FERNANDO J. PINEDA and ANDREAS G. ANDREOU

ANALOG NEUROMORPHIC COMPUTATION: AN APPLICATION TO COMPRESSION

Nature has evolved computing engines whose intelligence and natural abilities are unrivaled by modern computers. To match Mother Nature's abilities, we must overcome the same difficulties faced by natural systems, and we must learn to perform reliable computing with unreliable components. Steps in this direction are being taken by several groups at the Applied Physics Laboratory and at The Johns Hopkins University Department of Electrical and Computer Engineering. The purpose of the work is twofold: (1) to explore algorithms based on physical and neural models of computation and (2) to develop useful applications. We describe the basic approach and an experimental electronic neural network for the decompression of one-dimensional signals.

INTRODUCTION

Computation in biological systems is carried out on a substrate characterized by highly variable and noisy components. Nonlinearity and low precision abound. This is not the combination of properties we usually associate with today's electronic computing devices. Yet biological systems, with the fine-grained massively parallel organization of their processing elements, can solve problems in sensing, communication, and sensorimotor coordination with a speed and energetic efficiency that our best technology cannot match. For example, consider the mammalian retina, a tiny membrane at the back of the eye. As an outpost of the brain, it serves as both sensor and preprocessor for the visual cortex. Mead has performed several simple order of magnitude estimates[1] from which one can conclude that the combined supercomputers of the world would be unable to compute at a rate equal to that performed by the retina. The performance gap between natural and synthetic computation highlights two aspects of the development problem: (1) it suggests that conventional algorithms are not the best approach, and (2) it points out that even if the algorithms were to work well, conventional hardware would be suboptimal, since the power dissipated by all these machines would suffice to illuminate a small city. Yet we know that the entire human brain dissipates less power than a dim lightbulb.

Power dissipation ultimately limits physical computation, because computing machines must be able to dissipate the heat they generate. Modern computers are approaching this limit. As clock speeds have increased, efforts to channel heat out of processor chips have become more sophisticated and complex. In the end, one must reduce the amount of heat that is generated, since the thermal conductivity of the silicon itself limits the rate at which heat can be removed.

Current-mode subthreshold analog VLSI (very large scale integration) is an excellent medium for implementing neuromorphic algorithms.[2] Information is represented by currents typically measured in units of 10^{-1} to 10 nA. Voltage swings are typically measured in tens to hundreds of millivolts. The power dissipated in these circuits can be measured in microwatts. This dissipation is many orders of magnitude less than conventional digital microchips. Like biological neural networks, analog VLSI employs components that are noisy, imprecise, and nonlinear. Analog computation could be carried out with linear, precise, and low-noise components, much like it was done in the late 1950s before the advent of digital computation. This approach, however, does not scale to highly complex multimillion-component highly integrated systems. The neuromorphic approach, on the other hand, is to develop fine-grained massively parallel algorithms and architectures that are intrinsically insensitive to high noise, low precision, and device variability. Nonlinearity and noise are viewed as assets rather than liabilities.

In the neuromorphic approach, semiconductor physics and the physics of the microchip fabrication process play important roles in constraining candidate algorithms and architectures because the algorithms must use the natural operations of the substrate if they are to be scalable and energetically efficient. For example, simple charge conservation guarantees that the arithmetic operation of addition $(a + b)$ can be performed easily and precisely, provided the quantities a and b are represented as currents. Conversely, copying a quantity (i.e., a current), although easy, is not very precise. It can be performed by a two-transistor circuit called a current mirror. With minimum-size transistors in a 2-μm complementary metal oxide semiconductor (CMOS) process, current

mirrors will usually copy currents only to within a factor of 2! This rather large error is due to a process-dependent phenomenon known as transistor mismatch. In today's technology, transistor mismatch is caused mainly by variabilities in the fabrication process itself, not by fundamental physical limitations. However, as the technology scales down to ever-smaller sizes, the origin of mismatch will become fundamental, since it is related to the physics of ion implantation, an inherently stochastic process. Another operation, multiplication of signed quantities $(a \cdot b)$, requires as little as four transistors. The simplest circuit actually calculates the nonlinear quantity $f(a + \epsilon) \cdot f(b + \delta)$ instead of $a \cdot b$, where ϵ and δ are offsets caused by transistor mismatch, and where the function $f(.) = \tanh(.)$ arises from semiconductor physics. Thus, this nonlinear multiplication is a more natural operation in silicon than the usual multiplication.

The task of algorithm design for neural machines amounts to designing fine-grained massively parallel algorithms that use the natural arithmetic operations of the substrate and can cope with transistor mismatch. Algorithms and architectures for neural machines are not particularly well developed, nor are they particularly well understood. In this project we exploit the theory of a recently developed encoding algorithm to devise and implement a neural network machine that decodes previously encoded signals.

THE ALGORITHM

The architecture of our network is inspired by recent work on image encoding based on iterated transformation theory (ITT) as described by Jacquin.[3] The encoding algorithm is sometimes informally referred to as "fractal" coding. In fact, the algorithm is closely related to vector quantization. In traditional vector quantization, one transmits both the code book and the indices of the vectors that best approximate particular blocks in the image. In ITT coding, there is no code book. Instead, the encoder transmits a set of transformation coefficients that convey relationships between different blocks in the image. The transformations are used by the decoder, at the receiving end, to transform each block in an image into another block. The decoder starts with an initially random image and, by repeatedly applying the transformations, a sequence of images is generated that converges to a final image. The final image approximates the original uncoded image! Although this sounds like magic, it is actually based on some rather simple mathematics.

To see how the algorithm works, consider an image $\mathbf{I}$ to be a vector in a metric space. An *affine* transformation of a vector in this space is simply a transformation of the form

$$\mathbf{I}' = \mathbf{A}\mathbf{I} + \mathbf{B},$$

where $\mathbf{A}$ is an $N \times N$ component matrix and $\mathbf{B}$ is an N component vector. This transformation can be iterated to produce a sequence of vectors $\mathbf{I}^{(0)}, \ldots, \mathbf{I}^{(n)}$. It is simple to show that if the maximum eigenvalue of the matrix $\mathbf{A}$ is less than 1, the sequence converges to a unique final vector $\mathbf{I}^*$ that does not depend on the initial vector $\mathbf{I}^{(0)}$. The uniqueness of the final vector implies that if we want to transmit the vector $\mathbf{I}^*$ to a receiver, we can just as well transmit the $(N \times N) + N$ components of $\mathbf{A}$ and $\mathbf{B}$ and let the receiver perform the iteration to produce $\mathbf{I}^*$. In this case we say that $\mathbf{A}$ and $\mathbf{B}$ constitute an *encoding* of the vector $\mathbf{I}^*$. Furthermore, if the number of independent coefficients in $\mathbf{A}$ and $\mathbf{B}$ is less than the number of components in $\mathbf{I}^*$, and if we only transmit these independent components, we say that we have a *compressed* encoding of the vector $\mathbf{I}^*$. Compressed encodings are possible if $\mathbf{A}$ and $\mathbf{B}$ have sparse coefficients, functionally dependent coefficients, or some kind of symmetry. The transmitter and receiver must either agree beforehand on the structure of $\mathbf{A}$ and $\mathbf{B}$ or this information must be transmitted as part of the code. The ITT compression algorithm is simply a special case of the iterative approach just noted. It amounts to choosing $\mathbf{A}$ to be sparse in a blockwise sense. The transmitted code includes information about the block structure of the transformation as well as the actual values of the matrix elements.

Of course decoding a code is only half the problem, the other half being how to determine the code (i.e., the matrix elements of $\mathbf{A}$ and $\mathbf{B}$) in the first place. Suppose we want to encode and transmit a vector $\mathbf{T}$. How do we find the affine coefficients? The most obvious answer is to minimize $d(\mathbf{T}, \mathbf{I}^*)$, the distance between $\mathbf{T}$ and $\mathbf{I}^*$, with respect to the nonzero coefficients of $\mathbf{A}$ and $\mathbf{B}$. In fact this is the basis of all standard algorithms for performing this kind of compression. Different algorithms differ in the details of how they perform the minimization. Some algorithms are based on combinatorial search, whereas others are based on simple least-squares minimization.

The salient features of the affine theory just discussed are as follows: (1) the transformation only requires linear operations, and (2) convergence of the decoder requires that $\lambda_{max} < 1$. The general iterated transformation theory developed by Jacquin and others is similar to the affine theory, but it states that convergence is guaranteed, provided the transformation satisfies a more general technical condition known as contractivity. Any transformation that can be made contracting can be made the basis of a decoding and encoding scheme. In particular, *we can base a coding algorithm on the natural operations of the substrate*. Basically, we replace all multiplications in the simple transformation with nonlinear multiplications. The offsets and nonlinearities do not change the contractive nature of the transformation. To deal with random mismatch, we use a variant of the basic algorithm that aggregates outputs from many neurons. Simulations have shown this to be an effective method for reducing the effect of mismatch. The second change we make in the simple theory is to replace iteration of a transformation with relaxation of a differential equation. The differential equation can be implemented as a physical neural network with feedback. The use of feedback networks to implement iterative algorithms helps us to convert conventional algorithms into neural algorithms.[4]

To account for nonlinearities, the encoder must be modified by including a transistor model. This modification results in a technology-dependent encoder. The approach is actually quite similar to conventional

software practice. After all, we think nothing of using cross-compilers that run on one machine and produce code for another machine. The only difference is that, in our case, the encoder runs on a digital machine, whereas the target machine is an analog machine. We have, in fact, already developed a prototype software encoder that includes a simple transistor model. Given a vector that we wish to encode, the encoder calculates voltages that are appropriate inputs to the analog decoder.

THE ARCHITECTURE

The contractive iterated transformation approach to encoding and decoding can be implemented by a feedback neural network—a so-called recurrent neural network. The sparse connection topology and weights of the network correspond to the nonzero matrix elements of the transformation matrix **A**. Additions are performed by adding currents. Multiplications are performed with minimum transistor circuits. A simple example of the basic recurrent architecture is presented in Figure 1. The network consists of a computing layer, a switching layer, and an interconnection layer (the bus). The bus comprises wires that physically connect the neurons. In the figure there are two ranges of two neurons each. The ranges are identical in their computing and switching layers. They differ only in the way their outputs are connected to the bus. Each bus line is driven by a unique neuron. The parameters a_0, b_0, d_0 and a_1, b_1, d_1 are input as analog values. The parameters f_0, f_1 are presented as digital inputs. They represent a single bit of information used by the switching layer to determine whether the two neurons in a range are connected to even- or odd-numbered neurons. The four ways of connecting the neurons in this network correspond to the two bits of f_0 and f_1. The actual connections are selected at "run time" by the input to the switching layer. Of course in larger networks the possible connection topologies can be much more complex than the trivial connections shown in this example.

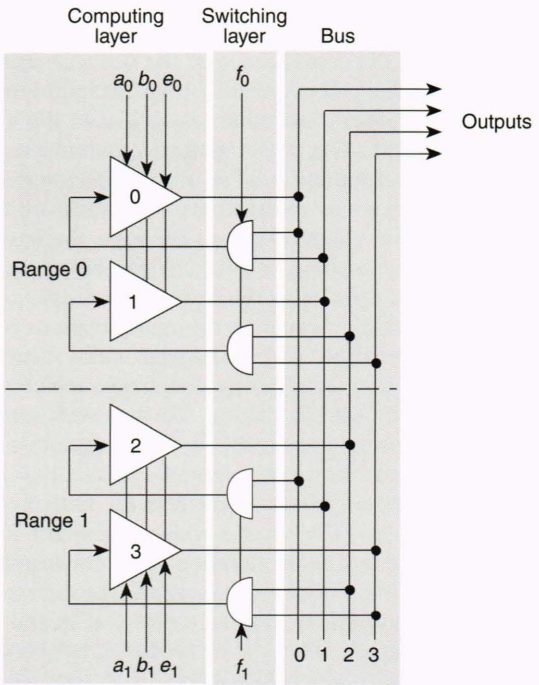


Figure 1. Schematic of a recurrent neural network architecture. Each neuron performs a transformation of the type $I_{out} = aI_{in} + bx + e$, where x is the neuron position ($x = 1, 2, 3, 4$). All neurons in a given range receive the same parametric input.

To run the network one simply sets the parameters. The network relaxes to a steady state where the currents coming out of the neurons represent the decoded vector. The network is always computing the vector that corresponds to the current input parameters. Significantly, there is no clock in this architecture: *continuous-time asynchronous signal/information processing is an important property of biological information processing systems.*

PROGRESS

We designed and laid out a sixteeen-neuron experimental chip in a 2-μm CMOS process and submitted the chip to the MOS implementation service (MOSIS). Four chips were recently received. Preliminary tests have deomonstrated that the design is sound and the circuit operates as expected.

To help lay out the chip we developed a simple, portable silicon compiler written specifically for analog design and for student projects. It has been successfully run on UNIX-, DOS-, and Macintosh-compatible workstations. The design for the entire chip is specified in a C language source code called a chip generator. When compiled and excecuted, this generator reads primitive cells as input and writes the layout for the entire chip as output. Routing is done automatically. The number of neurons, as well as the number of domains and ranges, is parameterized. Thus, the production of a larger chip or a chip with a different connection topology simply requires executing the generator with different parameters. We hope to use this new capability to establish a library of standard analog cells for application in future analog neuromorphic designs.

REFERENCES

[1] Faggin, F., and Mead, C., "VLSI Implementations of Neural Networks," in *An Introduction to Neural and Electronic Networks,* Zornetzer, S., David, J. L., and Lau, C. (eds.), Academic Press, San Diego, pp. 275-292 (1990).

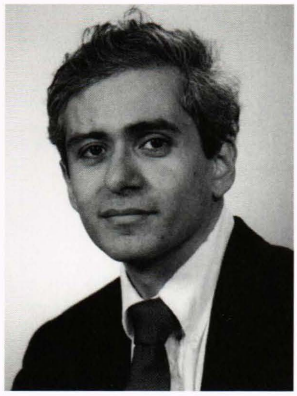
[2] Andreou, A. G., and Boahen, K. A., "Neural Information Processing I: The Current-Mode Approach," Chapter 6 in *Analog VLSI: Signal and Information Processing,* Ismail, M., and Fiez, T. (eds.), MacGraw-Hill, Inc., New York (1994).

[3] Jacquin, A. E., *A Fractal Theory of Iterated Markov Operators with Applications to Digital Image Coding*, Ph.D. Dissertation, Georgia Institute of Technology (1989).

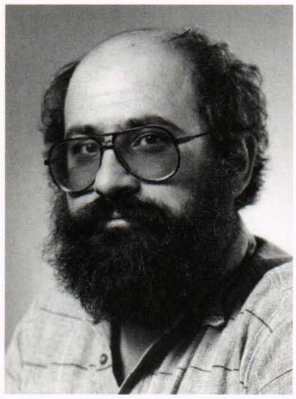
[4] Pineda, F., "Generalization of Back-Propagation to Recurrent Neural Networks," *Phys. Rev. Lett.* **59**, 2229-2232 (1987).

ACKNOWLEDGMENTS: The work described here is funded by APL Independent Research and Development as well as grant ECS9313934 from the National Science Foundation. The authors would like to thank Robert Jenkins and Kim Strohbehn for many useful conversations and comments on the manuscript.

THE AUTHORS



FERNANDO J. PINEDA received his B.S. in physics in 1977 from the Massachusetts Institute of Technology and his M.S. and Ph.D. in theoretical nuclear physics in 1981 and 1986, respectively, from the University of Maryland, College Park. He has been engaged in neural network theory, applications, and implementations since joining APL in 1986. Presently, he is a member of APL's Mathematics and Information Science Group, and is a lecturer in the Computer Science Department and a visiting scholar in the Department of Electrical and Computer Engineering at The Johns Hopkins University. Dr. Pineda has received research grants from the Air Force Office of Scientific Research. He serves as an editor on several publications including *Neural Computation*, *Applied Intelligence*, IEEE *Transactions on Neural Networks*, and *Neural Networks*.



ANDREAS G. ANDREOU received his M.Sc. (1982) and Ph.D. (1986) in electrical engineering and computer science from The Johns Hopkins University. He is an associate professor of electrical and computer engineering at JHU and a member of the APL Senior Staff in the Computer Science and Technology Group. Dr. Andreou received a Research Initiation Award from The National Science Foundation in 1990 and the JHU/APL R. W. Hart Prize for Independent Research and Development in 1989 and 1991. He is an associate editor of IEEE *Transactions on Neural Networks*.