FPGA IMPLEMENTATIONS OF NEURAL NETWORKS
FPGA Implementations of Neural Networks

Edited by

AMOS R. OMONDI
Flinders University, Adelaide, SA, Australia

and

JAGATH C. RAJAPAKSE
Nanyang Technological University, Singapore
Contents

Preface ix

1 FPGA Neurocomputers 1
Amos R. Omondi, Jagath C. Rajapakse and Mariusz Bajger

1.1 Introduction 1
1.2 Review of neural-network basics 3
1.3 ASIC vs. FPGA neurocomputers 9
1.4 Parallelism in neural networks 12
1.5 Xilinx Virtex-4 FPGA 13
1.6 Arithmetic 15
1.7 Activation-function implementation: unipolar sigmoid 21
1.8 Performance evaluation 32
1.9 Conclusions 34
References 34

2 Arithmetic precision for implementing BP networks on FPGA: A case study 37
Medhat Moussa and Shawki Areibi and Kristian Nichols

2.1 Introduction 37
2.2 Background 39
2.3 Architecture design and implementation 43
2.4 Experiments using logical-XOR problem 48
2.5 Results and discussion 50
2.6 Conclusions 55
References 56

3 FPNA: Concepts and properties 63
Bernard Girau

3.1 Introduction 63
3.2 Choosing FPGAs 65
3.3 FPNA, FPNN 71
3.4 Correctness 86
3.5 Underparameterized convolutions by FPNNs 88
3.6 Conclusions 96
References 97
4
FPNA: Applications and implementations

Bernard Girau

4.1. Summary of Chapter 3 104
4.2. Towards simplified architectures: symmetric boolean functions by FPNAs 105
4.3. Benchmark applications 109
4.4. Other applications 113
4.5. General FPGA implementation 116
4.6. Synchronous FPNNs 120
4.7. Implementations of synchronous FPNNs 124
4.8. Implementation performances 130
4.9. Conclusions 133
References 134

5
Back-Propagation Algorithm Achieving 5 GOPS on the Virtex-E

Kolin Paul and Sanjay Rajopadhye

5.1. Introduction 138
5.2. Problem specification 139
5.3. Systolic implementation of matrix-vector multiply 141
5.4. Pipelined back-propagation architecture 142
5.5. Implementation 144
5.6. MMALPHA design environment 147
5.7. Architecture derivation 149
5.8. Hardware generation 155
5.9. Performance evaluation 157
5.10. Related work 159
5.11. Conclusion 160
Appendix 161
References 163

6
FPGA Implementation of Very Large Associative Memories

Dan Hammerstrom, Changjian Gao, Shaojuan Zhu, Mike Butts

6.1. Introduction 167
6.2. Associative memory 168
6.3. PC Performance Evaluation 179
6.4. FPGA Implementation 184
6.5. Performance comparisons 190
6.6. Summary and conclusions 192
References 193

7
FPGA Implementations of Neocognitrons

Alessandro Noriaki Ide and José Hiroki Saito

7.1. Introduction 197
7.2. Neocognitron 198
7.3. Alternative neocognitron 201
7.4. Reconfigurable computer 205
7.5. Reconfigurable orthogonal memory multiprocessor 206
Contents

7.6. Alternative neocognitron hardware implementation 209
7.7. Performance analysis 215
7.8. Applications 218
7.9. Conclusions 221
References 222

8
Self Organizing Feature Map for Color Quantization on FPGA 225
Chip-Hong Chang, Menon Shibu and Rui Xiao
8.1. Introduction 225
8.2. Algorithmic adjustment 228
8.3. Architecture 231
8.4. Implementation 235
8.5. Experimental results 239
8.6. Conclusions 242
References 242

9
Implementation of Self-Organizing Feature Maps in Reconfigurable Hardware
Mario Porrmann, Ulf Witkowski, and Ulrich Rückert
9.1. Introduction 247
9.2. Using reconfigurable hardware for neural networks 248
9.3. The dynamically reconfigurable rapid prototyping system RAPTOR2000 250
9.4. Implementing self-organizing feature maps on RAPTOR2000 252
9.5. Conclusions 267
References 267

10
FPGA Implementation of a Fully and Partially Connected MLP 271
Antonio Canas, Eva M. Ortigosa, Eduardo Ros and Pilar M. Ortigosa
10.1. Introduction 271
10.2. MLP/XMLP and speech recognition 273
10.3. Activation functions and discretization problem 276
10.4. Hardware implementations of MLP 284
10.5. Hardware implementations of XMLP 291
10.6. Conclusions 293
Acknowledgments 294
References 295

11
FPGA Implementation of Non-Linear Predictors 297
Rafael Gadea-Girones and Agustín Ramírez-Agundis
11.1. Introduction 298
11.2. Pipeline and back-propagation algorithm 299
11.3. Synthesis and FPGAs 304
11.4. Implementation on FPGA 313
11.5. Conclusions 319
References 321
# The REMAP reconfigurable architecture: a retrospective

*Lars Bengtsson, Arne Linde, Tomas Nordström, Bertil Svensson, and Mikael Taveniku*

<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>12.1. Introduction</td>
<td>326</td>
</tr>
<tr>
<td>12.2. Target Application Area</td>
<td>327</td>
</tr>
<tr>
<td>12.3. REMAP-β – design and implementation</td>
<td>335</td>
</tr>
<tr>
<td>12.4. Neural networks mapped on REMAP-β</td>
<td>346</td>
</tr>
<tr>
<td>12.5. REMAP-γ architecture</td>
<td>353</td>
</tr>
<tr>
<td>12.6. Discussion</td>
<td>354</td>
</tr>
<tr>
<td>12.7. Conclusions</td>
<td>357</td>
</tr>
<tr>
<td>Acknowledgments</td>
<td>357</td>
</tr>
<tr>
<td>References</td>
<td>357</td>
</tr>
</tbody>
</table>
During the 1980s and early 1990s there was significant work in the design and implementation of hardware neurocomputers. Nevertheless, most of these efforts may be judged to have been unsuccessful: at no time have hardware neurocomputers been in wide use. This lack of success may be largely attributed to the fact that earlier work was almost entirely aimed at developing custom neurocomputers, based on ASIC technology, but for such niche areas this technology was never sufficiently developed or competitive enough to justify large-scale adoption. On the other hand, gate-arrays of the period mentioned were never large enough nor fast enough for serious artificial-neural-network (ANN) applications. But technology has now improved: the capacity and performance of current FPGAs are such that they present a much more realistic alternative. Consequently neurocomputers based on FPGAs are now a much more practical proposition than they have been in the past. This book summarizes some work towards this goal and consists of 12 papers that were selected, after review, from a number of submissions. The book is nominally divided into three parts: Chapters 1 through 4 deal with foundational issues; Chapters 5 through 11 deal with a variety of implementations; and Chapter 12 looks at the lessons learned from a large-scale project and also reconsiders design issues in light of current and future technology.

Chapter 1 reviews the basics of artificial-neural-network theory, discusses various aspects of the hardware implementation of neural networks (in both ASIC and FPGA technologies, with a focus on special features of artificial neural networks), and concludes with a brief note on performance-evaluation. Special points are the exploitation of the parallelism inherent in neural networks and the appropriate implementation of arithmetic functions, especially the sigmoid function. With respect to the sigmoid function, the chapter includes a significant contribution.

Certain sequences of arithmetic operations form the core of neural-network computations, and the second chapter deals with a foundational issue: how to determine the numerical precision format that allows an optimum tradeoff between precision and implementation (cost and performance). Standard single or double precision floating-point representations minimize quantization
errors while requiring significant hardware resources. Less precise fixed-point representation may require less hardware resources but add quantization errors that may prevent learning from taking place, especially in regression problems. Chapter 2 examines this issue and reports on a recent experiment where we implemented a multi-layer perceptron on an FPGA using both fixed and floating point precision.

A basic problem in all forms of parallel computing is how best to map applications onto hardware. In the case of FPGAs the difficulty is aggravated by the relatively rigid interconnection structures of the basic computing cells. Chapters 3 and 4 consider this problem: an appropriate theoretical and practical framework to reconcile simple hardware topologies with complex neural architectures is discussed. The basic concept is that of Field Programmable Neural Arrays (FPNA) that lead to powerful neural architectures that are easy to map onto FPGAs, by means of a simplified topology and an original data exchange scheme. Chapter 3 gives the basic definition and results of the theoretical framework. And Chapter 4 shows how FPNA lead to powerful neural architectures that are easy to map onto digital hardware. Applications and implementations are described, focusing on a class.

Chapter 5 presents a systolic architecture for the complete back propagation algorithm. This is the first such implementation of the back propagation algorithm which completely parallelizes the entire computation of learning phase. The array has been implemented on an Annapolis FPGA based coprocessor and it achieves very favorable performance with range of 5 GOPS. The proposed new design targets Virtex boards. A description is given of the process of automatically deriving these high performance architectures using the systolic array design tool MMAlpha, facilitates system specification. This makes it easy to specify the system in a very high level language (Alpha) and also allows perform design exploration to obtain architectures whose performance is comparable to that obtained using hand optimized VHDL code.

Associative networks have a number of properties, including a rapid, compute efficient best-match and intrinsic fault tolerance, that make them ideal for many applications. However, large networks can be slow to emulate because of their storage and bandwidth requirements. Chapter 6 presents a simple but effective model of association and then discusses a performance analysis of the implementation this model on a single high-end PC workstation, a PC cluster, and FPGA hardware.

Chapter 7 describes the implementation of an artificial neural network in a reconfigurable parallel computer architecture using FPGA’s, named Reconfigurable Orthogonal Memory Multiprocessor (REOMP), which uses $p^2$ memory modules connected to $p$ reconfigurable processors, in row access mode, and column access mode. REOMP is considered as an alternative model of the neural network neocognitron. The chapter consists of a description of the RE-
OMP architecture, a the case study of alternative neocognitron mapping, and a performance performance analysis with systems systems consisting of 1 to 64 processors.

Chapter 8 presents an efficient architecture of Kohonen Self-Organizing Feature Map (SOFM) based on a new Frequency Adaptive Learning (FAL) algorithm which efficiently replaces the neighborhood adaptation function of the conventional SOFM. The proposed SOFM architecture is prototyped on Xilinx Virtex FPGA using the prototyping environment provided by XESS. A robust functional verification environment is developed for rapid prototype development. Various experimental results are given for the quantization of a 512 X 512 pixel color image.

Chapter 9 consists of another discussion of an implementation of SOFMs in reconfigurable hardware. Based on the universal rapid prototyping system, RAPTOR2000, a hardware accelerator for self-organizing feature maps has been developed. Using Xilinx Virtex-E FPGAs, RAPTOR2000 is capable of emulating hardware implementations with a complexity of more than 15 million system gates. RAPTOR2000 is linked to its host – a standard personal computer or workstation – via the PCI bus. A speed-up of up to 190 is achieved with five FPGA modules on the RAPTOR2000 system compared to a software implementation on a state of the art personal computer for typical applications of SOFMs.

Chapter 10 presents several hardware implementations of a standard Multi-Layer Perceptron (MLP) and a modified version called eXtended Multi-Layer Perceptron (XMLP). This extended version is an MLP-like feed-forward network with two-dimensional layers and configurable connection pathways. The discussion includes a description of hardware implementations have been developed and tested on an FPGA prototyping board and includes systems specifications using two different abstraction levels: register transfer level (VHDL) and a higher algorithmic-like level (Handel-C) as well as the exploitation of varying degrees of parallelism. The main test bed application is speech recognition.

Chapter 11 describes the implementation of a systolic array for a non-linear predictor for image and video compression. The implementation is based on a multilayer perceptron with a hardware-friendly learning algorithm. It is shown that even with relatively modest FPGA devices, the architecture attains the speeds necessary for real-time training in video applications and enabling more typical applications to be added to the image compression processing.

The final chapter consists of a retrospective look at the REMAP project, which was the construction of design, implementation, and use of large-scale parallel architectures for neural-network applications. The chapter gives an overview of the computational requirements found in algorithms in general and motivates the use of regular processor arrays for the efficient execution of such
algorithms. The architecture, following the SIMD principle (Single Instruction stream, Multiple Data streams), is described, as well as the mapping of some important and representative ANN algorithms. Implemented in FPGA, the system served as an architecture laboratory. Variations of the architecture are discussed, as well as scalability of fully synchronous SIMD architectures. The design principles of a VLSI-implemented successor of REMAP-β are described, and the paper concludes with a discussion of how the more powerful FPGA circuits of today could be used in a similar architecture.

Amos R. Omondi and Jagath C. Rajapakse
Chapter 1

FPGA NEUROCOMPUTERS

Amos R. Omondi  
*School of Informatics and Engineering, Flinders University, Bedford Park, SA 5042, Australia*  
Amos.Omondi@flinders.edu.au

Jagath C. Rajapakse  
*School Computer Engineering, Nanyang Technological University, Singapore 639798*  
asrajapakse@ntu.edu.sg

Mariusz Bajger  
*School of Informatics and Engineering, Flinders University, Bedford Park, SA 5042, Australia*  
Mariusz.Bajger@flinders.edu.au

Abstract  

This introductory chapter reviews the basics of artificial-neural-network theory, discusses various aspects of the hardware implementation of neural networks (in both ASIC and FPGA technologies, with a focus on special features of artificial neural networks), and concludes with a brief note on performance-evaluation. Special points are the exploitation of the parallelism inherent in neural networks and the appropriate implementation of arithmetic functions, especially the sigmoid function. With respect to the sigmoid function, the chapter includes a significant contribution.

Keywords: FPGAs, neurocomputers, neural-network arithmetic, sigmoid, performance-evaluation.

1.1 Introduction  

In the 1980s and early 1990s, a great deal of research effort (both industrial and academic) was expended on the design and implementation of hardware neurocomputers [5, 6, 7, 8]. But, on the whole, most efforts may be judged
to have been unsuccessful: at no time have have hardware neurocomputers been in wide use; indeed, the entire field was largely moribund by the end the 1990s. This lack of success may be largely attributed to the fact that earlier work was almost entirely based on ASIC technology but was never sufficiently developed or competitive enough to justify large-scale adoption; gate-arrays of the period mentioned were never large enough nor fast enough for serious neural-network applications. Nevertheless, the current literature shows that ASIC neurocomputers appear to be making some sort of a comeback [1, 2, 3]; we shall argue below that these efforts are destined to fail for exactly the same reasons that earlier ones did. On the other hand, the capacity and performance of current FPGAs are such that they present a much more realistic alternative. We shall in what follows give more detailed arguments to support these claims.

The chapter is organized as follows. Section 2 is a review of the fundamentals of neural networks; still, it is expected that most readers of the book will already be familiar with these. Section 3 briefly contrasts ASIC-neurocomputers with FPGA-neurocomputers, with the aim of presenting a clear case for the former; a more significant aspects of this argument will be found in [18]. One of the most repeated arguments for implementing neural networks in hardware is the parallelism that the underlying models possess. Section 4 is a short section that reviews this. In Section 5 we briefly describe the realization of a state-of-the-art FPGA device. The objective there is to be able to put into a concrete context certain following discussions and to be able to give grounded discussions of what can or cannot be achieved with current FPGAs. Section 6 deals with certain aspects of computer arithmetic that are relevant to neural-network implementations. Much of this is straightforward, and our main aim is to highlight certain subtle aspects. Section 7 nominally deals with activation functions, but is actually mostly devoted to the sigmoid function. There are two main reasons for this choice: first, the chapter contains a significant contribution to the implementation of elementary or near-elementary activation functions, the nature of which contribution is not limited to the sigmoid function; second, the sigmoid function is the most important activation function for neural networks. In Section 8, we very briefly address an important issue — performance evaluation. Our goal here is simple and can be stated quite succintly: as far as performance-evaluation goes, neurocomputer architecture continues to languish in the “Dark Ages”, and this needs to change. A final section summarises the main points made in chapter and also serves as a brief introduction to subsequent chapters in the book.

1Unless otherwise indicated, we shall use neural network to mean artificial neural network.
1.2 Review of neural-network basics

The human brain, which consists of approximately 100 billion neurons that are connected by about 100 trillion connections, forms the most complex object known in the universe. Brain functions such as sensory information processing and cognition are the results of emergent computations carried out by this massive neural network. Artificial neural networks are computational models that are inspired by the principles of computations performed by the biological neural networks of the brain. Neural networks possess many attractive characteristics that may ultimately surpass some of the limitations in classical computational systems. The processing in the brain is mainly parallel and distributed: the information are stored in connections, mostly in myeline layers of axons of neurons, and, hence, distributed over the network and processed in a large number of neurons in parallel. The brain is adaptive from its birth to its complete death and learns from exemplars as they arise in the external world. Neural networks have the ability to learn the rules describing training data and, from previously learnt information, respond to novel patterns. Neural networks are fault-tolerant, in the sense that the loss of a few neurons or connections does not significantly affect their behavior, as the information processing involves a large number of neurons and connections. Artificial neural networks have found applications in many domains — for example, signal processing, image analysis, medical diagnosis systems, and financial forecasting.

The roles of neural networks in the afore-mentioned applications fall broadly into two classes: pattern recognition and functional approximation. The fundamental objective of pattern recognition is to provide a meaningful categorization of input patterns. In functional approximation, given a set of patterns, the network finds a smooth function that approximates the actual mapping between the input and output.

A vast majority of neural networks are still implemented on software on sequential machines. Although this is not necessarily always a severe limitation, there is much to be gained from directly implementing neural networks in hardware, especially if such implementation exploits the parellelism inherent in the neural networks but without undue costs. In what follows, we shall describe a few neural network models — multi-layer perceptrons, Kohonen’s self-organizing feature map, and associative memory networks — whose implementations on FPGA are discussed in the other chapters of the book.

1.2.1 Artificial neuron

An artificial neuron forms the basic unit of artificial neural networks. The basic elements of an artificial neurons are (1) a set of input nodes, indexed by, say, 1, 2, ... \( I \), that receives the corresponding input signal or pattern vector, say \( \mathbf{x} = (x_1, x_2, \ldots, x_I)^T \); (2) a set of synaptic connections whose strengths are
represented by a set of weights, here denoted by \( w = (w_1, w_2, \ldots, w_I)^T \); and (3) an activation function \( \Phi \) that relates the total synaptic input to the output (activation) of the neuron. The main components of an artificial neuron is illustrated in Figure 1.

Figure 1: The basic components of an artificial neuron

The total synaptic input, \( u \), to the neuron is given by the inner product of the input and weight vectors:

\[
u = \sum_{i=1}^{I} w_i x_i \tag{1.1}\]

where we assume that the threshold of the activation is incorporated in the weight vector. The output activation, \( y \), is given by

\[
y = \Phi(u) \tag{1.2}\]

where \( \Phi \) denotes the activation function of the neuron. Consequently, the computation of the inner-products is one of the most important arithmetic operations to be carried out for a hardware implementation of a neural network. This means not just the individual multiplications and additions, but also the alternation of successive multiplications and additions — in other words, a sequence of multiply-add (also commonly known as multiply-accumulate or MAC) operations. We shall see that current FPGA devices are particularly well-suited to such computations.

The total synaptic input is transformed to the output via the non-linear activation function. Commonly employed activation functions for neurons are
the threshold activation function (unit step function or hard limiter):

\[ \Phi(u) = \begin{cases} 1.0, & \text{when } u > 0, \\ 0.0, & \text{otherwise.} \end{cases} \]

the ramp activation function:

\[ \Phi(u) = \max\{0.0, \min\{1.0, u + 0.5\}\} \]

the sigmoidal activation function, where the unipolar sigmoid function is

\[ \Phi(u) = \frac{a}{1 + \exp(-bu)} \]

and the bipolar sigmoid is

\[ \Phi(u) = a \left( \frac{1 - \exp(-bu)}{1 + \exp(-bu)} \right) \]

where \( a \) and \( b \) represent, respectively, real constants the gain or amplitude and the slope of the transfer function.

The second most important arithmetic operation required for neural networks is the computation of such activation functions. We shall see below that the structure of FPGAs limits the ways in which these operations can be carried out at reasonable cost, but current FPGAs are also equipped to enable high-speed implementations of these functions if the right choices are made.

A neuron with a threshold activation function is usually referred to as the discrete perceptron, and with a continuous activation function, usually a sigmoidal function, such a neuron is referred to as continuous perceptron. The sigmoidal is the most pervasive and biologically plausible activation function.

Neural networks attain their operating characteristics through learning or training. During training, the weights (or strengths) of connections are gradually adjusted in either supervised or unsupervised manner. In supervised learning, for each training input pattern, the network is presented with the desired output (or a teacher), whereas in unsupervised learning, for each training input pattern, the network adjusts the weights without knowing the correct target. The network self-organizes to classify similar input patterns into clusters in unsupervised learning. The learning of a continuous perceptron is by adjustment (using a gradient-descent procedure) of the weight vector, through the minimization of some error function, usually the square-error between the desired output and the output of the neuron. The resultant learning is known as

\[ \text{In general, the slope of the ramp may be other than unity.} \]
as delta learning: the new weight-vector, \( \mathbf{w}^{\text{new}} \), after presentation of an input \( \mathbf{x} \) and a desired output \( d \) is given by

\[
\mathbf{w}^{\text{new}} = \mathbf{w}^{\text{old}} + \alpha \delta \mathbf{x}
\]

where \( \mathbf{w}^{\text{old}} \) refers to the weight vector before the presentation of the input and the error term, \( \delta \), is \((d - y)\Phi'(u)\), where \( y \) is as defined in Equation 1.2 and \( \Phi' \) is the first derivative of \( \Phi \). The constant \( \alpha \), where \( 0 < \alpha \leq 1 \), denotes the learning factor. Given a set of training data, \( \Gamma = \{ (x_i, d_i); i = 1, \ldots n \} \), the complete procedure of training a continuous perceptron is as follows:

**begin:** /* training a continuous perceptron */
Initialize weights \( \mathbf{w}^{\text{new}} \)
Repeat
For each pattern \((x_i, d_i)\) do
\[
\mathbf{w}^{\text{old}} = \mathbf{w}^{\text{new}}
\]
\[
\mathbf{w}^{\text{new}} = \mathbf{w}^{\text{old}} + \alpha \delta \mathbf{x}_i
\]
until convergence
**end**

The weights of the perceptron are initialized to random values, and the convergence of the above algorithm is assumed to have been achieved when no more significant changes occur in the weight vector.

### 1.2.2 Multi-layer perceptron

The multi-layer perceptron (MLP) is a feedforward neural network consisting of an input layer of nodes, followed by two or more layers of perceptrons, the last of which is the output layer. The layers between the input layer and output layer are referred to as hidden layers. MLPs have been successfully applied to many complex real-world problems consisting of non-linear decision boundaries. Three-layer MLPs have been sufficient for most of these applications. In what follows, we will briefly describe the architecture and learning of an L-layer MLP.

Let 0-layer and L-layer represent the input and output layers, respectively; and let \( w_{kj}^{l+1} \) denote the synaptic weight connected to the \( k \)-th neuron of the \( l + 1 \) layer from the \( j \)-th neuron of the \( l \)-th layer. If the number of perceptrons in the \( l \)-th layer is \( N_l \), then we shall let \( \mathbf{W}_l \triangleq \{ w_{kj}^{l+1} \}_{N_l \times N_{l-1}} \) denote the matrix of weights connecting to \( l \)-th layer. The vector of synaptic inputs to the \( l \)-th layer, \( \mathbf{u}_l = (u_{l,1}^1, u_{l,2}^1, \ldots u_{l,N_l}^1)\) is given by

\[
\mathbf{u}_l = \mathbf{W}_l \mathbf{y}_{l-1},
\]

where \( \mathbf{y}_{l-1} = (y_{1}^{l-1}, y_{2}^{l-1}, \ldots y_{N_{l-1}}^{l-1})\) denotes the vector of outputs at the \( l-1 \) layer. The generalized delta learning-rule for the layer \( l \) is, for perceptrons,
given by
\[ W_l^{\text{new}} = W_l^{\text{old}} + \alpha \delta_l y_{l-1}^T, \]
where the vector of error terms, \( \delta_l^T = (\delta_{l1}^l, \delta_{l2}^l, \ldots, \delta_{lN_l}^l) \) at the \( l \) th layer is given by
\[
\delta_j^l = \begin{cases} 
2\Phi_{lj}'\left(u_{lj}^l\right)(d_j - o_j), & \text{when } l = L, \\
\Phi_{lj}'\left(u_{lj}^l\right) \sum_{k=1}^{N_l+1} \delta_{l+1}^k w_{lj}^{l+1}, & \text{otherwise},
\end{cases}
\]
where \( o_j \) and \( d_j \) denote the network and desired outputs of the \( j \)-th output neuron, respectively; and \( \Phi_{lj} \) and \( u_{lj}^l \) denote the activation function and total synaptic input to the \( j \)-th neuron at the \( l \)-th layer, respectively. During training, the activities propagate forward for an input pattern; the error terms of a particular layer are computed by using the error terms in the next layer and, hence, move in the backward direction. So, the training of MLP is referred as error back-propagation algorithm. For the rest of this chapter, we shall generally focus on MLP networks with backpropagation, this being, arguably, the most-implemented type of artificial neural networks.

**Figure 2:** Architecture of a 3-layer MLP network
1.2.3 Self-organizing feature maps

Neurons in the cortex of the human brain are organized into layers of neurons. These neurons not only have bottom-up and top-down connections, but also have lateral connections. A neuron in a layer excites its closest neighbors via lateral connections but inhibits the distant neighbors. Lateral interactions allow neighbors to partially learn the information learned by a winner (formally defined below), which gives neighbors responding to similar patterns after learning with the winner. This results in topological ordering of formed clusters. The self-organizing feature map (SOFM) is a two-layer self-organizing network which is capable of learning input patterns in a topologically ordered manner at the output layer. The most significant concept in a learning SOFM is that of learning within a neighbourhood around a winning neuron. Therefore not only the weights of the winner but also those of the neighbors of the winner change.

The winning neuron, \( m \), for an input pattern \( x \) is chosen according to the total synaptic input:

\[
m = \arg \max_j w_j^T x,
\]

where \( w_j \) denotes the weight-vector corresponding to the \( j \)-th output neuron. \( w_m^T x \) determines the neuron with the shortest Euclidean distance between its weight vector and the input vector when the input patterns are normalized to unity before training.

Let \( \mathcal{N}_m(t) \) denote a set of indices corresponding to the neighbourhood size of the current winner \( m \) at the training time or iteration \( t \). The radius of \( \mathcal{N}_m \) is decreased as the training progresses; that is, \( \mathcal{N}_m(t_1) > \mathcal{N}_m(t_2) > \mathcal{N}_m(t_3) \ldots \), where \( t_1 < t_2 < t_3 \ldots \). The radius \( \mathcal{N}_m(t = 0) \) can be very large at the beginning of learning because it is needed for initial global ordering of weights, but near the end of training, the neighbourhood may involve no neighbouring neurons other than the winning one. The weights associated with the winner and its neighbouring neurons are updated by

\[
\Delta w_j = \alpha(j, t) (x - w_j) \quad \text{for all} \quad j \in \mathcal{N}_m(t),
\]

where the positive learning factor depends on both the training time and the size of the neighbourhood. For example, a commonly used neighbourhood function is the Gaussian function

\[
\alpha(\mathcal{N}_m(t), t) = \alpha(t) \exp \left( -\frac{\|r_j - r_m\|^2}{2\sigma^2(t)} \right),
\]

where \( r_m \) and \( r_j \) denote the positions of the winning neuron \( m \) and of the winning neighbourhood neurons \( j \), respectively. \( \alpha(t) \) is usually reduced at a
rate that is inversely proportional to $t$. The type of training described above is known as Kohonen’s algorithm (for SOFMs). The weights generated by the above algorithms are arranged spatially in an ordering that is related to the features of the trained patterns. Therefore, the algorithm produces topology-preserving maps. After learning, each input causes a localized response with positions on the output layer that reflects dominant features of the input.

### 1.2.4 Associative-memory networks

Associative memory networks are two-layer networks in which weights are determined in order to store a set of pattern associations, say, \{(s_1, t_1), (s_2, t_2), \ldots (s_k, t_k), \ldots (s_n, t_n)\}, where input pattern $s_k$ is associated with output pattern $t_k$. These networks not only learn to produce associative patterns, but also are able to recall the desired response patterns when a given pattern is similar to the stored pattern. Therefore they are referred to as content-addressible memory. For each association vector $(s_k, t_k)$, if $s_k = t_k$, the network is referred to as auto-associative; otherwise it is hetero-associative. The networks often provide input-output descriptions of the associative memory through a linear transformation (then known as linear associative memory). The neurons in these networks have linear activation functions. If the linearity constant is unity, then the output layer activation is given by

$$y = Wx,$$

where $W$ denotes the weight matrix connecting the input and output layers. These networks learn using the Hebb rule; the weight matrix to learn all the associations is given by the batch learning rule:

$$W = \sum_{k=1}^{n} t_k s_k^T.$$

If the stored patterns are orthogonal to each other, then it can be shown that the network recalls the stored associations perfectly. Otherwise, the recalled patterns are distorted by cross-talk between patterns.

### 1.3 ASIC vs. FPGA neurocomputers

By far, the most often-stated reason for the development of custom (i.e. ASIC) neurocomputers is that conventional (i.e. sequential) general-purpose processors do not fully exploit the parallelism inherent in neural-network models and that highly parallel architectures are required for that. That is true as far as it goes, which is not very far, since it is mistaken on two counts [18]: The first is that it confuses the final goal, which is high performance — not merely parallelism — with artifacts of the basic model. The strong focus on
parallelism can be justified only when high performance is attained at a reasonable cost. The second is that such claims ignore the fact that conventional microprocessors, as well as other types of processors with a substantial user-base, improve at a much faster rate than (low-use) special-purpose ones, which implies that the performance (relative to cost or otherwise) of ASIC neurocomputers will always lag behind that of mass-produced devices – even on special applications. As an example of this misdirection of effort, consider the latest in ASIC neurocomputers, as exemplified by, say, [3]. It is claimed that “with relatively few neurons, this ANN-dedicated hardware chip [Neuricam Totem] outperformed the other two implementations [a Pentium-based PC and a Texas Instruments DSP]”. The actual results as presented and analysed are typical of the poor benchmarking that afflicts the neural-network area. We shall have more to say below on that point, but even if one accepts the claims as given, some remarks can be made immediately. The strongest performance-claim made in [3], for example, is that the Totem neurochip outperformed, by a factor of about 3, a PC (with a 400-MHz Pentium II processor, 128 Mbytes of main memory, and the neural netwoks implemented in Matlab). Two points are pertinent here:

- In late-2001/early 2002, the latest Pentiums had clock rates that were more than 3 times that of Pentium II above and with much more memory (cache, main, etc.) as well.

- The PC implementation was done on top of a substantial software (base), instead of a direct low-level implementation, thus raising issues of “best-effort” with respect to the competitor machines.

A comparison of the NeuriCam Totems and Intel Pentiums, in the years 2002 and 2004 will show the large basic differences have only got larger, primarily because, with the much large user-base, the Intel (x86) processors continue to improve rapidly, whereas little is ever heard of about the neurocomputers as PCs go from one generation to another.

So, where then do FGPAs fit in? It is evident that in general FPGAs cannot match ASIC processors in performance, and in this regard FPGAs have always lagged behind conventional microprocessors. Nevertheless, if one considers FPGA structures as an alternative to software on, say, a general-purpose processor, then it is possible that FPGAs may be able to deliver better cost:performance ratios on given applications.\(^3\) Moreover, the capacity for reconfiguration means that may be extended to a range of applications, e.g. several different types of neural networks. Thus the main advantage of the FPGA is that it may offer a better cost:performance ratio than either custom

---

\(^3\)Note that the issue is cost:performance and not just performance
ASIC vs. FPGA neurocomputers

ASIC neurocomputers or state-of-the-art general-purpose processors and with more flexibility than the former. A comparison of the NeuriCam Totem, Intel Pentiums, and M FPGAs will also show that improvements that show the advantages of the FPGAs, as a consequence of relatively rapid changes in density and speed.

It is important to note here two critical points in relation to custom (ASIC) neurocomputers versus the FPGA structures that may be used to implement a variety of artificial neural networks. The first is that if one aims to realize a custom neurocomputer that has a significant amount of flexibility, then one ends up with a structure that resembles an FPGA — that is, a small number of different types functional units that can be configured in different ways, according to the neural network to be implemented — but which nonetheless does not have the same flexibility. (A particular aspect to note here is that the large variety of neural networks — usually geared towards different applications — gives rise to a requirement for flexibility, in the form of either programmability or reconfigurability.) The second point is that raw hardware-performance alone does not constitute the entirety of a typical computing structure: software is also required; but the development of software for custom neurocomputers will, because of the limited user-base, always lag behind that of the more widely used FPGAs. A final drawback of the custom-neurocomputer approach is that most designs and implementations tend to concentrate on just the high parallelism of the neural networks and generally ignore the implications of Amdahl’s Law, which states that ultimately the speed-up will be limited by any serial or lowly-parallel processing involved. (One rare exception is [8].) Thus non-neural and other serial parts of processing tend to be given short shrift. Further, even where parallelism can be exploited, most neurocomputer-design seem to take little account of the fact that the degrees of useful parallelism will vary according to particular applications. (If parallelism is the main issue, then all this would suggest that the ideal building block for an appropriate parallel-processor machine is one that is less susceptible to these factors, and this argues for a relatively large-grain high-performance processor, used in smaller numbers, that can nevertheless exploit some of the parallelism inherent in neural networks [18].)

All of the above can be summed up quite succinctly: despite all the claims that have been made and are still being made, to date there has not been a custom neurocomputer that, on artificial neural-network problems (or, for that matter, on any other type of problem), has outperformed the best conventional computer of its time. Moreover, there is little chance of that happening. The

---

4Although not quite successful as a neurocomputer, this machine managed to survive longer than most neurocomputers — because the flexibility inherent in its design meant that it could also be useful for non-neural applications.
FPGA Neurocomputers

promise of FPGAs is that they offer, in essence, the ability to realize “semi-
custom” machines for neural networks; and, with continuing developments in
technology, they thus offer the best hope for changing the situation, as far as
possibly outperforming (relative to cost) conventional processors.

1.4 Parallelism in neural networks

Neural networks exhibit several types of parallelism, and a careful examina-
tion of these is required in order to both determine the most suitable hardware
structures as well as the best mappings from the neural-network structures onto
given hardware structures. For example, parallelism can be of the SIMD type
or of the MIMD type, bit-parallel or word-parallel, and so forth [5]. In general,
the only categorical statement that can be made is that, except for networks of a
trivial size, fully parallel implementation in hardware is not feasible — virtual
parallelism is necessary, and this, in turn, implies some sequential processing.
In the context of FPGa, it might appear that reconfiguration is a silver bullet,
but this is not so: the benefits of dynamic reconfigurability must be evaluated
relative to the costs (especially in time) of reconfiguration. Nevertheless, there
is little doubt that FPGAs are more promising than ASIC neurocomputers. The
specific types of parallelism are as follows.

- **Training parallelism**: Different training sessions can be run in parallel,
e.g. on SIMD or MIMD processors. The level of parallelism at this level
is usually medium (i.e. in the hundreds), and hence can be nearly fully
mapped onto current large FPGAs.

- **Layer parallelism**: In a multilayer network, different layers can be
processed in parallel. Parallelism at this level is typically low (in the
tens), and therefore of limited value, but it can still be exploited through
pipelining.

- **Node parallelism**: This level, which corresponds to individual neurons, is
perhaps the most important level of parallelism, in that if fully exploited,
then parallelism at all of the above higher levels is also fully exploited.
But that may not be possible, since the number of neurons can be as
high as in the millions. Nevertheless, node parallelism matches FPGAs
very well, since a typical FPGA basically consists of a large number of
“cells” that can operate in parallel and, as we shall see below, onto which
neurons can readily be mapped.

- **Weight parallelism**: In the computation of an output

\[ y = \Phi \left( \sum_{i=1}^{n} w_i x_i \right), \]
where \( x_i \) is an input and \( w_i \) is a weight, the products \( x_i w_i \) can all be computed in parallel, and the sum of these products can also be computed with high parallelism (e.g. by using an adder-tree of logarithmic depth).

- **Bit-level parallelism**: At the implementation level, a wide variety of parallelism is available, depending on the design of individual functional units. For example, bit-serial, serial-parallel, word-parallel, etc.

From the above, three things are evident in the context of an implementation. First, the parallelism available at the different levels varies enormously. Second, different types of parallelism may be traded off against others, depending on the desired cost:performance ratio (where for an FPGA cost may be measured in, say, the number of CLBs etc.); for example, the slow speed of a single functional unit may be balanced by having many such units operating concurrently. And third, not all types of parallelism are suitable for FPGA implementation: for example, the required routing-interconnections may be problematic, or the exploitation of bit-level parallelism may be constrained by the design of the device, or bit-level parallelism may simply not be appropriate, and so forth. In the Xilinx Virtex-4, for example, we shall see that it is possible to carry out many neural-network computations without using much of what is usually taken as FPGA fabric.\(^5\)

### 1.5 Xilinx Virtex-4 FPGA

In this section, we shall briefly give the details an current FPGA device, the Xilinx Virtex-4, that is typical of state-of-the-art FPGA devices. We shall below use this device in several running examples, as these are easiest understood in the context of a concrete device. The Virtex-4 is actually a family of devices with many common features but varying in speed, logic-capacity, etc.. The Virtex-E consists of an array of up to 192-by-116 tiles (in generic FPGA terms, configurable logic blocks or CLBs), up to 1392 Kb of Distributed-RAM, up to 9936 Kb of Block-RAM (arranged in 18-Kb blocks), up to 2 PowerPC 405 processors, up to 512 Xtreme DSP slices for arithmetic, input/output blocks, and so forth.\(^6\)

A tile is made of two DSP48 slices that together consist of eight function-generators (configured as 4-bit lookup tables capable of realizing any four-input boolean function), eight flip-flops, two fast carry-chains, 64 bits of Distributed-RAM, and 64-bits of shift register. There are two types of slices:

\(^5\)The definition here of FPGA fabric is, of course, subjective, and this reflects a need to deal with changes in FPGA realization. But the fundamental point remains valid: bit-level parallelism is not ideal for the given computations and the device in question.

\(^6\)Not all the stated maxima occur in any one device of the family.
SLICEM, which consists of logic, distributed RAM, and shift registers, and SLICEL, which consists of logic only. Figure 3 shows the basic elements of a tile.

Figure 3: DSP48 tile of Xilinx Virtex-4

Blocks of the Block-RAM are true dual-ported and reconfigurable to various widths and depths (from \(16K \times 1\) to \(512 \times 36\)); this memory lies outside the slices. Distributed RAM are located inside the slices and are nominally single-port but can be configured for dual-port operation. The PowerPC processor core is of 32-bit Harvard architecture, implemented as a 5-stage pipeline. The
significance of this last unit is in relation to the comment above on the serial parts of even highly parallel applications — one cannot live by parallelism alone. The maximum clock rate for all of the units above is 500 MHz.

Arithmetic functions in the Virtex-4 fall into one of two main categories: arithmetic within a tile and arithmetic within a collection of slices. All the slices together make up what is called the XtremeDSP [22]. DSP48 slices are optimized for multiply, add, and multiply-add operations. There are 512 DSP48 slices in the largest Virtex-4 device. Each slice has the organization shown in Figure 3 and consists primarily of an 18-bit × 18-bit multiplier, a 48-bit adder/subtractor, multiplexers, registers, and so forth. Given the importance of inner-product computations, it is the XtremeDSP that is here most crucial for neural-network applications. With 512 DSP48 slices operating at a peak rate of 500 MHz, a maximum performance of 256 Giga-MACs (multiply-accumulate operations) per second is possible. Observe that this is well beyond anything that has so far been offered by way of a custom neurocomputer.

1.6 Arithmetic

There are several aspects of computer arithmetic that need to be considered in the design of neurocomputers; these include data representation, inner-product computation, implementation of activation functions, storage and update of weights, and the nature of learning algorithms. Input/output, although not an arithmetic problem, is also important to ensure that arithmetic units can be supplied with inputs (and results sent out) at appropriate rates. Of these, the most important are the inner-product and the activation functions. Indeed, the latter is sufficiently significant and of such complexity that we shall devote to it an entirely separate section. In what follows, we shall discuss the others, with a special emphasis on inner-products. Activation functions, which here is restricted to the sigmoid (although the relevant techniques are not) are sufficiently complex that we have relegated them to separate section: given the ease with which multiplication and addition can be implemented, unless sufficient care is taken, it is the activation function that will be the limiting factor in performance.

*Data representation:* There is not much to be said here, especially since existing devices restrict the choice; nevertheless, such restrictions are not absolute, and there is, in any case, room to reflect on alternatives to what may be on offer. The standard representations are generally based on two’s complement. We do, however, wish to highlight the role that residue number systems (RNS) can play.

It is well-known that RNS, because of its carry-free properties, is particularly good for multiplication and addition [23]; and we have noted that inner-product is particularly important here. So there is a natural fit, it seems. Now,
to date RNS have not been particularly successful, primarily because of the
difficulties in converting between RNS representations and conventional ones.
What must be borne in mind, however, is the old adage that computing is about
insight, not numbers; what that means in this context is that the issue of con-
version need come up only if it is absolutely necessary. Consider, for example,
a neural network that is used for classification. The final result for each input is
binary: either a classification is correct or it is not. So, the representation used
in the computations is a side-issue: conversion need not be carried out as long
as an appropriate output can be obtained. (The same remark, of course, applies
to many other problems and not just neural networks.) As for the constraints
of off-the-shelf FPGA devices, two things may be observed: first, FPGA cells
typically perform operations on small slices (say, 4-bit or 8-bit) that are per-
fectly adequate for RNS digit-slice arithmetic; and, second, below the level
of digit-slices, RNS arithmetic will in any case be realized in a conventional
notation.

![Figure 4: XtremeDSP chain-configuration for an inner-product](image)

The other issue that is significant for representation is the precision used.
There have now been sufficient studies (e.g. [17]) that have established 16
bits for weights and 8 bits for activation-function inputs as good enough. With
this knowledge, the critical aspect then is when, due to considerations of performance or cost, lower precision must be used. Then a careful process of numerical analysis is needed.

\[ \sum_{i=1}^{N} w_i X_i \]

\( w_i \) is a weight and \( X_i \) is an input. (In general, this is the matrix-vector computation expressed by Equation 1.1.) In such a case, with a device such as the Xilinx Virtex-4, there are several possible implementations, of which we now give a few sketches. If \( N \) is small enough, then two direct implementations consist of either a chain (Figure 4) or a tree (Figure 5) of DSP48 slices. Evidently, the trade-off is one of latency versus efficient use of device logic: with a tree the use of tile logic is quite uneven and less efficient than with a chain. If \( N \) is large, then an obvious way to proceed is to use a combination of these two approaches. That is, partition the computation into several pieces, use a chain for each such piece, and then combine in a tree the results of these chains, or the other way around. But there are other possible approaches: for example, instead of using chains, one DSP48 slice could be used (with a feedback loop) to compute the result of each nominal chain, with all such results then combined in a chain or a tree. Of course, the latency will now be much higher.

**Figure 5:** XtremeDSP tree-configuration for an inner-product

*Sum-of-products computations:* There are several ways to implement this, depending on the number of datasets. If there is just one dataset, then the operation is \( \sum_{i=1}^{N} w_i X_i \), where \( w_i \) is a weight and \( X_i \) is an input. (In general, this is the matrix-vector computation expressed by Equation 1.1.) In such a case, with a device such as the Xilinx Virtex-4, there are several possible implementations, of which we now give a few sketches. If \( N \) is small enough, then two direct implementations consist of either a chain (Figure 4) or a tree (Figure 5) of DSP48 slices. Evidently, the trade-off is one of latency versus efficient use of device logic: with a tree the use of tile logic is quite uneven and less efficient than with a chain. If \( N \) is large, then an obvious way to proceed is to use a combination of these two approaches. That is, partition the computation into several pieces, use a chain for each such piece, and then combine in a tree the results of these chains, or the other way around. But there are other possible approaches: for example, instead of using chains, one DSP48 slice could be used (with a feedback loop) to compute the result of each nominal chain, with all such results then combined in a chain or a tree. Of course, the latency will now be much higher.
With multiple datasets, any of the above approaches can be used, although some are better than others — for example, tree structures are more amenable to pipelining. But there is now an additional issue: how to get data in and out at the appropriate rates. If the network is sufficiently large, then most of the inputs to the arithmetic units will be stored outside the device, and the number of device pins available for input/output becomes a minor issue. In this case, the organization of input/output is critical. So, in general, one needs to consider both large datasets as well as multiple data sets. The following discussions cover both aspects.

Storage and update of weights, input/output: For our purposes, Distributed-RAM is too small to hold most of the data that is to be processed, and therefore, in general Block-RAM will be used. Both weights and input values are stored in a single block and simultaneously read out (as the RAM is dual-ported). Of course, for very small networks, it may be practical to use the Distributed-RAM, especially to store the weights; but we will in general assume networks of arbitrary size. (A more practical use for Distributed-RAM is the storage of constants used to implement activation functions.) Note that the disparity (discussed below) between the rate of inner-product computations and activation-function computations means that there is more Distributed-RAM available for this purpose than appears at first glance. For large networks, even the Block-RAM may not be sufficient, and data has to be periodically loaded into and retrieved from the FPGA device. Given pin-limitations, careful consideration must be given to how this is done.

Let us suppose that we have multiple datasets and that each of these is very large. Then, the matrix-vector product of Equation 1.1, that is,

$$\mathbf{u} = \mathbf{W}\mathbf{y}$$

becomes a matrix-matrix product,

$$\mathbf{U} = \mathbf{W}\mathbf{Y},$$

where each column of $\mathbf{Y}$ is associated with one input dataset. The most common method used for matrix-matrix multiplication is the inner-product method; that is, each element of the output matrix is directly generated as an inner-product of two vectors of the input matrices. Once the basic method has been selected, the data must be processed — in particular, for large datasets, this includes bringing data into, and retrieving data from, the FPGA — exactly as indicated above. This is, of course, true for other methods as well.

Whether or not the inner-product method, which is a highly sequential method, is satisfactory depends a great deal on the basic processor microarchitecture, and there are at least two alternatives that should always be consid-
Consider a typical “naive” sequential implementation of matrix multiplication. The inner-product method would be encoded as three nested loops, the innermost of which computes the inner-product of a vector of one of the input matrices and a vector of the other input matrix:

\[
\text{for } i := 1 \text{ to } n \text{ do } \\
\quad \text{for } j := 1 \text{ to } n \text{ do } \\
\quad\quad \text{for } k := 1 \text{ to } n \text{ do } \\
\quad\quad \quad U[i, j] := U[i, j] + W[i, k] \times Y[k, j];
\]

(where we assume that the elements of \(U[i, j]\) have all been initialized to zero.) Let us call this the \textit{ijk-method}, based on the ordering of the index-changes. The only parallelism here is in the multiplication of individual matrix element and, to a much lesser extent (assuming the tree-method is used instead of the chain method) in the tree-summation of these products. That is, for \(n \times n\) matrices, the required \(n^2\) inner-products are computed one at a time. The middle-product method is obtained by interchanging two of the loops so as to yield the \textit{jki-method}. Now more parallelism is exposed, since \(n\) inner-products can be computed concurrently; this is the middle-product method. And the outer-product method is the \textit{kij-method}. Here all parallelism is now exposed: all \(n^2\) inner products can be computed concurrently. Nevertheless, it should be noted that no one method may be categorically said to be better than another — it all depends on the architecture, etc.

To put some meat to the bones above, let us consider a concrete example — the case of \(2 \times 2\) matrices. Further, let us assume that the multiply-accumulate (MAC) operations are carried out within the device but that all data has to be brought into the device. Then the process with each of the three methods is shown in Table 1. (The horizontal lines delineate groups of actions that may take place concurrently; that is within a column, actions separated by a line must be performed sequentially.)

A somewhat rough way to compare the three methods is measure the ratio, \(M : I\), of the number of MACs carried out per data value brought into the array. This measure clearly ranks the three methods in the order one would expect; also note that by this measure the kij-method is completely efficient (\(M : I = 1\)): every data value brought in is involved in a MAC. Nevertheless, it is not entirely satisfactory: for example, it shows that the kij-method to be better than the jki-method by factor, which is smaller that what our intuition

---

7 The reader who is familiar with compiler technology will readily recognise these as \textit{vectorization (parallelization)} by loop-interchange.

8 We have chosen this terminology to make it convenient to also include methods that have not yet been “named”.

---