STOCHASTIC COMPUTING IN NEURAL NETWORKS

Brian R Gaines and John G Cleary

Department of Computer Science, University of Calgary
Calgary, Alberta, Canada T2N 1N4

ABSTRACT

In conventional information systems uncertainty in information is regarded as "noise", and treated as a problem. In neural networks, however, uncertainty is an intrinsic feature of much of the information processing, with many of the computations involving the deliberate injection of stochastic processes. This paper analyzes uncertainty in information representation and processing in terms of the benefits arising, their theoretical foundations, their practical applications, and their hardware implementation. It gives a new perspective on network computations and the information involved. As an example of this approach the paper analyzes the representation of analog quantities by stochastic sequences of bits. Benefits of this approach are both theoretical and practical. Realization of communication links between units is greatly simplified and the number of wires needed in interconnecting VLSI chips correspondingly reduced. The hardware realization of basic operations on analog quantities is very simple, for example, in some cases a single and-gate can be used as a multiplier. The stochastic approach also gives elegant solutions to problems including learning, representation of system controllers, uncertain inference, and partial differential equations.

Introduction

Since the very beginning of computing in the late 1940s there has been interest in a class of computer architectures radically different from the traditional von Neumann style. The
major characteristic of these systems is a high degree of overt parallelism with many simple computational units heavily interconnected by low bandwidth communication lines. These systems are of interest because they seem well suited to solving problems of pattern recognition and of adaptation and learning. Also they seem more characteristic of the type of processing in the brain. Unfortunately, to implement such systems using modern technology (here equated with VLSI electronics) poses a considerable challenge. The great deal of connectivity between different units is essential to such systems and implies a lot of wiring to interconnect them. Wiring, however, is the dominant cost in VLSI systems (Sutherland, 1977). The scale of the problem may be appreciated when considering the interconnections in the brain. A human brain has about $10^{11}$ neurons. Depending on the type of neural cell and the part of the brain an individual cell may have from 1,000 to 100,000 synaptic connections to it.

For someone attempting to recreate a brain in VLSI technology there are (at least) two major problems; the provision of active computing elements corresponding to synaptic connections; and the provision of the interconnections themselves. The fundamental restrictions imposed by VLSI technology are twofold. First that within a chip (or wafer) wiring is essentially two dimensional with only a small number of layers available for wires to cross each other. Second, that only a very small number of wires can be brought off a chip (typically less than 100) and that routing and wiring off chip is much more expensive than that within a chip. The major advantage the brain has is that it can perform three dimensional wiring (although the convoluted folding of the neo-cortex seems to be an attempt to deal with a two dimensional pattern of wiring at a relatively gross level of organization). If this were all that there were to the comparison between the brain and VLSI technology the prospect for a neural style of VLSI computer would be poor. However, VLSI technology has one huge advantage over the brain, it operates much more quickly. Single gate delays are typically of the order of tens of nanoseconds while in the brain times between pulses are typically of the order of milliseconds, a difference of five orders of magnitude. The approach taken here is to trade the
difficulty of doing wiring in VLSI against its speed. This is done in two ways, first, by communicating using stochastic sequences of bits to represent the analog quantities, and second, by heavily multiplexing the resulting signals.

Early work on connectionist structures such as the Perceptron focussed on such topics as the ability of such systems to learn to do ‘anything.’ This sprang from very early work by (Hebb, 1949), (McCulloch, 1943) drawing heavily on analogies from the brain. Even von Neumann (1945) spoke of the new designs for stored instruction computers in terms of distributed processing and neural structures. This early context is well summarized in (Hinton, 1981) Perceptrons have been largely ignored by the main stream of AI since the work of Minsky and Papert (1968) which showed that in practice they were incapable of learning all tasks. However, recent work (Hinton, 1981) has focussed not on the universal properties of these systems but on their practical importance. They have many useful features that seem at best very awkward in the context of von Neumann computers. For example Hinton (1984) and Kohonen (1984) discuss the ability of connectionist systems to implement content-addressable memory, automatic generalization, learning, and robustness in the face of input errors. One of the important things about these properties is that they appear as primitive properties of connectionist systems, they do not need to be grafted on top of an existing structure. Another important property is that they make intensive and efficient use of the available hardware, Gaines (1969b) has also pointed out that this will be necessary if a number of very difficult computing problems are to be solved. This point about use of hardware may be seen in context by noting that most of the circuitry in von Neumann computers is tied up in memory chips each part of which is almost never used for processing and furthermore that as the size of the memory grows less and less use is made of any particular part of the memory.
**Stochastic Computing**

One radical approach to connectionist architectures which is of particular relevance to modern developments in neural nets is the *stochastic computer* based on the use of probabilistic variables. Phenomena in networks such as Boltzmann machines (Rumelhart, McClelland & PDP Research Group 1986) can be treated as those of binary variables subject to random processes. However, we will consider an alternative perspective which treats the *probability* of a binary variable as itself representing information (Gaines 1967a,b,c 1969a,b). Many of the network computations can then be treated as continuous operations on analog variables rather than logical operations on binary variables. A complete range of computational operations in such representations can be carried out in digital networks.

The concept of using the probability of a binary variable to encode an analog value was introduced in 1965 in research on learning machines (Gaines & Andreae 1966) at Standard Telecommunications Laboratories in England and termed 'stochastic computing' (Gaines 1965). The same approach was discovered independently at the same time at the University of Illinois in research on parallel image processing systems (Poppelbaum & Afuso 1965). Research on stochastic computing proceeded at many sites in Europe, the USA and Japan, resulting in two books (Massen 1977, Mars & Poppelbaum 1981) and an International Symposium in Toulouse in 1978 (Stochastic Computing 1978). However, the speed and cost limitations of using conventional digital circuit elements in a random mode restricted practical applications. In 1969 it was suggested that such applications depended on the harnessing of natural stochastic processes such as photon interactions in optical computers (Gaines 1969b), and the lack of practical technology has limited developments in stochastic computing in the past eighteen years.

However, advances in technology have made very large scale parallel computing networks feasible, and many of the high-density electronic and photonic devices being developed for such networks exhibit natural quantum-mechanical stochastic effects suitable for
stochastic computing. Many proposed networks incorporate random processes although they have not been developed from a stochastic computing perspective. Early work on stochastic computing was targeted on the applications being developed today: machine learning; image processing; Bayesian estimation; partial differential equation solution; pattern recognition and so on. Computational elements, operational systems and some deep theoretical results have been developed for these systems.

This paper reviews stochastic computers, their applications, their realization in VLSI circuits, and theoretical results. These suggest that the stochastic computing perspective of viewing the networks as performing analog computations on probabilities may be valuable in the analysis of a variety of parallel distributed processing systems.

Operation of a Stochastic Computer

The most obvious way to perform stochastic computation in a digital network is to treat the probability that a digital signal will be in the 1 state as an analog variable. If two statistically independent digital signals are brought together in an AND gate then their probabilities multiply. This has been the starting point for the invention, and re-invention, of stochastic computing. Figure 1 shows how this basic idea may be extended to give the additional operations of weighted addition and integration. Addition has to be weighted by the incorporation of an extra random source since the straightforward addition of probabilities could lead to values outside the representable range. A digital counter acts as integrator, and its parallel digital output can be converted back to a stochastic signal by comparing it with a random parallel digital source. The integrator is the basis for the stochastic neuron which is introduced later.

Bipolar Representation

In most applications negative numbers must also be represented, and Figure 2 shows a simple extension which allows stochastic computing to be carried out with variables taking both
positive and negative values. Two stochastic signals the \textit{up} and \textit{down} signals are used, with probabilities \( u \) and \( d \). The analog value \( x = u - d \). This representation is used later when we discuss uncertain reasoning.

\textit{Bipolar Representation - Single Wire}

Figure 3 shows another bipolar representation using only a single stochastic sequence. If the probability of a 1 is \( p \) then the value represented is \( x = p(1-p) = 2p - 1 \). An EXCLUSIVE-NOR gate acts as a multiplier in this representation. The weighted adder continues to be effective. A simple inverter gives negation, and hence allows subtraction. Gating at the input of the integrator allows for two summing inputs and a HOLD line.

The noise source and comparator of the integrator act as a natural input device in this representation if the counter is replaced by a parallel digital signal. Analog inputs can be converted by comparing them with a random analog noise source. Negative feedback around the integrator may be used to construct an output interface in that the contents of the counter may be shown to track the probability of the input. The resultant stochastic averaging element shown at the bottom of Figure 4 is an important component in many applications such as machine learning and has been termed an ADDIE (ADaptive DIgital Element, Gaines 1969b).

\textit{Nonlinear Representations}

A particularly interesting class of stochastic computations are those in which the representation is nonlinear. For example, the representation \( x = p/(1-p) \), where \( p \) is the probability of an internal binary event representing an external analog variable \( x \), allows variables in the range zero to infinity to be used in computations. Figure 5 shows that a range of computational operations in this representation can be carried out in digital networks.

An interesting feature of this nonlinear representation is that it shows behavior related to that of the human vision process such as wide range of representation, differential light-dark adaption and Weber-Fechner law of sensitivity. These arise through its logarithmic error
sensitivity—that the just noticeable difference in two values is proportional to those values—and the natural input conversion which is a device whose sensitivity decreases with the rate of input and increases by a relaxation process. It may be that the $p/(1-p)$ representation is a natural one for illumination in the neuro-optical system.

**Sigma Pi Units**

Many variant stochastic computations may be generated by suitable combinational networks. "Sigma pi" units combining multiplication and addition may readily be designed for each representation as shown in Figure 6. The $u$ and $v$ inputs may be seen as controls varying the effects of the $x$ and $y$ inputs. Such units are known to be capable of generating any monotonic function (Rumelhart, McClelland & PDP Research Group 1986). However, in stochastic computing it is usually appropriate to design such functions through specific combinational networks rather than by cascading simpler units.

**Partial Differential Equation Solution in Networks**

Uniform networks of simple stochastic computing elements may be used to solve partial differential equations, such as the diffusion equation, from their boundary conditions. Figure 7 shows a four input summer into an ADDIE used to solve the diffusion equation in two dimensions. Such *thermodynamic* nets have become of great practical interest through the development of the Boltzmann machine, and it would be interesting to investigate the relation between digital learning networks and spatial partial differential equation solvers.

**Stochastic Learning Algorithms**

A variety of stochastic computing networks for machine learning have been proposed including those implementing Bayesian prediction (Gaines 1967b, 1969b). Stochastic digital perceptrons have been shown to be more powerful in their learning behavior than deterministic
digital perceptrons (see Section 3.2). Practical applications of stochastic learning systems have been studied (Mars & Poppelbaum 1981, Stochastic Computing 1978).

**Stochastic Neuron**

Many connectionist systems make use of some analog of a neuron. Such units take a linear weighted sum of their inputs and compute some output of this value. By adapting the integrator such a unit is easily constructed. A digital running count is maintained at each time step using the following formula:

\[ S = \sum w_i s_i \]

\[ \text{count} = (1-a)\text{count} + S - c_0 \]

where the \( s_i \) are the inputs to the neuron and the \( w_i \) are the associated weights.

The (stochastic) output is computed using

\[
\text{if } \text{count} > 0 \text{ then output 1; count = count - c_1} \\
\text{else output 0;}
\]

Because each of the \( s_i \) are 0 or 1 (using the simplest stochastic representation) implementation involves only incrementing the counter. If \( a \) is made a power of 1/2 then a shift and decrement suffices to compute \((1-a)\text{count}\). A number of different functions can be obtained depending on the values of the constants \( a, c_0 \) and \( c_1 \). 'a' plays the role of feedback (as in the ADDIE of Figure 4.) and provides a time averaging of the inputs. It also prevents the counter from over(under) flowing by limiting its maximum value to \( \text{max}/(1-a) \) where \( \text{max} \) is the largest possible value of the sum \( S \). Figure 8 shows the output of the neuron as a function of \( S \) for different values of the constants.

If \( a = 1 \) then there is no time averaging and the output depends only on \( c_0 \). The function is then a step function at \( S = c_0 \) with a small amount of smoothing provided by the variance of the inputs. If count were compared to a random reference variable as in the
integrators then this would convert this function from a step function to one which varied linearly with $S$.

If $a = 0$ then the output saturates at 1 when $S \geq c_1 + c_0$ and saturates at 0 when $S \leq c_0$.

Between these extremes the output is a linear function of $S$. This configuration is not practical because the counter can easily overflow if it is in the saturation region, however, $a$ can be brought arbitrarily close to 1 at the expense of more bits in the counter.

For $0 < a < 1$ the curve takes on a smoother sigmoid curve characteristic of all good neurons. The output is still 0 when $S < c_0$ and saturates close to 1 when $S \geq (1-a)c_1 + c_0$

**The Power of Randomness**

The perspective that sees stochastic phenomena in networks as a form of analog computation on probabilities is useful as a conceptual device. However, the role of truly random behavior in networks also has more fundamental consequences. Stochastic automata are intrinsically more powerful computing devices than deterministic automata, and uncertain networks can perform computations that are beyond deterministic networks.

**Accuracy of Stochastic Computing**

Before looking at more complex applications a natural question which arises is the accuracy of the representations in a stochastic computer. Unlike most computer systems the accuracy is not a function of some architectural parameter such as word size but of the time over which a computation is done. The source of inaccuracy is the randomness of the stochastic variables. If a variable is sampled $N$ times then the variance in the answer is $p(1-p)/N$ so the accuracy of the resulting estimate of $p$ is $\sqrt{p(1-p)/N}$. That is to achieve an accuracy of 1 part in $n n^2/p(1-p)$ samples are needed. This may seem to be very inefficient as the usual computer representations require only log $n$ bits for the same accuracy. However, there are three mitigating factors.
In neural nets one is usually more interested in approximate answers quickly arrived at
than very accurate answers slowly arrived at. In many applications 10% accuracy requiring a
100 samples is good enough (Shortliffe, 1976, p183). Secondly the stochastic variables
potentially carry much more information than just a simple single valued probability. For
example, later when discussing uncertain reasoning we will find that the correlations between
different stochastic variables enables correct uncertainty values to be calculated in situations
where more standard approaches are totally intractable. Thirdly because the calculations of
functions such as multiplication can be done with only a few gates high clocking speeds and
packing densities are possible for hardware implementations.

In any stochastic computer it is necessary to generate many uncorrelated random
sequences. As noted in (Gaines, 1969, Sec. 4.16) it is easy to generate a large number of
uncorrelated sequences with p=1/2 using suitably long shift registers. For example, a single 33
bit shift register can deliver $2^{32}$ independent sequences. These are readily combined to deliver
sequences with appropriate probabilities.

System Controller

It is known that stochastic automata can define languages that cannot be defined by a
deterministic automaton, and that even when an equivalent deterministic automaton exists it
will generally have many more states (Rabin 1963). There are practical examples of this
phenomenon in learning applications (Witten & Gaines 1976). For example, consider the
problem of regulating a discrete dynamical system to maintain its state within a prescribed
region. Represent the system as an automaton, $(I,P,S,\pi,\sigma)$, where $I$ is a finite input alphabet,
$P={0,1}$ is a binary set of outputs, $S$ is a set of states, $\sigma:S\times I\rightarrow S$ is the next state function, and
$\pi:S\rightarrow P$ is the output function. In this formulation is a performance function and the problem is
to regulate the inputs to the automaton to cause its output to become, and remain, 1.
Suppose that there is some distinguished element \( \lambda \in I \) (the ‘zero’ input for the autonomous system), and consider the following sets of states:

\[
\begin{align*}
W &= \{ s : \pi(s) = 1 \} \\
A &= \{ s : \forall n \geq 0, \sigma(s,\lambda^n) \in W \} \\
B &= \{ s : \exists n : \sigma(s,\lambda^n) \in A \}
\end{align*}
\]

\( W \) is the subset of \( S \) in which it is desired that the state should reside; \( A \) is a weak attractor within \( W \); and \( B \) is its region of attraction. We assume that both \( B \) and \( S-B \) are non-empty so that the autonomous system has a region of local asymptotic stability but is not asymptotically stable in the large, and consider the family of control automata whose inputs are from \( P \) and whose outputs are in \( I \) which induce global stability. For this family to be non-empty it is necessary that \( B \) be reachable from \( S \), that is:

\[
\forall s \in S, u \in I^*, \exists \sigma(s,u) \in B
\]

where \( I^* \) is the free semigroup generated by \( I \).

This problem is one of a class considered independently by Gold (1971) in his paper on *Universal Goal-seekers*, and by Gaines (1971) in a paper on stochastic automata. Both authors demonstrate that no finite state deterministic automaton can act as a universal regulator for this class of problems. On the contrary it is shown that there is a finite state dynamical system which is universally insoluble for all such regulators with less than a given number of states. Both authors give a construction for such a system which has the property that any action by the regulator other than that which it actually does would immediately solve the problem. Gaines calls such a system a *frustration automaton*, and Gold terms the resulting behavior *strongly worst*.

Gold demonstrates that there is a universal primitive-recursive (and hence potentially infinite state) deterministic regulator, and Gaines demonstrates that there is a universal two-
state stochastic regulator. This equivalence between recursive and stochastic solutions, together with the non-existence of finite state deterministic solutions, demonstrates the savings in memory and complexity resulting from the introduction of randomness. It suggests a role for random processes in computational and neurological systems in that they enable a simple, finite memory structure to achieve the same capability as a complex structure requiring unbounded memory.

It might be supposed that the stochastic regulator is very much less effective than the recursive one in terms of the time taken to solve the problem. However, a simple example shows that this is not so. Moore's (1956) combination lock problem involves an automaton whose output is zero until the correct sequence of inputs is given (and the lock opens). At any stage an incorrect input resets the automaton to its initial state. Moore shows that a sequence of length at least \( 2^{N-1} \) is necessary to open an arbitrary binary lock with \( N \) states. It can be shown that the mean length of sequence to open the lock with a random generator is \( 2^N \). Theoretical upper bounds on the length of a deterministic solution are very high, but simulation results suggest that the actual mean length is close to that for the stochastic solution (Witten 1975).

These theoretical results have practical applications. They are the foundations for the use of simple localized random processes to break deadlocks in distributed systems rather than the development of complex central arbitration systems. In terms of Dijkstra's (1965) famous dining philosophers problem, one notes that each will get a fair share of spaghetti if, on finding they have only one fork, they toss it in the air and, and if it falls the right side up, put it back on the table for a while so that another philosopher (whose toss has led him to keep his fork) may grab it.

The results also underlie the difficulties that people have in coping with randomness in the world (Gaines 1976a) and the difficulty of modeling stochastic automata (Gaines 1976b, 1977).
Convergence of Learning Networks

These theoretical results have practical consequences for neural networks. Convergence theorems for pattern-recognizing networks depend on the continuity of the weights and convergence fails in practice if the weights are discrete (Gaines 1969b, Gaines & Quarmby 1968). Convergence can be re-established both theoretically and practically if the weight changes are made discretely but through biassed random variables. That is, a perceptron-like adaptive network with discrete weights is only able to converge to available solutions if the weight changes are made stochastically.

The phenomenon underlying this result can be seen intuitively if the process is thought of as attempt to follow a steepest descent path in a discrete space. The analog steepest descent point will be approximated by the nearest available discrete point. This does not allow partial descents to accumulate and may lead to limit cycles in which the perceptron repeatedly makes a non-optimal decision.

It is interesting that this behavior was observed empirically in research at IBM on machine learning in the 1960s (Clapper 1967). It was found that a digital perceptron applied to speech converged to reasonable discriminations but that the same device applied to handwriting did not converge when solutions were available. It was noted that the speech data was 'noisier' than the handwriting data, so noise was added to the handwriting data and the system converged.

Application to Probabilistic Inference in Knowledge-Based Systems

Uncertainty is an important part of many ruled based expert systems. For example, applications such as medical diagnosis do not allow anything but the weakest inferences to be made from available evidence. To report certainty in conclusions is both incorrect and misleading. A number of schemes for expressing and computing such uncertainties have been developed. For example, in MYCIN and EMYCIN "certainty factors" are used (Shortliffe, 1976). A certainty factor is a number between -1 and +1, -1 is intended to express sure knowledge that something is false and +1 sure knowledge that it is true. Intermediate values
express varying degrees of ambivalence about the truth. For example 0 expresses a complete lack of knowledge about truth or falsity. Such certainty factors can be used in a number of ways. For example a rule such as:

\[ \text{will-rain} \leftarrow \text{dark-cloud and falling-pressure} \text{ with certainty 0.6;} \]

says that it is almost certainly true that if there is dark cloud around and the barometric pressure is falling then it will rain, although, there will be some cases where this is not true. Certainty factors can be propagated through the system to evaluate the certainty of conclusions. For example the conclusion 'will-rain' would be given a certainty factor based on the certainty of the rule above and the certainties of 'dark-cloud' and 'falling-pressure'. Unfortunately there are grave problems with using certainty factors in this way. Consider the additional rule:

\[ \text{will-rain} \leftarrow \text{lightning} \text{ with certainty 0.4;} \]

If both these rules fire then 'will-rain' is given a higher certainty factor than if only one of them fires. Unfortunately the second rule is merely another way of saying that storm clouds are present and the conclusion is not much more true as a result.

These problems can be seen more starkly by considering the expression '(dark-cloud or not dark-cloud)' or the expression '(dark-cloud and not dark-cloud)' which are respectively always true or false. However certainty factors ignore the fact that the two parts of the expression are correlated (the same) and report intermediate values for both expressions.

Another way to approach this is to use the probability that an expression is true rather than certainty factors. Again problems arise that are similar to those above. For example one scheme used to compose such probabilities is obtained by assuming the various parts of an expression are uncorrelated:

\[
\begin{align*}
p(x \text{ and } y) &= p(x) \times p(y) \\
p(x \text{ or } y) &= p(x) + p(y) - p(x) \times p(y) \\
p(\text{not } x) &= 1-p(x)
\end{align*}
\]

Let \( p(x) = 1/2 \) then \( p(x \text{ and not } x) = 1/4 \) and \( p(x \text{ or not } x) = 3/4 \), neither of which is correct.
A second evaluation scheme (Zadeh, 1965) assumes that some correlation can occur between expressions and lets:

\[
\begin{align*}
    p(x \text{ and } y) &= \min(p(x), p(y)) \\
    p(x \text{ or } y) &= \max(p(x), p(y)) \\
    p(\text{not } x) &= 1 - p(x)
\end{align*}
\]

Again let \(p(x) = 1/2\) then \(p(x \text{ and not } x) = 1/2\) and \(p(x \text{ or not } x) = 1/2\) is still wrong. This scheme does not overweight rules which are similar but does underweight rules which are independent.

A related problem is that some conclusions may have evidence both indicating that they are true and evidence indicating that they are false. It seems, intuitively, that the situation where there is no evidence about something is different from the one where there is a known 0.5 probability that it is true and a known 0.5 probability that it is false. The schemes mentioned above also have problems in consistently accommodating both positive and negative evidence.

Considerations such as these show that it does not seem to be possible to provide a quantitative theory of truth and falsity using just probability values for expressions. Other attempts to provide a logical basis for uncertain reasoning have been made (Shapiro, 1983), (van Emden, 1986) but because they are based on the probability values of expressions they also are heir to the ills described above. More details on the ideas presented here can be found in (Cleary 1987a) and an extension to Prolog incorporating negation in a similar way can be found in (Cleary 1987c).

**A PROBABILISTIC LOGIC**

The technique used here is to assign each logical expression a stochastic sequence of true/false values rather than just one true/false possibility. This is trivially different from the normal propositional calculus and all theorems hold for example \((x \text{ or not } x) = (true, true, ...)\) and \((x \text{ and not } x) = (false, false, ...)\). In order to make this useful a new family of logical
constants are introduced. Each constant is some random sequence of true/false values with a fixed probability that it is true. The constants are written in the form \( \tau(p) \) where \( p \) is the probability that an item in the sequence is true. So, it is always true that

\[
\tau(1) = (\text{true, true, ..}) \quad \text{and} \quad \tau(0) = (\text{false, false, ..}).
\]

To express an uncertain rule the constants can be used as follows:

\[
\text{will-rain} \leftarrow \text{dark-cloud and falling-pressure and } \tau(0.6);
\]

This says that if there is dark cloud and falling barometric pressure then in 0.6 of the cases there will be rain. The second rule can be expressed as:

\[
\text{will-rain} \leftarrow \text{lightning and } \tau(0.4);
\]

This has still not solved the problem that the two rules are highly correlated but they can be reformulated as:

\[
\begin{align*}
\text{will-rain} & \leftarrow \text{dark-cloud and falling-pressure and } \tau_1(0.75) \\
\text{and} \\
\text{will-rain} & \leftarrow \text{lightning and } \tau_2(0.5)
\end{align*}
\]

\[\tau_3(0.8)\]

This reformulation says that the two rules have a common cause which says that they are true 80% of the time. As a result of this the probability of will-rain will only be weakly augmented when both rules fire, solving the original problem of expressing the fact the two rules are not independent of each other. By rewriting these rules in the equivalent form below it can be seen that the original \( \tau(0.6) \) has been replaced by \( (\tau_1(0.75) \text{ and } \tau_3(0.8)) \) and \( \tau(0.4) \) by \( (\tau_2(0.5) \text{ and } \tau_3(0.8)) \):

\[
\begin{align*}
\text{will-rain} & \leftarrow \text{dark-cloud and falling-pressure and } \tau_1(0.75) \text{ and } \tau_3(0.8) \\
\text{will-rain} & \leftarrow \text{lightning and } \tau_2(0.5) \text{ and } \tau_3(0.8)
\end{align*}
\]

In this way sets of rules with arbitrary correlations between them can be expressed. This is an example of how stochastic sequences carry more information than just the probability that they will be true. The result of all this is a probability logic (Gaines, 1984), (Rescher, 1963). The logic obeys all the usual logical axioms including the tautology (x or not x).
NEGATION

When including negation in uncertain reasoning the statement of facts must include information that some things are true, that others are false and that some are just not known. The usual way of handling this is to make the closed world assumption that "if something cannot be proven then it is false." This is far too draconian for the current purposes as it makes it impossible to express the fact that a particular fact is just unknown. It is ridiculous to conclude that because I do not know whether it is sunny therefore it must definitely be cloudy.

A resolution of this is to extend the truth values to include undefined as well as true and false and to introduce two distinct forms of negation. As above the truth values are infinite sequences. The interpretation of true is "provably true" and of false is "provably false". The two forms of negation are denoted by \( \sim \) and \( \neg \). Their truth tables are:

\[
\begin{array}{c|ccc}
\sim & t & u & f \\
\hline
f & u & t
\end{array}
\quad
\begin{array}{c|ccc}
\neg & t & u & f \\
\hline
f & t & t & t
\end{array}
\]

\( \sim \) should be interpreted as "provably not" and \( \neg \) as "not provable". So \( \neg \) corresponds to the normal closed world notion of negation.

The truth tables for the various connectives are:

\[
\begin{array}{c|ccc}
\land & t & u & f \\
\hline
t & t & u & f \\
u & u & u & f \\
f & f & f & f
\end{array}
\quad
\begin{array}{c|ccc}
\lor & t & u & f \\
\hline
t & t & t & t \\
u & u & u & u \\
f & f & f & f
\end{array}
\quad
\begin{array}{c|ccc}
\Rightarrow & t & u & f \\
\hline
t & t & t & t \\
u & u & t & t \\
f & f & f & t
\end{array}
\]

Such logical variables are easily represented by two stochastic sequences for each value. This gives a representation similar to the bipolar values in Figure 2.

In the introduction the major problems of uncertain reasoning were encapsulated in the problem of ensuring that the expression \((x \text{ or not } x)\) was always true and that \((x \text{ and not } x)\) was
always false. These need to be restated carefully in the new logic but there do indeed exist statements with the correct properties. For example, \( x \lor \neg x \) is always true (\( x \) is provably true or not provably false) and \( \neg x \lor x \) is never false. Similarly, \( \neg (\neg x \land x) \), \( \neg (\neg x \land x) \) and \( \neg (\neg x \land x) \) are always true.

**Extended Horn Clauses**

To accommodate these new notions the form of clauses allowed in the rule set can be extended as follows. The general form of rules is:

\[
a \leftarrow b_1, b_2, b_3, \ldots, b_n
\]

where \( a \), the head of the clause, can be a term of the form \( x \) or \( \neg x \) and the \( b_i \) can be of the form \( \neg x \), \( \neg x \) or \( x \) where \( x \) is some atomic formula. Also, the \( b_i \) can be constants of the form \( \tau(p) \) or \( \neg \tau(p) \). \( \neg x \) is not permitted in the head of a clause. Because the \( \tau \) constants can only occur on the right-hand side of rules which will not "fire" if any of their terms are undefined there is no need to allow for undefined values in the infinite constants. There is no logical reason to exclude them, they just serve no useful purpose in this Horn clause logic.

The operational interpretation of these rules is that whenever \( x \) appears in the head of a rule then this forces the truth value of \( x \) to true if the body evaluates to true, and similarly if \( \neg x \) appears in the head of a rule then this forces the truth value of \( x \) to false. A weaker form of the closed world assumption is needed to completely define this procedure, that is, "if a value cannot be proven true or proven false then it is undefined". This seems much more palatable than the original form. These procedures ensure that none of the rules is provably false.
Contradiction

This opens the possibility that two rules will attempt to force the same conclusion to be both true and false. In systems with a single truth value such a contradiction is catastrophic, the rule set has to be rejected (or debugged). If an infinite sequence of values is available the situation is not as bad. Any position along the sequence which generates a contradiction on any value at the head of a rule will cause all conclusions at that position to be ignored. In a rule set where this happens a lot, the precision of the inferred probabilities will drop as a smaller number of positions are available to count from. Although any high probability of contradiction will be untenable, the system is at least graceful enough to allow some degree of contradiction and not fail catastrophically.

The $\Sigma$ Chip—A VLSI Architecture for Stochastic Computers

The construction of VLSI hardware is probably essential if the potential of stochastic neural systems are to be realized. In this section we briefly examine a design for a very simple chip to support such computing. Each $\Sigma$ chip supports some 100 stochastic neurons. To reduce the wiring complexity of the resulting system 100 stochastic sequences are time multiplexed on to each physical wire. The resulting chip has 10 input wires and one output wire. Figure 9 illustrates this architecture. The major restriction of the current design is that weights are restricted to single bit values, this is a restriction of chip area rather than a conceptual problem or one of computing speed. Also no provision is made for adapting or learning the weights. Even in this restricted form it seems capable of executing a number of interesting tasks. Further details on the system can be found in (Cleary 1986a, 1987b), (Sahebkar 1987).

Calculations

The calculation done by each of the units on the chip is a weighted sum of its 1000 inputs. As each of the inputs is a 0 or 1 and each of the weights is similarly a 0 or 1 the weighted sum is equivalent to "and"ing the inputs against a mask of weights and counting the
number of 1's which result. The output of the unit is computed by comparing this sum against a threshold (which may be different for each unit). Each unit can also be separately programmed to produce an output 1 depending on whether the sum is greater, less, equal, not equal, not greater, or not less than the threshold. This set of operations is sufficient to program many interesting logical operations. For example, to determine if all a units inputs are on (an "and" of all the input values): all the weights for the unit should be set to 1; the threshold set to 1000; and an output 1 produced if the sum equals the threshold. Similarly an "or" of any subset of the inputs can be produced by: setting the weights of the subset to 1; setting the threshold to 0; and producing a 1 output if the sum is greater than the threshold. A great number of other operations which have no simple logical form can also be programmed, for example, "any two of some subset of the inputs are on".

Each Σ-chip then needs to contain sufficient memory for 100×1000 = 100,000 weights and for 100 thresholds, as well as logic for doing the summations and threshold calculations for each of the 100 units. Details of a VLSI design for the Σ-chip are given below after some applications are considered. Vital statistics of this design are given in Table I. This includes an estimated clocking frequency of 4 MHz, which implies that every stochastic neuron will produce an output at 0.04 MHz that is every 25μsecs.

A Σ-chip on its own is not a complete system. How the chips are interconnected and arranged will depend on the particular class of applications. In general though there will need to be some general purpose host computer to provide programming and control for the system. To this end the weights, thresholds and mode of output calculation for each unit can be loaded onto the chip under external control.

III Applications

A number of applications for the Σ-chip have been programmed and simulated. A language based on Prolog and a compiler which generates chip weights and thresholds have
been written. The following applications were all written and debugged using the compiler and simulator. Details are given by Sahebkar (1987) and Cleary (1986a).

**Rule Based Reasoning**

The uncertain reasoning described earlier can be implemented on the $\Sigma$ chip. All rules will be recalculated every 25$\mu$secs and any conclusions to be drawn from them will take another 25$\mu$secs. Thus, the time for the system to complete will be determined by the length of the longest chain of rules used in going from facts to final conclusions times 50$\mu$secs. The rules and their interrelationships are determined by the weights loaded, the particular problem being computed is determined by the initialization of the outputs. For 1,000 or less rules and facts a system with 10 $\Sigma$-chips fully interconnected is sufficient. For larger systems some more structured wiring pattern would be necessary.

**Constraint Satisfaction - 8-Queens**

A type of problem which occurs in many contexts is searching to find a set of true/false values which satisfy some constraints. This type of problem is best represented by assigning each variable in the system two units, one which says that the variable is true and one which says that the variable is false. If both these units are on then the system is contradictory and no solution can be found, if neither is on then the value of the variable is undetermined. Each constraint in the system is expressed as a rule where a variable is forced true or false depending on the values of the other variables in the system.

One well known example of this type of problem is the 8 Queens problem where eight queens have to be placed on a chess board so that they do not attack each other vertically, horizontally, or diagonally. This problem can be represented by having one logical variable for each position on the chess board, 64 logical variables taking 128 units. The variable will be true if a queen is positioned there and false if no queen can possibly be positioned there. Typical rules are: a variable should be false if any other position which can attack it is true; or a
variable should be true if all other positions in the same row (column) are false (both these rules can be programmed using a single unit). Termination of the system is detected by any variable having both a true and false value, or by 8 variables being true. To be useful the system needs to be used as part of backtracking search controlled by the host.

The host would load the rules and their corresponding weights into the system and then initialize the system for some partial board position where one or more queens are placed on the board. During execution a number of things can happen: it may be discovered that some positions are forced to contain queens; it may be discovered (possibly after some queens are forced) that the position is unfeasible (contradictory); it may be found that no further conclusions can be drawn even though the whole problem is not yet complete. If the position is unfeasible then the host needs to backtrack in its search and reinitialize the system for some other branch of the search. If some positions have been forced but the problem is still not complete, the host can take one branch of the search from the new position, returning and trying an alternative branch if there is later failure.

What the $\Sigma$-chip system contributes is a very rapid determination of the feasibility of any partial position suggested by the host, as well as filling in any conclusions forced by the partial position. This can greatly speed a search although some searching does still need to be done. Tests with the 8 Queens problem show that almost all positions terminate after 7 cycles (175\mu secs). One advantage of the parallelism being used is that rules that are only applicable very occasionally can be usefully used. In a sequential system the time they take to be checked would probably outweigh the reduction in searching they cause but in parallel they only use up extra units and do not slow down the system. The net effect of this is often that the inclusion of many of these weak rules causes a combinatorial implosion where one of them firing causes other rules to fire in a cascade. For example, in the 8 Queens problem it is possible to eliminate duplicate solutions by inclusion of appropriate rules. These rules reduce not only the final
reported solutions but also the search conducted to find them. The simplest 8 Queens problem can be fitted onto a 4 \( \Sigma \)-chip system.

**Picture Processing - a parallel thinning algorithm**

Holt (1987) describes a thinning algorithm for black-white pictures. This is easily implemented using a \( \Sigma \)-chip system. A system of 4 \( \Sigma \)-chips appropriately interconnected can handle a 10×10 pixel picture. It is possible to scale this, so to process a 64×64 pixel picture would require a 200 chip system. The time to do one thinning operation on the whole picture is four cycles of the chip, that is, 100\( \mu \)secs. In contrast Holt using the 64×64 ICL DAP multiprocessor system required 390\( \mu \)secs to complete one thinning operation for a 64×64 pixel picture. A typical picture requires 6 thinning steps, so using \( \Sigma \)-chips a picture could be completely processed in 600\( \mu \)secs, sufficient for real time operation.

**Game of Life**

The game of life can be programmed so that 4 chip system can handle a 10×10 board, and this can be scaled, so for example, a 400 chip system could handle a 100×100 board. Each generation of the game of life takes three cycles(75\( \mu \)secs), that is, 12,500 generations a second.

<table>
<thead>
<tr>
<th>Table I. Vital Statistics of ( \Sigma )-chip.</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Current Design</strong></td>
</tr>
<tr>
<td><strong>Ideal Design</strong></td>
</tr>
<tr>
<td>Fabrication 2( \mu ) CMOS</td>
</tr>
<tr>
<td>Multiplexing (N) 100</td>
</tr>
<tr>
<td>Physical Input Lines (i) 10</td>
</tr>
<tr>
<td>Virtual Input Lines (Ni) 1000</td>
</tr>
<tr>
<td>Number Weights (iN^2) 100000</td>
</tr>
<tr>
<td>Transistors on Chip 450000</td>
</tr>
<tr>
<td>External Clock 4 MHz</td>
</tr>
<tr>
<td>Internal Clock 25 MHz</td>
</tr>
<tr>
<td>Successive Results from Unit 0.04 MHz</td>
</tr>
<tr>
<td>2500000</td>
</tr>
<tr>
<td>10250000</td>
</tr>
</tbody>
</table>
**VLSI Design**

The basic calculation to be performed by the Σ-chip is a linear weighted sum of its inputs. That is, each of its 100 outputs is a different weighted sum. Thus the chip must contain two things: somewhere to store the weights and somewhere to accumulate the sums. On the Σ-chip the weights are stored as values circulating in shift registers, this provides a compact and simple storage scheme and allows convenient access to the values while accumulating the sums. Figure 10 shows the layout of the Σ-chip, the weights in the shift registers occupy the bulk of the chip. Down one side of the chip runs space for accumulating sums and distributing the inputs. So as to avoid long signal wires driving many devices the inputs are also distributed by shift registers. Similarly the signals for the output line are placed on a shift register and moved off the chip in sequence.

The whole chip is very regular. Not only is the bulk of the chip occupied by shift registers it can also be divided into 100 identical horizontal slices. Each slice contains all the weights for one unit line and enough logic to accumulate the weighted sum of the inputs and to output the result at the appropriate time.

The parameters chosen for the current design are a compromise between the speed of the system, the size of the chip and the amount of interchip wiring. The values for the current system are summarized in Table I together with values for an "ideal" system which awaits further advances in VLSI technology. The most critical factor for programming the chip and for its usability is the total number of inputs available to each unit, this will be the product of the number of input lines (i = 10) and the degree of multiplexing (N = 100). Ideally this would be higher but it is constrained by the total number of weights which need to be stored (given by \( iN^2 \)) and the maximum size of chip which can be constructed. The number of input lines seems to be about right at 10, this is sufficient for most problems including the mesh wiring patterns needed for the Game of Life and picture processing. The amount of multiplexing is constrained both by the chip size and by the overall execution time of the system. The time for all outputs
to be computed from one group of multiplexed inputs is proportional to $N$. So the less multiplexing the less time is taken. However, less multiplexing implies more interchip wiring down to the ridiculous limit where $N=1$ and nothing has been gained.

The clock speed of the chip is limited in practice by our ability to distribute a clock signal on a PC board layout. The bulk of the chip including the adders and the shift registers are clocked at a higher rate than the external multiplexed signals. This allows the adders to be simplified and so reduces the number of gate delays in each adder and the area they occupy. Current estimates are that this internal clocking will be at 25MHz and the external signals will be clocked at 4 MHz.

Conclusions

Uncertainty is a source of computational power in neural networks, and stochastic phenomena are of fundamental importance, not “noise” as in conventional information processing. The stochastic computing perspective of regarding digital operations on random sequences as being analog operations on the generating probabilities of those sequences may be useful in the design and analysis of parallel distributed systems. The representation of potentially infinite quantities in the stochastic computer has many properties reminiscent of those of visual perception and may be a useful model of some neuro-optical processes.

The natural processes available to implement stochastic networks are those involving quantum mechanical uncertainties, and it may be that the barriers that these are seen as placing on packing densities on conventional integrated circuits should be reconstrued as an opportunity for more powerful forms of computation. Similarly, optical computing elements may offer better opportunities for the implementation of stochastic networks than they do for the emulation of conventional information processing.

Many of our limitation theorems on conventional information processing derive from assumptions of discreteness in space and time. Continuity in space and time allows for
computations that are not subject to Church’s and Godel’s limitations. However, this is a trivial result unless alternative limitations are considered, notably those of noise. Shannon’s channel capacity results apparently map continuous stochastic processes onto discrete deterministic processes in their computational power. However, there are aspects to this mapping that are worth further investigation. Stochastic information may be used in a variety of ways to test hypotheses and there are counter-intuitive results such that hypothesis A may be tested, or B, but not the conjunct of A and B. It is possible that this choice of the way in which to use noisy data is in itself an important aspect of anticipatory processes in the brain. It is suggested that consideration of continuous computational processes may be significant in analyzing the modeling capabilities of the brain and replicating them in networks.

Acknowledgements

Financial assistance for this work has been made available by the National Sciences and Engineering Research Council of Canada. We are grateful to Phil Mars, Ted Poppelbaum and Ian Witten for access to their research on stochastic computing and learning systems.

References


Figure Captions

Figure 1 Arithmetic units for unipolar representation.

Figure 2 Arithmetic units for bipolar representation.

Figure 3 Basic computing units for single wire bipolar representation.

Figure 4 Symbols for basic stochastic computing units.

Figure 5 Basic computing units for infinite representation.

Figure 6 Stochastic Sigma Pi units.

Figure 7 Networks solving Laplace's diffusion equation in stochastic computing

Figure 8 Output of stochastic neuron.

Figure 9 Basic structure of Σ chip.

Figure 10 Layout of Σ chip.
Unipolar representation  \[ x = p \quad 0 \leq x \leq 1 \]

Figure 1. Arithmetic units for unipolar representation
Bipolar representation  \( x = u - d \)  \(-1 \leq x \leq 1\)

Multiplier

Figure 2. Arithmetic units for bipolar representation
**Bipolar representation**

\[ x = p - (1-p) \quad -1 \leq x \leq 1 \]

**Multiplier**

**Weighted Adder**

---

**Figure 3.** Basic computing elements for single wire bipolar representation.
Figure 4. Symbols for basic stochastic computing units.
Figure 5. Basic computing elements for infinite representation.
Bipolar representation
\[ x = p - (1 - p) \quad -1 \leq x \leq 1 \]

Infinite representation
\[ x = \frac{p}{(1 - p)} \quad 0 \leq x \leq \infty \]

Figure 6. Stochastic Sigma Pi units.
Figure 7. Solution of differential equation.
Figure 8. Output of stochastic neuron.
Figure 9. Basic Structure of $\Sigma$-chip.
Figure 10. Layout of $\sum$-chip.