cases}$	$v_{[q]_i} := \begin{cases} 1 & \text{if } \mu_{[q]_i} \neq N_{\min} \\ 0 & \text{if } \mu_{[q]_i} = N_{\min} \end{cases}$

(b)

Algorithm B.2: The Kohonen model with minimum and Euclidean distance using virtual matrices. (b) Winner selection (where $q = \lceil \frac{m}{N} \rceil$).

	Function-of-Y unit	GENES IV array	Sigma unit
$C_{[1,1]}$		$\vec{\nu}_{[1]} := \alpha \cdot \lambda_{[1,1]} \cdot \vec{\upsilon}_{[1]}$	
$C_{[1,2]}$		$\vec{\nu}_{[1]} := \vec{\nu}_{[1]} + \alpha \cdot \lambda_{[1,2]} \cdot \vec{\upsilon}_{[2]}$	
$\vdots$		$\vdots$	
$C_{[1,q]}$		$\vec{\nu}_{[1]} := \vec{\nu}_{[1]} + \alpha \cdot \lambda_{[1,q]} \cdot \vec{\upsilon}_{[q]}$	$\vec{\nu}_{[1]} := \vec{\nu}_{[1]}$
$D_{[1,1]}$	$\vec{\nu}_{[1]} := \vec{\nu}_{[1]}$	$\mathbf{w}_{[1,1]_i} := \mathbf{w}_{[1,1]_i} + \nu_{[1]_i} \cdot (\vec{\mathbf{x}}^{\mathrm{T}}_{[1]} - \mathbf{w}_{[1,1]_i})$	
$\vdots$	$\vdots$	$\vdots$	
$D_{[1,r]}$	$\vec{\nu}_{[1]} := \vec{\nu}_{[1]}$	$\mathbf{w}_{[1,r]_i} := \mathbf{w}_{[1,r]_i} + \nu_{[1]_i} \cdot (\vec{\mathbf{x}}^{\mathrm{T}}_{[r]} - \mathbf{w}_{[1,r]_i})$	
		$\vdots$	
$C_{[q,1]}$		$\vec{\nu}_{[q]} := \alpha \cdot \lambda_{[q,1]} \cdot \vec{\upsilon}_{[1]}$	
$C_{[q,2]}$		$\vec{\nu}_{[q]} := \vec{\nu}_{[q]} + \alpha \cdot \lambda_{[q,2]} \cdot \vec{\upsilon}_{[2]}$	
$\vdots$		$\vdots$	
$C_{[q,q]}$		$\vec{\nu}_{[q]} := \vec{\nu}_{[q]} + \alpha \cdot \lambda_{[q,q]} \cdot \vec{\upsilon}_{[q]}$	$\vec{\nu}_{[q]} := \vec{\nu}_{[q]}$
$D_{[q,1]}$	$\vec{\nu}_{[q]} := \vec{\nu}_{[q]}$	$\mathbf{w}_{[q,1]_i} := \mathbf{w}_{[q,1]_i} + \nu_{[q]_i} \cdot (\vec{\mathbf{x}}^{\mathrm{T}}_{[1]} - \mathbf{w}_{[q,1]_i})$	
$\vdots$	$\vdots$	$\vdots$	
$D_{[q,r]}$	$\vec{\nu}_{[q]} := \vec{\nu}_{[q]}$	$\mathbf{w}_{[q,r]_i} := \mathbf{w}_{[q,r]_i} + \nu_{[q]_i} \cdot (\vec{\mathbf{x}}^{\mathrm{T}}_{[r]} - \mathbf{w}_{[q,r]_i})$	

(c)

Algorithm B.2: The Kohonen model with minimum and Euclidean distance using virtual matrices. (c) Weight update (where $q = \lceil \frac{m}{N} \rceil$ and $r = \lceil \frac{n}{N} \rceil$).

C

Benchmark Description

It is a commonly accepted principle of good practice that any study involving benchmarking should include a precise description of the benchmarks, so that any reader could re-implement them and get the same results. On this subject, J. Hennessy and D. Patterson [HEN90] write:

> *The guiding principle of reporting performance measurements should be* reproducibility—*list everything another experimenter would need to duplicate the results.*

This appendix describes first the implementation of the delta rule on a floating-point microprocessor (TMS320C40) and on the MANTRA I machine, followed by a description of the benchmark used in the analysis of section 7.4.

C.1 The delta rule

The floating-point version of the delta rule is a C procedure running on the control processor of the MANTRA I machine (TMS320C40). This routine implements equation (1.32). The learning set is divided into epochs according to equation (1.25), the epoch length e being a parameter. Prototypes are repeatedly presented without shuffling. The activation function σ, its derivative σ', and the learning coefficient α are also passed to the procedure as parameters. Data are represented using the proprietary TMS320C40 floating-point format [TI91A]. They are stored in memory as 32-bit words, but some intermediate results are represented on 40 bits.

The MANTRA I version of the delta rule implements also equation (1.32), and the prototypes are presented exactly in the same order. This model is mapped on the machine as shown by algorithm 6.1. The activation function σ, its derivative σ', and the learning coefficient α are coded in the sigma and function-of-Y units (see sections 5.1.2 and 5.1.3). The detailed computation is discussed in sections 4.1.1 and 5.1.1. The arithmetic precision and the overflow handling are described in section 4.2.5 and 4.2.7 respectively.

C.2 Convergence analysis of the delta rule

The benchmark used to analyze the convergence of the delta rule on the MANTRA I machine is a classification problem of linearly separable regions. The input space is a hyper-cube of $n = 99$ dimensions, with each element x_j of the input vector belonging to the range $[-x_{\max}, x_{\max}]$, where $x_{\max} = 1$. With the addition of a constant input $x_{n+1}(s) = \frac{x_{\max}}{2} = \frac{1}{2}$ used to handle thresholds as weights, the total number of network inputs is $n^* = n + 1 = 100$.

A network of $m = 20$ neurons performs an equal number of linear separations in parallel. For each neuron, a hyper-plane is randomly generated. Its distance to the origin is limited to $x_{\max} = 1$. A training set with $S_{\text{lrn}} = 10000$ prototypes and a testing set of $S_{\text{tst}} = 1000$ prototypes are randomly generated, and the signed distance $\Delta(\vec{\mathbf{x}})$ of each input vector to each hyper-plane is computed. The corresponding desired output is set to $\beta = 1$ if this vector is on the positive side and to $-\beta = -1$ if it is on the negative one, with an error probability (noise) of:

$$\frac{0.5}{1 + 100 \cdot \frac{|\Delta(\vec{\mathbf{x}}(s))|}{x_{\max}}}$$

A hyperbolic tangent is used as activation function:

$$\sigma(v) = \beta \cdot \tanh\left(\frac{v-\theta}{T}\right) = \tanh(10v) \tag{C.1}$$

Hence, the activation function's derivative is:

$$\sigma(v) = \frac{\beta}{T} \cdot \left(1 - \tanh^2\left(\frac{v-\theta}{T}\right)\right) = 10 \cdot \left(1 - \tanh^2(10v)\right) \tag{C.2}$$

The weight matrix being initialized to zero (i.e., $\mathbf{w}(0) = \mathbf{0}$), all elements of the initial error vector $\vec{\varepsilon}(0,s) = \vec{\mathbf{d}}(s) - \vec{\mathbf{y}}(0,s)$ are equal to $+1$ or -1 for all prototypes. Therefore, the average quadratic error is equal to $\xi_{\text{avr}}(0) = 1$.

In the MANTRA I version, all data are scaled as specified in section 6.2.1, using the factors a_w, a_x, and a_y. The sigma unit (see section 5.1.2) is initialized with equation (6.10), and the function-of-Y unit (see section 5.1.3) with equation (6.13). The $\text{SCGSH}_{2..0}$ and $\text{SFGSH}_{2..0}$ fields of the configuration register are set to 1 and 7, respectively. The sigma fine-grain table is centered on 0.

C.3 Programming tools

TMS320C40 code has been produced using the `c130` C compiler [TI91C] and the `asm30` assembler [TI91B] from Texas Instruments (version 4.50). The flags used for the experiments of sections 7.3 and 7.4 are `-o2 -om` and `-o1 -om`, respectively. Standard C routines are linked from the `rts` library (compiled with `-o2 -om`), while the `mathasm` library is used for mathematical functions.

References

[HEN90] J. L. Hennessy and D. A. Patterson. *Computer Architecture: A Quantitative Approach.* Morgan Kaufmann Publishers, San Mateo, CA (USA), 1990.

[TI91A] Texas Instruments. *TMS320C4x User's Guide*, May 1991. Revision A.

[TI91B] Texas Instruments. *TMS320 Floating-Point DSP Assembly Language Tools*, September 1991. Revision A.

[TI91C] Texas Instruments. *TMS320 Floating-Point DSP Optimizing C Compiler*, October 1991. Revision A.

List of Acronyms

ACM	Association for Computing Machinery.
AFIPS	American Federation of Information Processing Societies.
AI	Artificial intelligence.
AIP	American Institute of Physics.
ALU	Arithmetic and logic unit.
ANN	Artificial neural network.
ANNA	Artificial Neural Network ALU.
ANSI	American National Standards Institute.
ASE	*Association Suisse des Électriciens.*
ASIC	Application-specific integrated circuit.
CDC	Control Data Corporation.
CISC	Complex instruction-set computer.
CM	Connection Machine.
CMOS	Complementary metal-oxide semiconductor.
CMU	Carnegie-Mellon University.
CNAPS	Connected Network of Adaptive Processors.
COMPCON	IEEE Computer Society Conference.
COST	European Cooperation in the Field of Scientific and Technical Research.
CPS	Connections per second.
CUPS	Connections updates per second.
DEC	Digital Equipment Corporation.
DI	*Département d'Informatique.*
DMA	Direct memory access.
DRAM	Dynamic random access memory.
DSP	Digital signal processor.
DVMA	Direct virtual memory access.

ECE	Electrical and computer engineering.
EDMCC2	2nd European Distributed Memory Computing Conference.
EDSAC	Electronic Delay Storage Automatic Calculator.
EDVAC	Electronic Discrete Variable Automatic Computer.
ENIAC	Electronic Numerical Integrator and Calculator.
EPFL	*École Polytechnique Fédérale de Lausanne.*
EPLD	Erasable programmable logic device.
ETH	*Eidgenössische Technische Hochschule.*
FIFO	First-in first-out queue.
FLOPS	Floating-point operations per second.
FPU	Floating-point unit.
GACD1	GENES Auxiliary Circuit for the Delta Rule, 1st version.
GENES	Generic Element for Neuro-Emulator Systolic Arrays.
GF11	11 GFLOPS.
HH8	Hopfield Network with Hebbian Learning Rule, 8 Bits.
HN8	Hopfield Network, 8 Bits.
HNC	Hecht-Nielsen Corporation.
IBA	Instruction broadcasting array.
IBM	International Business Machines.
ICANN	International Conference on Artificial Neural Networks.
ICSI	International Computer Science Institute.
IEEE	Institute of Electrical and Electronics Engineers.
IEICE	Institute of Electronics, Information and Communication Engineers.
IFIP	International Federation of Information Processing.
IIS	*Institut für Integrierte Systeme.*
INNS	International Neural Network Society.
INPG	*Institut National Polytechnique de Grenoble.*
IRE	Institute of Radio Engineers.
IRL	Incremental and robust learning.
ISA	Instruction systolic array.
IWANN	International Workshop on Artificial Neural Networks.
LAMI	*Laboratoire de Microinformatique.*
LED	Light emitting diode.
LEP	*Laboratoire d'Électronique Philips.*
LMS	Least mean square.
LSB	Least-significant bit.
LSI	Large scale integration.
MIMD	Multiple instruction stream – multiple data stream.
MISD	Multiple instruction stream – single data stream.
MIT	Massachusetts Institute of Technology.
MP	MasPar.
MSB	Most-significant bit.
MSC	Mass Storage Controller.
MU	Memory unit.
MUSIC	Multiprocessor System with Intelligent Communication.

NEC	Nippon Electric Corporation.
NIPS	IEEE Conference on Neural Information Processing Systems.
NN	Neural network.
NOP	No operation.
NPE	Neural processing element.
NUMA	Non-uniformed memory access.
PAL	Programmable array logic.
PC	Personal computer.
PCB	Printed circuit board.
PE	Processing element.
PE	*Processeur élémentaire.*
PGA	Programmable gate array.
RAM	Random access memory.
RAP	Ring Array Processor.
RISC	Reduced instruction-set computer.
RNA	*Réseau de neurones artificiels.*
ROM	Read only memory.
RP3	Research Parallel Processor Prototype.
SAM	Serial access memory.
SEV	*Schweizerischer Elektrotechnischer Verein.*
SIGARCH	Special Interest Group on Computer Architecture.
SIMD	Single instruction stream – multiple data stream.
SISD	Single instruction stream – single data stream.
SMART	Sparse Matrix and Recursive Transform.
SMD	Surface mounted device.
SNAP	SIMD Neurocomputer Array Processor.
SNIK	Serial Network Interface Controller of K2.
SOLOMON	Simultaneous Operation Linked Ordinal Modular Network.
SPARC	Scalable Processor Architecture.
SPE	Synaptic processing element.
SPERT	Synthetic Perceptron Testbed.
SPIE	Society of Photo-Optical Instrumentation Engineers.
SPRINT	Systolic Processor with a Reconfigurable Interconnection Network of Transputers.
SYNAPSE	Synthesis of Neural Algorithms on a Parallel Systolic Engine.
TI	Texas Instruments.
TNP	Toroidal Neural Processor.
TU	*Technische Universität.*
UK	United Kingdom.
UMA	Uniformed memory access.
USA	United States of America.
VLIW	Very long instruction word.
VLSI	Very large scale integration.
VM16	Virtual Matrices, 16 Synapses.
WAP	Wavefront array processor.

WESCON	Western Electronic Show and Convention.
WG4	Working Group 4.
WSI	Wafer scale integration.
XOR	Exclusive or.

List of Symbols

$\mathbf{M}^{\mathrm{T}}$	Matrix transposition.
$\mathbf{M}_i$	i^{th} row of a matrix.
$\vec{\mathbf{M}}_j$	j^{th} column of a matrix.
$\mathbf{M}_1 \cdot \mathbf{M}_2$	Matrix multiplication.
$\mathbf{M}_1 \circ \mathbf{M}_2$, $\vec{\mathbf{v}}_1 \circ \vec{\mathbf{v}}_2$	Hadamard product (term-to-term multiplication of matrices and vectors).
$\lvert v \rvert$	Absolute value of a scalar.
$\lvert \vec{\mathbf{v}} \rvert$	Euclidean norm of a vector.
$\lceil v \rceil$	Ceiling function (smallest integer larger or equal to v).
$\lfloor v \rfloor$	Floor function (largest integer smaller or equal to v).
$[v_1, v_2]$	Range of numbers (v_1 and v_2 inclusive).
Δv, $\Delta \vec{\mathbf{v}}$, $\Delta \mathbf{M}$	Variation of a scalar, a vector, and a matrix.
$\Delta(\vec{\mathbf{v}})$	Generalized distance between a vector and a reference.
$\Delta(\vec{\mathbf{v}}_1, \vec{\mathbf{v}}_2)$	Generalized distance between two vectors.
$v_1 \oplus v_2$	Exclusive or (XOR) operation.
$\vec{\mathbf{0}}$	Null vector (vector whose components are all zeroes).
$\vec{\mathbf{1}}$	Diagonal of unity hyper-cube (vector whose components are all ones).
a_v	Scaling factor for floating-point to integer conversion of a value v.
C	Number of connections.
$\vec{\mathbf{d}}$, d_i	Desired output vector of an ANN.
$\mathrm{d}_{\lrcorner}$	Down step function with discontinuity point equal to 0.
$\mathrm{d}_{\urcorner}$	Down step function with discontinuity point equal to 1.
d	Down step function.
$\vec{\mathbf{E}}^h$, E_i^h, $\vec{\mathbf{E}}^v$, E_j^v	Horizontal and vertical enable vector for a systolic array.
E	Set of prototypes forming an epoch.
e	Number of prototypes per epoch (epoch length).

f	Clock frequency.
f_{par}	Fraction of a serial program to be parallelized (in execution time).
$\vec{\mathbf{f}}$, f_i, $\vec{\mathbf{f}}^{[k]}$, $f_i^{[k]}$	Function-of-Y unit's result vector for an ANN and for a layer.
f^{M}	Function-of-Y unit's function.
$h_{i,j}$	Horizontal data stream in the GENES IV systolic array.
$\vec{\mathbf{I}}^h$, I_i^h, $\vec{\mathbf{I}}^v$, I_j^v	Horizontal and vertical input vector of the GENES IV systolic array.
$\mathbf{I}$	Identity matrix ($I_{i,i} = 1$ and $I_{i,j} = 0$ for $i \neq j$).
L	Number of layers of an ANN.
m, m_k	Numbers of neurons of an ANN and of a layer.
m_L	Numbers of output neurons of an ANN.
N	Number of PEs per systolic array's edge.
$N_{\max}$	Largest representable value in a system.
$N_{\min}$	Smallest representable value in a system.
N_{R}	Number of bits of a register or a device R.
$\mathbb{N}$	Set of natural integers.
$\mathbb{N}^*$	Set of natural integers excluding 0.
n	Number of inputs of an ANN.
n_{op}	Number of GENES IV operations required to process a prototype.
n_{PE}	Number of PEs of a parallel computer.
n_{reg}	Number of registers of a PE.
$\vec{\mathbf{O}}^h$, O_i^h, $\vec{\mathbf{O}}^v$, O_i^v	Horizontal and vertical output vector of the GENES IV systolic array.
O	Order function (O-notation).
P	Performance of a neural computer.
$\vec{\mathbf{p}}$, p_i, $\vec{\mathbf{p}}^{[k]}$, $p_i^{[k]}$	Potential vector of an ANN and of a layer.
$\mathbf{Q}$, $Q_{i,j}$	Auxiliary desired output matrix in the Adaline model.
$\mathbf{R}$, $R_{i,j}$	Auxiliary input matrix in the Adaline model.
$\mathbb{R}$	Set of real numbers.
$\mathbb{R}_+$	Set of positive real numbers.
$\mathbb{R}^*$	Set of real numbers excluding 0.
$\mathbb{R}_+^*$	Set of positive real numbers excluding 0.
S	Number of prototypes in a training or testing set.
S_{up}	Speed-up.
s	Prototype number.
T	"Temperature" parameter of the sigmoid function (hyperbolic tangent).
t	Discretized update time (or index).
t	Time.
t_{clk}	Clock-cycle time (i.e., discretized time measured in clock cycles).
t_{mc}	Macro-cycle time (i.e., discretized time measured in macro-cycles).
U	Utilization rate.
$\mathrm{u}_{\lrcorner}$	Up step function with discontinuity point equal to 0.
$\mathrm{u}_{\lrcorner}$	Up step function with discontinuity point equal to 1.
u	Up step function.
$v_{i,j}$	Vertical data stream in the GENES IV systolic array.
$\mathbf{W}$, $W_{i,j}$	Synaptic weight matrix of a systolic array.
$\mathbf{w}$, $w_{i,j}$, $\mathbf{w}^{[k]}$, $w_{i,j}^{[k]}$	Synaptic weight matrix of an ANN and of a layer.

$\vec{\mathbf{X}}, X_j$	Input vector of a systolic array.
$\vec{\mathbf{x}}, x_j$	Input vector of an ANN.
$\vec{\mathbf{Y}}, Y_i$	Output vector of a systolic array.
$\vec{\mathbf{y}}, y_i, \vec{\mathbf{y}}^{[k]}, y_i^{[k]}$	Output vector of an ANN and of a layer.
α	Learning coefficient.
β	Amplitude of the activation function.
β_{left}	Left asymptote of the activation function.
β_{right}	Right asymptote of the activation function.
Γ	Division constant for the sigma unit.
γ	Momentum coefficient.
$\vec{\delta}, \delta_i, \vec{\delta}^{[k]}, \delta_i^{[k]}$	Error signal vector of an ANN and of a layer.
$\vec{\varepsilon}, \varepsilon_i, \vec{\varepsilon}^{[k]}, \varepsilon_i^{[k]}$	Error vector of an ANN and of a layer.
Θ	Threshold in a digital implementation of the activation function.
$\vec{\theta}, \theta_i, \vec{\Theta}^{[k]}, \theta_i^{[k]}$	Threshold vector of an ANN and of a layer.
$\boldsymbol{\lambda}, \lambda_{i,j}$	Neighborhood coefficient matrix in the Kohonen model.
$\vec{\mu}, \mu_i$	Result vector of the minimum operation in the Kohonen model.
$\vec{\nu}, \nu_i$	Neighborhood vector in the Kohonen model.
$\vec{\xi}, \xi_i$	Quadratic error vector of an ANN or output layer.
ρ	Presentation number.
σ	Activation function of a neuron.
σ^{M}	Sigma unit's function.
τ	Discretized iteration time (or index).
$\vec{\upsilon}, \upsilon_i$	Winner vector in the Kohonen model.

Curriculum Vitae

Marc A. Viredaz is a Swiss citizen (place of origin: Crissier, VD and Geneva, GE), born September 14, 1965, in Frankfurt am Main (Germany).

Academic and professional curriculum

1972–1982	Primary and secondary school in Pully and Lausanne.
July 1982	*Certificat d'études secondaires*, with the award of mathematics.
1982–1984	High school in Lausanne.
July 1982	*Certificat de maturité–type C* (*baccalauréat ès sciences*), with the award of mathematics and the award of physics.
1984–1989	Engineering studies in microtechnics at the Swiss Federal Institute of Technology at Lausanne (EPFL).
March 1985	Five weeks in a mechanical workshop at Hasler S.A., Orbe.
1986–1987	Student exchange program at Carnegie-Mellon University (CMU) in Pittsburgh, as an under-graduate student in electrical and computer engineering (ECE).
1987–1988	Two semester project with titles "*Implantation en Prolog d'un algorithm d'apprentissage incrémental et robuste (IRL)*" and "*Instrument aéronautique pour le vol à voile.*"
1988	Graduation project with title "*Analyseur logique à mémoire associative*" (2 months).
January 1989	*Diplôme d'ingénieur en microtechnique* (master equivalent) from the Swiss Federal Institute of Technology at Lausanne (EPFL)

1989–1991 Two and a half years at the Integrated Systems Laboratory (IIS) of the Swiss Federal Institute of Technology at Zurich (ETH), as an architect and hardware designer of the K2 distributed-memory parallel processor, with the Professors M. Annaratone and W. Fichtner.

1991–1994 Two and a half years at the Microcomputing Laboratory (LAMI) of the Swiss Federal Institute of Technology at Lausanne (EPFL), as a Ph.D. student.

International publications

- M. Annaratone, M. Fillo, K. Nakabayashi, and M. Viredaz. *The K2 Parallel Processor: Architecture and Hardware Implementation.* In *Proceedings of the 17th Annual International Symposium on Computer Architecture*, pp. 92–101, Seattle, WA (USA), May 1990. IEEE Computer Society, ACM SIGARCH, IEEE Computer Society Press.
- M. Annaratone, G. zur Bonsen, M. Fillo, M. Halbherr, R. Rühl, P. Steiner, and M. Viredaz. *Architecture, Implementation, and System Software of K2.* In A. Bode (ed.), *Distributed Memory Computing*, vol. 487 of *Lecture Notes in Computer Science*, pp. 473–484. Springer-Verlag, Berlin Heidelberg (D), April 1991. Proceedings of EDMCC2.
- M. Annaratone, G. zur Bonsen, M. Fillo, M. Halbherr, R. Rühl, P. Steiner, and M. Viredaz. *A Overview of the K2 Project. Physics Reports (Review Section of Physics Letters)*, 207(3–5):333–349, September 1991. Proceedings of the 3rd graduate summer course on computational physics: Parallel architectures and applications.
- M. Annaratone, M. Fillo, M. Halbherr, R. Rühl, P. Steiner, and M. Viredaz. *The K2 Distributed Memory Parallel Processor: Architecture, Compiler, and Operating System.* In *Proceedings of SUPERCOMPUTING '91*, pp. 900–909, Albuquerque, NM (USA), November 1991. IEEE Computer Society, ACM SIGARCH, IEEE Computer Society Press.
- M. A. Viredaz, C. Lehmann, F. Blayo, and P. Ienne. *MANTRA: A Multi-Model Neural-Network Computer.* In J. G. Delgado-Frias and W. R. Moore (eds.), *Proceedings of the 3rd International Workshop on VLSI for Neural Networks and Artificial Intelligence*, Oxford (GB), September 1992.
- C. Lehmann, M. Viredaz, and F. Blayo. *A Generic Systolic Array Building Block for Neural Networks with On-Chip Learning. IEEE Transactions on Neural Networks*, 4(3):400–407, May 1993. Special issue on neural network hardware.
- M. A. Viredaz. *MANTRA I: An SIMD Processor Array for Neural Computation.* In P. P. Spies (ed.), *Euro-ARCH'93*, Munich, Informatik aktuell, pp. 99–110. Springer-Verlag, Berlin Heidelberg (D), October 1993.
- P. Ienne and M. A. Viredaz. *GENES IV: A Bit-Serial Processing Element for a Multi-Model Neural-Network Accelerator.* In L. Dadda and B. Wah (eds.), *Proceedings of the International Conference on Application-Specific Array Processors*, pp. 345–356, Venice (I), October 1993. Euromicro, IEEE, IEEE Computer Society Press.
- M. A. Viredaz and P. Ienne. *MANTRA I: A Systolic Neuro-Computer.* In *Proceedings of the International Joint Conference on Neural Networks*, vol. 3, pp. 3054–3057, Nagoya (J), October 1993. IEEE, INNS.

- T. Cornu, P. Ienne, D. Niebur, and M. A. Viredaz. *A Systolic Accelerator for Power System Security Assessment.* In *Proceedings of the International Conference on Intelligent System Application to Power Systems*, Montpellier (F), September 1994. To appear.
- P. Ienne and M. A. Viredaz. *Implementation of Kohonen's Self-Organizing Maps on MANTRA I.* In *Proceedings of the 4th International Conference on Microelectronics for Neural Networks and Fuzzy Systems*, Turin (I), September 1994. To appear.
- P. Ienne and M. A. Viredaz. *Bit-Serial Multipliers and Squarers. IEEE Transactions on Computers*, 1994. To appear.
- P. Ienne and M. A. Viredaz. *GENES IV: A Bit-Serial Processing Element for a Multi-Model Neural-Network Accelerator. Journal of VLSI Signal Processing*, 1994. To appear.

National publications

- M. Annaratone, G. zur Bonsen, M. Fillo, K. Nakabayashi, C. Pommerell, R. Rühl, P. Steiner, and M. Viredaz. *Ein Parallel-Computer mit verteiltem Speicher: Das K2-Projekt. SEV/ASE Bulletin*, 17:11–18, August 1990.
- C. Lehmann, F. Blayo, and M. Viredaz. *MANTRA I–Autonomous Neural-Slice Machine for Kohonen's Self-Organized Feature Maps.* In *Working Group 4 (WG4) Workshop on Massively Parallel Computing*, Leysin (CH), March 1992. COST 229.

Technical reports

- M. A. Viredaz. *The Serial Network Interface Controller (SNIK) of K2: Hardware Description.* Technical report no. 90/16, IIS, ETH, Zurich (CH), October 1990.
- M. A. Viredaz. *Design of Test Routines for the SNIK.* Technical report no. 90/17, IIS, ETH, Zurich (CH), October 1990.
- P. Steiner and M. A. Viredaz. *The Serial Network Interface Controller (SNIK) of K2.* Technical report no. 91/14, IIS, ETH, Zurich (CH), October 1991.
- M. A. Viredaz. *The Mass Storage Controller (MSC) of K2.* Technical report no. 91/15, IIS, ETH, Zurich (CH), December 1991.
- M. A. Viredaz. *Design Specifications for the Backplane of K2.* Technical report no. 91/16, IIS, ETH, Zurich (CH), October 1991.
- M. A. Viredaz, C. Lehmann, F. Blayo, and P. Ienne. *MANTRA: A Multi-Model Neural-Network Computer.* Technical report no. 92/6, DI, EPFL, Lausanne (CH), July 1992.
- P. Ienne and M. A. Viredaz. *Bit-Serial Input and Output Multipliers and Squarers.* Technical report no. 92/7, DI, EPFL, Lausanne (CH), October 1992.
- M. A. Viredaz. *The MANTRA I Prototype Machine: Hardware Description.* MANTRA internal report no. 93/2, EPFL, Lausanne (CH), September 1993.