REAL BRAINS ARTIFICIAL MINDS

EDITED BY

John L. Casti
International Institute for Applied Systems Analysis
Laxenburg, Austria

Anders Karlqvist
The Royal Swedish Academy of Sciences
Stockholm, Sweden

North–Holland
New York • Amsterdam • London

Computable Functions and Complexity in Neural Networks

OMER EGECIOGLU[2], TERENCE R. SMITH AND JOHN MOODY[3]

1. INTRODUCTION

Networks of linear threshold functions (LTFs) have been employed both as models of biological neural networks [1,2], and as models of computational devices [3]. A given LTF v_i may be defined by:

$$v_i = \begin{cases} 1 & \text{if } U_i > 0, \qquad U_i = \sum t_{ij} v_j + \theta_i \\ 0 & \text{else} \end{cases} \tag{1}$$

where v_j is the output from the jth LTF; t_{ij} represents the strength of connection between the ith LTF and the jth LTF; and θ_i is a threshold value. A network of LTFs is defined in terms of the connectivities between the various LTFs. These connectivities are given explicitly in terms of the connectivity matrix $T = \{t_{ij}\}$.

An LTF may be viewed as a limiting case of a more general class of functions. One such class of functions, the so-called class of "semi-linear activation functions" [4], takes the form:

$$v_i = g(U_i) \tag{2}$$

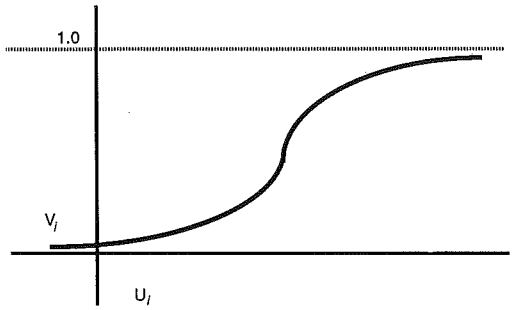
where g is a monotonically increasing, differentiable function of its argument and θ_i is termed the "bias". In many applications, g is typically a sigmoid function with left asymptote 0 and right asymptote 1 (Figure 1(a)).

We employ the terminology SAF to denote the class of sigmoidal, semilinear activation functions. Rumelhart et al. [4], for example, employ the class of logistic functions as an instantiation of the class of SAFs. These functions take the form:

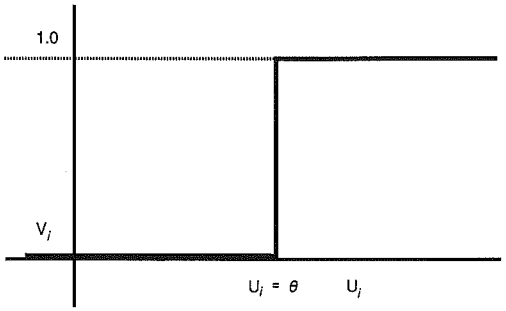
$$v_i = \frac{1}{1 + e^{-\beta \Sigma (t_{ij} v_j + \theta_i)}} \tag{3}$$

in which θ_i is termed the "bias", and is analogous to the threshold θ_i in an LTF and β is a parameter. In this instantiation, the LTF may be viewed as a limiting case of the SAF in which g is a step function (and

[2]Supported in part by NSF Grant No. DCR-8603722.
[3]Supported by NSF Grant No. PHY82-17852, supplemented by funds from NASA.



(a) General sigmoidal, semi-linear activation function



(b) Special case step function

Figure 1 General and limiting cases of sigmoidal, semi-linear activation functions

is hence not differentiable at all points of its domain, see Figure 1(b)). For many applications, networks of SAFs have come to be called neural networks. In this paper, we focus our attention on the case of neural networks that are composed of LTFs, although it will prove necessary on occasion to consider networks of SAFs.

We believe that it is important to view networks of SAFs in general, and networks of LTFs in particular, from a computational point of view. While Minsky [5] showed that networks of LTFs (and hence networks of SAFs) possess the computational power of a Turing machine, we believe it to be of particular value to adopt an approach in which we:

(a) define special classes (or "architectures") of networks of SAFs;
(b) define procedures by which a specified SAF architecture may be "programmed";

(c) examine which classes of functions are computable by a given SAF architecture and an associated programming procedure;
(d) examine the question of the computational complexity of a given class of architectures and associated programming procedures with respect to a given class of functions that they are capable of computing.

We term an approach to the study of neural networks that involves these four aspects the "computational approach" in order to distinguish it from approaches adopted by other disciplines, including biology, physics and psychology.

The outcome of such an approach is of significance in at least two major areas of scientific endeavor. In relation to the field of neurophysiological investigation, it is important to characterize the class of functions that a given model of some neural subsystem is capable of computing, and how efficiently such functions are computed, in order to know the degree to which the model "explains" neurophysiological phenomena. There is still a great deal of controversy in the neurophysiological sciences as to whether an SAF is an adequate model of a neuron. There are researchers who believe that neurons have much greater computing power than SAFs, with some researchers likening them more to microprocessors than to SAFs. Others view ensembles of SAF-like units as the basic computational unit in biological neural nets.

It is premature, however, to make final judgments concerning such issues, and it is important to continue to study computational (and other) issues concerning neural nets of SAFs, since they offer a model that is appealing in both its simplicity and in its computational power. Only when we have found a relatively complete characterization of a large class of devices,

$$\{D = (A, P) |\ A \text{ is a network of SAFs, } P \text{ is a programming procedure}\},$$

both in terms of the classes of functions computable and the complexity of the devices with respect to the classes of functions computable, and only when the outcome of such research is related to the computational abilities of biological neural networks, will we be in a position to make adequate judgments concerning the biological applicability of such classes of devices.

A similar set of questions is even more relevant in the investigation and design of specialized computational devices in which massive parallelism appears to be either required or at least highly desirable in order to compute certain functions in reasonable amounts of time. There are many difficult computational problems, however, such as visual processing and

speech understanding, in which the classical approach of employing sequential devices and standard analysis of sequential algorithms, has so far failed to yield major results in terms of systems able to solve the problems in reasonable amounts of time. Furthermore, the technology available for constructing ever-faster sequential devices is now approaching a state in which the laws of physics are appearing as bounds on the speed of such devices.

On the other hand, approaches to such problems that involve the use of both massively parallel systems of relatively simple processing units and programming procedures based on "learning" appear to offer significant promise for major advances in our ability to construct devices that are able to solve these problems in acceptable amounts of time. Such devices will probably prove to be of major significance, for example, in the construction of "intelligent" mobile robots. Hence it is again important to characterize the class of functions that a given device $D = (A, P)$ is capable of computing, and how efficiently such functions are computed, in order to design and construct specialized devices for solving difficult problems.

We believe, therefore, that it is important to emphasize the need for a theoretical basis for the models of computation described in this paper. Such a basis, which may be provided in terms of the approach to networks of SAFs that we have stated above, permits a more rational approach both to the construction of models of neurophysiological phenomena and to the design and construction of specialized computing devices that require massive parallelism. This point of view is particularly significant in view of the recent profusion of empirical results concerning such computation.

Special classes of SAF networks

It is evident that one may define many special classes of networks of SAFs by placing constraints on both the nature of the SAFs involved and the nature of the network connecting the SAFs. For example, we may place constraints on:

(a) the matrix of connectivity coefficients $T = \{t_{ij}\}$;

(b) the response of the computing units in the system. In the case of SAFs in general, this involves specifying the class of SAFs to be used, while in the case of LTFs, this involves specifying the set of threshold values $\{\theta_i\}$. Furthermore, one may specify whether the response of the unit is deterministic, as in the case of LTFs, or stochastic, as in the case of some classes of SAFs;

(c) the protocols that determine when a given LTF will "compute";

(d) the relative speeds of computation within the SAFs and the relative speeds of communication between the SAFs.

We will term any such set of constraints A the "architecture" of the device. We now briefly characterize a few of the more significant types of constraint that have been employed in the analysis of networks of SAFs.

Concerning the matrix T of connectivity values, we may classify different topologies of connectivity among the SAFs (in terms of the set of pairs of SAFs for which $t_{ij} = 0$). It is of particular interest to distinguish between locally connected systems and more globally connected systems as well as between systems in which feedback between SAFs is permitted (directly or indirectly) and systems in which only feedforward is permitted. It is also of interest to classify systems on the basis of the values of the non-zero t_{ij} and in particular to distinguish systems on the basis of whether or not the t_{ij} are symmetric. The nature of the matrix T, as we note below, may be quite important in determining both the class of functions computable by a network of SAFs and on the complexity of computation.

Concerning the protocols that determine when a given unit will compute, we may differentiate continuous response (as in the case of many SAFs investigated) from discrete response (as in the case of networks of LTFs). For networks of LTFs in particular, we may distinguish between synchronous and asynchronous systems. A synchronous system is one in which each of the LTFs execute simultaneously, typically driven by some clock in the system. In asynchronous systems of LTFs, the units execute at different times. There are various protocols that may be used to determine when a given LTF will execute. For example, each LTF may execute at times that are randomly selected by the unit itself, or each LTF may execute at times that are determined by the execution of at least one unit that is connected by a non-zero t_{ij} to the unit in question. As we note below, the execution protocols may be quite important in determining which functions a given architecture of device is capable of computing.

Classes of functions computable by networks of SAFs

All of the devices that we consider in this paper may be viewed as mapping a binary string of length n into a one-parameter family of binary strings of varying length. In the case of networks of LTFs, however, an initial string may be mapped into a sequence of strings of different lengths. The length of the sequence may be finite or infinite. If the sequence is of finite length and terminates with a binary string of length

m, then the computation is convergent and the device computes a function from binary strings of length n to binary strings of length m. In particular, the lengths $m = 1$ and $m = n$ are cases of special interest, with the case $m = 1$ corresponding to the class of boolean functions.

While a given set of constraints on a network of SAFs defines the class of functions computable by the device, it will in all probability prove to be an intractable problem to determine such a class in the general case. From the experience gathered in the cases examined below, a more tractable task involves determining the class of functions computable using a specific architecture A in conjunction with a given procedure P. The procedure P may be employed, for example, to specify the matrix $T = \{t_{ij}\}$ and the set of biases $\{\theta_i\}$. The procedure P will, in general, depend on the class of functions that one wishes the device to compute.

It is important to emphasize that one should attempt to specify the class of functions computable by any device $D = (A, P)$ in terms of the pair (A, P), rather than purely in terms of the architecture A of the device.

Programming networks of SAFs

It follows that a question of major significance relating to any given class of SAF networks, and in particular to very large networks, concerns the procedures that may be employed for programming the device in order to compute a specific function. As noted above, such programming generally involves the choice of the matrix T and a set of biases $\{\theta_i\}$. There are some cases (see below) in which the matrix T can be written down upon inspection or after some relatively simple analysis of the function to be computed. There are other cases in which it is apparently easier to invoke some (iterative) learning procedure P by which the device comes to learn the nature of the matrix T from examples of the class of functions to be computed. Questions concerning the existence and construction of powerful, iterative learning procedures for systems of SAFs are of major significance from a computational point of view.

Computational complexity and networks of SAFs

It is important to emphasize the need for an understanding of computational complexity issues for networks of SAFs. The formalization of this concept is not only interesting from a purely theoretical point of view, but also important in terms of applications. For example, characterizing the computation of certain classes of functions on networks of SAFs in terms of an appropriate measure of computational complexity has practical implications for the design and production of analog

VLSI that may be used for visual processing and speech understanding in robotic devices.

With the advent of the sequential computer, the analysis of the complexity of computations and the study of hierarchies of language classes brought to the fore a formal theory of complexity and a correspondence between complexity classes of languages and functions computable by restricted versions of the Turing Machine (TM) model. The interpretation of decision problems as languages over a finite alphabet under a suitable encoding scheme allows such problems to be viewed as the computation of the characteristic function of the language of "yes" instances of the problem. Futhermore, the identification of classes of functions that are in principle computable by TMs but that, from a practical point of view, seem to require unacceptable amounts of time and/or space for their computation, initiated a fruitful study into the separation of decision problems into polynomial and exponential classes, as well as a distinction between deterministic and nondeterministic computations.

The simplest class of languages, regular expressions, were shown to capture the power of McCulloch–Pitts nerve nets by Kleene [6]. The corresponding, restricted TM in this case is the finite automaton (deterministic or nondeterministic).

The class of machines with a pushdown stack and finite control, like finite automata, corresponds in the deterministic case to deterministic context-free languages and in the non-deterministic case to context-free languages. Allowing the TM only an amount of tape linear in the length of the input string yields context-sensitive languages. Further up in the hierarchy are recursive functions (corresponding to recursive sets) and the full computing power of unrestricted TM (deterministic or nondeterministic) defines the class of partially recursive functions (corresponding to recursively enumerable sets and phrase structure grammars).

Certain other restrictions on the TMs, however, do not reduce its computing power. If the input alphabet is not restricted, for instance, recursively enumerable sets can be recognized by a TM with only three states. Similarly, TMs with only a binary alphabet and a single tape can be shown to be as powerful as the unrestricted model.

Complexity issues in the case of networks of SAFs are more difficult to define precisely. While the number of iterations required by a given device $D = (A, P)$ to converge to a solution, as a function of the length of the input string, is an obvious measure of "time" complexity, it may prove to be an extremely difficult quantity to analyze in general. Furthermore, in the instances of networks of SAFs that are programmed with the use of learning procedures, any realistic time-complexity measure for such a device should involve the running time of the learning

procedure itself. This is because even in the case of simple LTF devices for which there exist effective learning algorithms, the procedure may require exponential time in the size of the input string to terminate [7]. Similarly, learning algorithms that employ probabilistic hill-climbing algorithms may require exponential time in the worst case. It is possible to define simulation complexity of a network as the time required to simulate a network computation on a serial device. Note that this measure is dependent on simulation techniques.

The question of the spatial complexity of a given device $D = (A, P)$ with respect to a given class of functions is also a problem of great interest. In the case of convergent computations (with the device mapping an n-bit string into an m-bit string) it is natural to use as a measure of computational complexity the manner in which the number of SAFs required to compute a given class of functions increases as the length of the input string to the device. Since the complexity of a network in terms of connections required is dependent on the particular implementation employed, spatial (or hardware) complexity for networks is similar to the time complexity of sequential algorithms. With this analogy in mind, we may apply the usual distinction between exponential and polynomial hierarchies of sequential algorithms to the hardware complexity of neural networks.

The classes of device considered

In the present paper, we examine a small set of classes of SAF networks, concentrating in particular on networks of LTFs, and associated programming procedures. We examine each in terms of the computational approach stated above. The devices that we examine include:

(a) the perceptron (LTFs);
(b) the generalized feedforward device (SAFs);
(c) the Hopfield device (LTFs and SAFs);

We will also briefly examine several variants of the three classes of networks listed above, and investigate the various computational questions in relation to such devices.

2. Special classes of networks of SAFs

Perceptrons

(a) Architecture

In the classification of neural networks in terms of their architecture, the simplest and probably the best understood LTF device is the perceptron. In addition to the simplicity of its dynamics, the architecture

of the perceptron does not include multilayers or feedback. Due to the limited nature of the computing power of the processing units and the simplicity of the topology of the device, an essentially comprehensive analysis of the class of functions that they are capable of computing is available.

Perceptrons in their simplest form were introduced by Rosenblatt [2]. In their neuron-like assembly structure, they are similar to McCulloch–Pitts nerve nets [1]. Although they were proposed to account for, and to pave the way to an understanding of, simple aspects of nervous activity, they were not meant to be precise and faithful models of neurophysiological systems.

As a mathematical model, a perceptron ψ is an extremely simple network of LTFs, consisting of three basic components:

(i) A matrix of binary inputs $x_1 x_2 \cdots x_n$ (or "retina");
(ii) A set of LTFs $v_1, v_2, \ldots, v_N$ (or predicates in the most general case) with fixed connections to subsets of the retina ("feature detectors");
(iii) An LTF with modifiable connection weights to these predicates (a "decision unit").

Thus the perceptron model is characterized by feed-forward connections and a single layer of modifiable weights. In the simplest case where $n = N$ and $v_i = x_i$, the processing units themselves can be viewed as the input layer. Then ψ becomes a single LTF. This is the classical model of an n-input neuron or Rosenblatt's simple perceptron.

In the general case, each processing unit v_i of the perceptron may be viewed as a boolean function that depends on some fixed small subset of the retina. The activation value of these processing units is then the value of the function v_i, which is either 0 or 1. The device responds to an input I with the output of $\psi = 1$ or $\psi(I) = 0$, depending on whether

$$\sum v_j(I) t_j > \theta$$

Here t_j is the strength of the connection between the unit v_i and the decision unit, and θ is the threshold of the device. Thus ψ is a special type of a boolean function. It separates the image of the vector valued function $\mathbf{v} = (v_1, v_2, \ldots, v_N)$ defined on the retina by means of the N dimensional hyperplane defined by

$$\{\mathbf{x} \in E^N \mid \mathbf{x} \cdot \mathbf{t} = \theta\}$$

(b) Programming the perceptron

Supervised learning is possible for the perceptron by using a variant of

the Hebbian rule [8] to adjust the weights t_i of the synaptic connections. The algorithm that achieves this is the *perceptron convergence procedure* of Rosenblatt [2]. More precisely, suppose that f is a boolean function defined on the retina, which is 1 or 0 depending on whether or not $\sum x_i c_i > \theta$ for some vector c and a constant θ. Put $A = f^{-1}(1)$ and $B = f^{-1}(0)$. Then the sets A and B are *linearly separable.* It is then possible to construct a perceptron with processing units given by $v_i = x_i$ that computes f in an iterative manner as follows: starting with an arbitrary initial weight vector w, and a pattern I for which $\psi(I) \neq f(I)$, the value of the weight w_i is increased if $I \in A$ and $v_i = 1$, and decreased if $I \in B$ and $v_i(I) = 1$ for $i = 1,2,\ldots,N$. Given that the two sets A and B are linearly separable, then cycling through the patterns I will result in a vector of weights w for which $\psi \equiv f$ in a finite number of steps.

(c) Functions computable by the perceptron

By the perceptron convergence theorem, only linearly separable functions are computable by simple perceptrons. On the other hand, any boolean function f can be computed by a perceptron in trivial ways if we allow a larger set of predicates v_i. For instance, we can take one of the v_is to be f itself. Alternately, setting for each subset S of the retina

$$v_S = \prod_{i \in S} x_i \prod_{j \in S^c} (1 - x_j)$$

we can trivially define a perceptron that computes f by linear inequality

$$\sum_S v_S(\cdot) f(S) > 0 \qquad (4)$$

Thus in theory, if no restrictions are put on the power of the v_is, the device becomes boolean complete. Clearly, in this generality, the original idea of parallelism is lost, and in the above cases the global properties of the given function are not computed from local information.

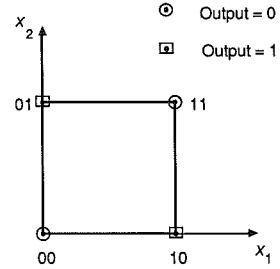
Therefore, the question here is not one of pure existence but: what classes of functions are computable given some restrictions on the device and how economically can this be done with the given restrictions? The economy in question here can be considered in terms of the complexity of the connections, the computing power of the individual v_is, and the magnitude of the weights (and the time taken by the learning algorithm involved). More specifically, we would like to restrict the total number of connections of LTFs to a polynomial in the size of the input binary string. Note that the above construction for ψ to compute an arbitrary f requires an exponential number of processing units and connections

in the size of the input pattern. An interesting issue then is the class of functions computable given such restrictions on the device.

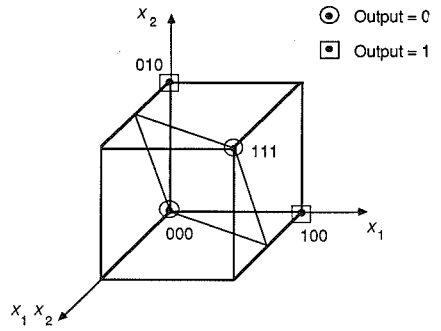
A case in point for the limitations of the simple perceptron is the computation of the logical function XOR. For the XOR function, two of the least similar input patterns are required to generate identical outputs as given below:

$x_1 x_2$	*output*
00	0
10	1
01	1
11	0

Geometrically, it is easy to see that the two sets of input vectors {00,11} and {01,10} are not linearly separable (Figure 2(a)).



(a) XOR in 2-Space



(b) XOR in 3-Space

Figure 2 XOR function in 2-space and 3-space

Algebraically, the existence of a simple perceptron ψ to compute XOR is equivalent to the existence of a solution to the system of inequalities

$$0 \leq \theta$$
$$t_1 > \theta$$
$$t_2 > \theta$$
$$t_1 + t_2 \leq \theta$$

which is clearly impossible. Note that by the same token a simple perceptron is also unable to compute the function XOR.

If we are allowed to use higher order predicates, the XOR problem has an easy solution. For example, searching for a second order predicate of the form $x_1 t_1 + x_2 t_2 + x_1 x_2 t_{12} > \theta$ requires the existence of a solution to the system of linear inequalities

$$0 \leq \theta$$
$$t_1 > \theta$$
$$t_2 > \theta$$
$$t_1 + t_2 + t_{12} \leq \theta$$

One possible solution is given by $\theta = 0$, $t_1 = t_2 = 1$, and $t_{12} = -2$. This procedure corresponds to lifting the given function to three-dimensional space via the embedding given in Figure 2(b) with coordinate axes $x_1, x_2, x_1 x_2$. Viewed in this manner, XOR becomes linearly separable in accordance with (4). However, in this case the resulting perceptron that can learn to compute XOR is no longer simple and requires a "hidden unit".

Generalizing from the example of XOR, it it possible to define a hierarchy of the class of boolean functions in terms of computability by perceptrons by considering the least dimensional space required in which a boolean function f becomes linearly separable. This and related concepts were made precise by Minsky and Papert [7].

(d) Issues of complexity for the perceptron

An in-depth analysis of the computing power of the general perceptron was performed in the sixties by Minsky and Papert [7]. This extensive analysis spans restrictions on the class of functions v_i and the topology of the device in terms of the connectivities, along with questions of magnitude of the weights required and the convergence rate of the learning procedure for a wide variety of functions. The following notation proves of value when we briefly discuss some of their results concerning the power of perceptrons in terms of order. For a subset S of the retina, let

$$\chi_S = \Pi_{i \in S} x_i$$

Then by (4), any perceptron ψ has a representation of the form

$$\sum_S \chi_S(.) w_S > \theta \tag{5}$$

In this representation, the size of the largest subset S of the retina for a given ψ for which the weight w_S is nonzero may be called the *order* of the perceptron. This can be thought of as the maximun fan-in of the processing elements of this ψ.

For predicates that express geometric properties of plane figures, it is useful to view the retina as a two-dimensional lattice, and for some predicates as a torus. Invariance of ψ under a group of transformations of the retina translates into a condition on the equality of the coefficients w_S and w_T in (5), whenever the subset S is a translate of a subset T by an element of the group. This is the group invariance theorem of Minsky and Papert.

As an example, given that the predicate ψ is of order k and is invariant under the full symmetry group on the retina, then at least $\binom{n}{k}$ processing units must have a fan-in of size k. For the generalization of XOR to n dimensions (the parity function), Minsky and Papert show further that the coefficient w_S must be nonzero for *every subset* S of the retina. As a consequence, the number of nodes plus the number of connections of a perceptron that can compute the parity of an n-bit string can be shown to be $\Omega(n2^n)$. Furthermore, the weights of such a perceptron are also exponential in the size of the input in the sense that the ratio of the largest one to the smallest one must be $\Omega(2^n)$.

Another function that cannot be computed by perceptrons of constant order, in particular by simple perceptrons, is *connectedness*. The question here is whether or not there is a path connecting any two points of a given set S that lies entirely in S. Minsky and Papert showed that any perceptron that computes connectedness must have order $\Omega(\sqrt{n})$. Furthermore, the total number of processing units and connections must again be exponential in n.

Note that as soon as we know that a class of boolean functions is computable by a perceptron of order k for some constant k, then the total number of processing units and the number of connections required is a function of the input size $O(n^k)$.

The following results concerning the upper bound on the orders required to compute the given geometric predicates, furnish some interesting examples of functions computable by perceptrons that are polynomial in size. Here I is a geometric pattern on the retina:

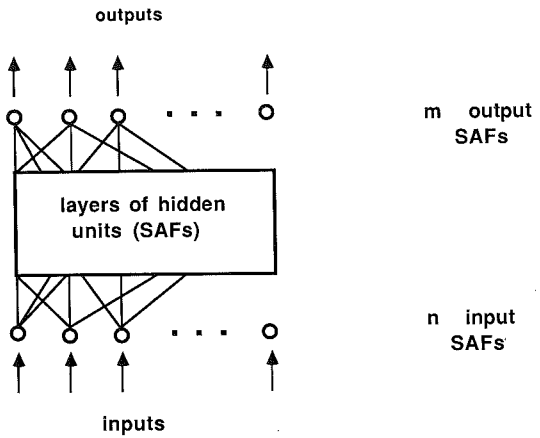


Figure 3 A multilayer network of SAFs involving hidden units

(1) Counting predicates "*Is the size of $I < m$?*", "*Is the size of $I > m$?*" are of order 1;

(2) "*Is I convex?*" is of order ≤ 3;

(3) "*Is I a solid rectangle?*", "*Is I a hollow rectangle?*", "*Is I a solid square?*", "*Is I a hollow square?*" are all of order 3;

(4) "*Is the Euler number of $I < m$?*" has order ≤ 4, "*Is the Euler number of $I = m$?*" has order ≤ 8;

(5) "*Is I a circle?*" has order 4;

(6) If the retina is a horizontal strip, then "*Does I have a symmetry under reflection about some point?*" has order ≤ 4;

The generalized feedforward device

(a) Architecture

While Minsky and Papert [2] discussed generalizations of the perceptron to devices with more than two layers of LTFs, they were pessimistic about whether it would be possible to discover a powerful analogue to the perceptron learning (or "programming") procedure. There has been a recent resurgence of interest in multilayer machines, and one outcome of this research has been the architecture and the associated "programming" procedures investigated by Le Cun [9], Parker [10] and Rumelhart et al. [4], which we term the generalized feedforward (GFF) device.

The architecture of the GFF device involves a set of n input units and a set of m output units. The device also possesses a set of intermediate layers of units (known as "hidden units"), with each layer containing an arbitrary number of units (see Figure 3).

It is course possible to constrain both the number of levels and the number of hidden units at each level. The device is feedforward in its operation, with communication between units occurring only from units at some lower level to units at a higher level. It is possible to constrain the degree of localness in the feedforward connections between levels. The machine, like the perceptron, is essentially synchronous in its execution protocol, although subtle timing considerations may be important when conections between units span several layers.

The units in the GFF device are not the LTFs defined in (1), but are SAFs. For example, Rumelhart et al. [4] employ a member of the class of logistic functions as the function g in (2). Such units were chosen for reasons of programmability discussed below. As noted above, however, the LTF is a limiting case of the logistic SAF and it is possible to approximate an LTF to any required degree of accuracy by varying the parameter β in the logistic function (3). This fact, and evidence presented below, suggests that many of the important results of Rumelhart et al. may carry over to systems in which the units are LTFs.

(b) Programming the GFF device

Minsky and Papert [7] noted that, given a sufficient set of hidden units and the appropriate matrix $T = \{t_{ij}\}$, it is possible to find a device that performs any mapping from input strings to output strings. The main problem of concern to them was the existence and construction of a programming procedure P by which the device could be configured to compute some member of a class of input–output pairs. Rumelhart et al., however, have investigated a programming procedure by which their architecture "learns" to compute a large subclass of functions mapping n-bit strings into m-bit strings.

One successful programming procedure proposed for GFFs is termed the generalized delta rule [4,11]. It is an essentially iterative procedure that uses examples of the functions to be computed in order to modify the elements t_{ij} of the matrix T. The procedure is based upon a simpler procedure known as the Widrow–Huff or "delta" rule [4,11]. The generalized delta rule operates in two phases. During the first phase, an input is presented to the system and propagated through the network and an output value is computed for each unit. In the second phase, differences between actual output values and desired output values are propagated backwards through the system, starting from the output units. During this backwards propagation the t_{ij}s are modified according to the generalized delta rule.

The generalized delta rule takes the form:

$$\Delta_p t_{ij} = \eta \delta_{jp} o_{ip} \tag{6}$$

where the change in the connection strength t_{ij} for the pth input–output pair is given by the product of some coefficient η, an error term δ_{jp} (which depends in form on whether the unit is an input/output unit or an internal unit) and a term o_{ip}, which is the output pattern produced on the presentation of the pth input pattern to the ith SAF. The error terms involve the derivative of the SAF, and for this reason Rumelhart et al. employed the logistic SAF rather than the LTF, since the derivative of the LTF becomes infinite at the point of inflection.

Hence the GFF device involves a procedure P that is based on a process of steepest descent on a surface in the space of the connectivities t_{ij}. This surface represents the error measure. An important theoretical contribution made by the investigators was to show that the derivatives involved may be computed in an efficient manner. Furthermore, empirical results indicated that the problem of the procedure becoming stuck in local minima is irrelevant in a large variety of learning tasks. Another empirically based fact is that the time to reach a solution in the learning procedure nearly always decreases with an increase in the number of hidden units.

Finally we note that since for every network with feedback there is a corresponding feedforward network with identical behavior over a finite period of time [7], it is possible to extend the technique to networks involving feedback. Hence the procedure is able to learn sequences.

(c) Functions computable by the GFF device

It is presently unclear as to the set of functions computable by the GFF devices, either in general or in terms of the subclass of GFF devices in which the logistic function response approaches the limit of the step function response of an LTF. There are no mathematical proofs concerning the class of "programmable" functions under the GFF learning procedure, and all conclusions are based upon a large number of computational experiments carried out, for example, by Rumelhart et al. [4]. As of the present writing, however, the device has yet to fail in this regard in any significant manner.

The functions that the GFF device has learned to compute include, among others:

(1) the XOR (exclusive or) function;
(2) the parity testing function;
(3) the encoding function;
(4) the symmetry testing function;

(5) the binary addition function;
(6) the negation testing function;
(7) the function for discriminating between alphabetic characters;

In Figure 4 we illustrate two of the solutions to the XOR function that were learned by the GFF device [4]. The two solutions correspond to different constraints on the topology of the connections between the units of the device.

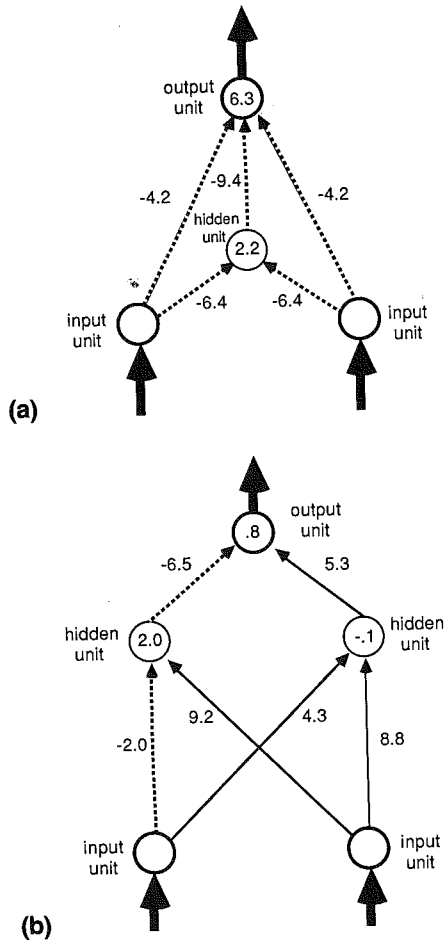


Figure 4 Two solutions to the XOR function problem as learned by the GFF device (after Rumelhart et al. [4])

In Figure 4(a), for example, thesolution represents a case in which direct connections arepermitted between input and output units.

Since many of these functions are not computable by the simple perceptron, it is of interest to enquire as to the reasons for the increased computational power of the GFF device. An important point to note about devices without hidden units, i.e., those composed of only an input and an output layer of units like the perceptron, is that they map the input patterns directly into the output layer. In such devices, there is no internal representation of the inputs, and thus similar input patterns tend to generate similar output patterns. This has the advantage of enabling generalizations from a limited set of input samples by way of a *smoothing out* effect, but prevents the device from learning arbitrary mappings in an economical manner.

Such limitations of the perceptron do not apply to the more general GFF networks. If layers of hidden units are allowed between the input and the output units, then the model gains enough power to generate representations of the input space internally, thereby acquiring the ability to allow the computation of interesting classes of functions. As noted below, for example, GFF networks can compute the parity function with a reasonable number of processing units.

(d) Issues of complexity for the GFF device

Since most of the examples examined by Rumelhart et al. involved feedforward systems, the natural measure of complexity to use is the rate at which the required number of hidden units and connections between such units increase with the number of input and output units. For several of the problems examined by the investigators, this measure of complexity is well-defined. However, the minimal number is a function of the architecture chosen, since one may constrain connections from spanning more than one level of hidden units.

We now provide some examples of these measures of space complexity. For the parity testing function (of which the XOR function is a special case with two inputs), with no direct connections between the n input units and the output unit permitted, a minimum of n hidden units and n^2+n connections are required. For the symmetry testing function with n input units and one output unit, only two hidden units and $2n+2$ connections are required. The encoding function with n input units and n output units requires $\log_2 n$ hidden units in one layer. Finally, for the binary addition function, with $2n$ input units and $n+1$ output units, only n hidden units are sufficient.

The Hopfield device and generalizations

(a) Architecture

The architecture of the basic Hopfield device and its generalizations differs from the feedforward networks discussed previously. In particular, a restricted form of feedback plays an essential role in the computational dynamics of Hopfield devices. In Hopfield's original formulation [12], the device consists of a fully connected network of units, and computations are accomplished by all units executing in parallel.

A critical constraint on the Hopfield architecture is that the connection matrix $T = \{t_{ij}\}$ is required to be symmetric ($t_{ij} = t_{ji}$). This symmetry, together with the asynchronous execution protocol, allows the computational dynamics to be described as a relaxation process in which a Lyapunov (or "energy") function E is minimized. When the computational units in the network are modeled as LTFs or SAFs, the energy function is necessarily quadratic. Hence, the Hopfield network is able to compute classes of functions that may be characterized in terms of the minima of a certain class of quadratic forms.

A number of alternative architectural constraints have been imposed on Hopfield devices. These constraints mainly concern the nature of the response function of the computational units. Three alternatives have been studied extensively:

(i) The basic device studied by Hopfield [12] employs deterministic LTFs (1) executing in an asynchronous mode. Typically, the execution of each unit occurs randomly in time. The energy function E may be expressed as:

$$E = -\left(\tfrac{1}{2}\sum t_{ij}v_iv_j + \theta_i v_i\right)$$

It is easy to show that the dynamics given by (1) minimize E. First, we note that U_i in (1) can be written as

$$U_i = -dE/dv_i$$

and that the change in energy can therefore be written as

$$\Delta E = -U_i \cdot \Delta v_i$$

With the LTF update rule of (1),

$$\Delta v_i = \tfrac{1}{2}(\sin(U_i) + 1)$$

it is clear that E will decrease monotonically as the LTFs execute. Since E is bounded, a state of minimum energy will be reached.

(ii) Networks in which the units are similar to LTFs but with a stochastic response function, have been employed in the so-called Boltzmann machine [13]. The computational dynamics of such systems differ from those of a network of LTFs in that the response of the ith unit v_i is a probabilistic function of the input U_i. The probability distribution function g is typically sigmoidal and involves a scaling parameter (or "temperature") T. A common choice for this distribution is the logistic function:

$$g(U_i) = \tfrac{1}{2}(1 + \tanh(\beta U_i/2)) \qquad \beta = 1/T$$

The deterministic case (i) may be viewed as a limiting case of the stochastic case (ii) in which $T = 0$.

Such systems have been extensively studied by physicists, because of their relationship to Ising spin systems at finite temperature [3,14–18]. In many computations employing networks of stochastic LTFs, including all those implemented as Boltzmann machines, a simulated annealing procedure [19,20] is employed, whereby the temperature T is slowly lowered as the computation proceeds. The annealing procedure commences at an initially high temperature, allowing the system to find good approximations to the global minimum of the energy function E. As the temperature is lowered toward zero, these approximate solutions tend to converge to the global energy minimum provided that the annealing schedule is slow enough. Annealing that proceeds faster than a well specified schedule [21] will result in the network finding a local, rather than a global minimum of the energy.

(iii) Hopfield devices have been studied in which the computational units are SAFs [22]. The response functions of these units are deterministic and continuous in time, with the states of all units evolving simultaneously. If the response function g is taken to be the logistic function (3), as in the case of the GFF device, β represents the "gain" of a unit, rather than an inverse temperature. A unit g_i can be thought of as an operational amplifier (op amp). A network can then be realized as an analog electronic circuit comprised of op amps connected by resistors with conductances t_{ij} and biased by input currents θ_i (see Figure 5).

The appropriate circuit equations are a set of coupled differential equations, which express the conservation of current:

$$C_i\left(\frac{dU_i}{dt}\right) = \sum_j t_{ij} v_j - \frac{U_i}{R_i} - \theta_i$$

$$U_i = g_i^{-1}(v_i)$$

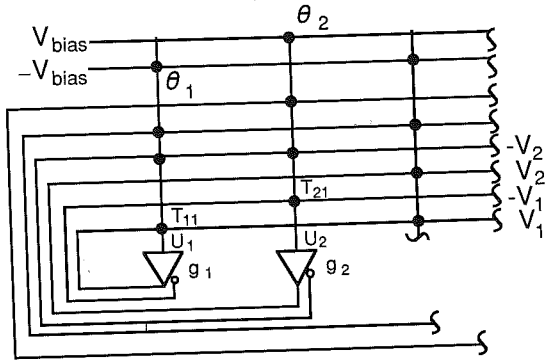


Figure 5 Analog implementation of a continuous Hopfield device

Here U_i represents the input current to op amp g_i, C_i the input capacitance, and R_i the total input resistance. The total input resistance R_i depends upon the op amp input resistance ρ_i and the connections t_{ij}:

$$\frac{1}{R_i} = \frac{1}{\rho_i} + \sum_j t_{ij}$$

$$t_{ij} = \frac{1}{R_{ij}}$$

Experimental VLSI implementations described by these equations have already been constructed.

As in the case of the basic Hopfield device (i), these analog networks may also be characterized in terms of a quadratic energy function [22]:

$$E = -\tfrac{1}{2}\sum_{ij} t_{ij} v_i v_j + \sum_i \left(\frac{1}{R_i}\right) \int_0^{v_i} g_i^{-1}(v)\, dv - \sum_i \theta_i v_i$$

A proof similar to that given above shows that the circuit equations guarantee convergence to a stable minimum energy, i.e.:

$$\frac{dE}{dt} \leq 0$$

$$\frac{dE}{dt} = 0 \rightarrow \frac{dv_i}{dt} = 0 \;\forall\; i$$

The reader is referred to Hopfield's original work [12,22] for a beautiful, and more detailed, discussion.

In computational applications, the state of a network of SAFs is given a boolean interpretation. For example, any output greater than 0.9 may

be interpreted as 1 and any output less than 0.1 may be interpreted as 0, with intermediate states being left as indeterminate. In the infinite gain limit $\beta = \infty$, the SAFs become LTFs. This fact is sometimes used in analog networks to enforce a discrete, boolean representation as a calculation converges. The process of slowly increasing the gain during a computation is analogous to the process of simulated annealing in the stochastic, discrete case (ii).

As a concrete example, two op amps may be connected with mutual inhibition to form a bistable device known as a flip–flop. For finite gain, the attractors or energy minima [0,1] and [1,0] will not be precisely at the corners of the two-dimensional state space. Rather, they will lie on the interior of the space. Figure 6 (adapted from ref. 22) shows the energy contours and "down-hill" gradients for such a system. If the op amp gain is increased, the two attractors will move closer to the corners; if the gain is decreased, they will move closer to the center and eventually coalesce.

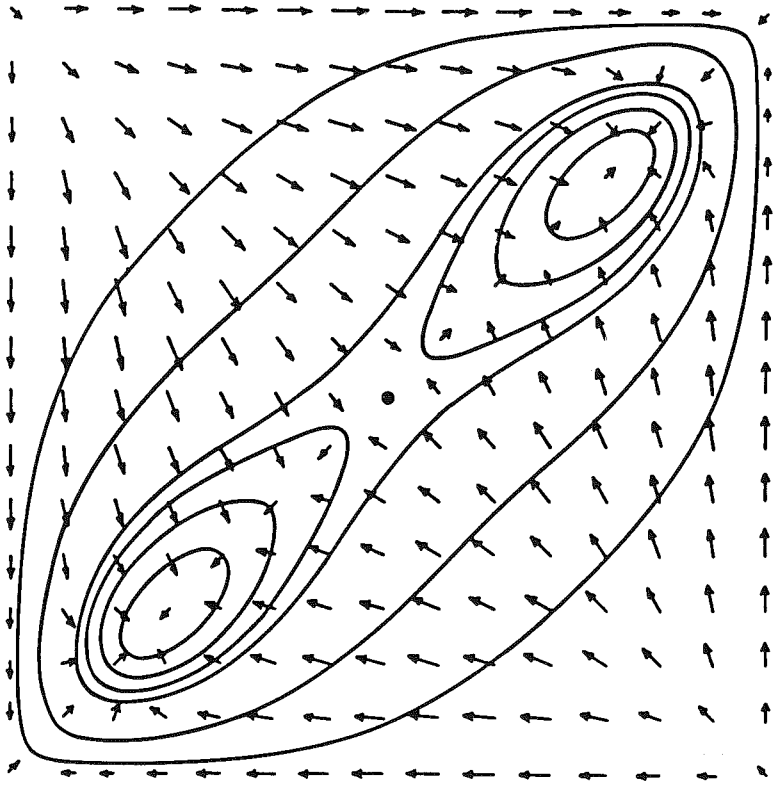


Figure 6 The flow field and E-contours for a bi-stable circuit of two amplifiers (flip–flop)

(b) Programming the Hopfield device

Programming a Hopfield device consists of first choosing a representation for the problem of interest (i.e., assigning a meaning to the state of each unit i) and then specifying the pattern of connections t_{ij} and input biases θ_i.

The simplest and most elegant approach involves selecting a quadratic energy function $E(v_1, \ldots, v_n)$, which represents the constraints on the optimization problem. One may then extract the connections t_{ij} and biases θ_i from the energy function. Alternative approaches determine the connection weights and biases directly, without reference to an energy function, via various "supervised learning" or "encoding" algorithms.

(c) Functions computable by Hopfield devices

As noted above, the class of functions computable by a Hopfield device may be defined implicitly in terms of the minima of a symmetric quadratic form. It is therefore quite difficult to provide an explicit characterization of this class of functions. Hence we discuss three specific examples of functions computable by Hopfield devices. Each example corresponds, respectively, to a function that is computable by one of the three variants of the Hopfield device described above. These examples include associative memory functions, the XOR function, and functions that represent solutions to the traveling salesman problem.

(i) Associative memory

Associative memory was the first application of Hopfield devices. Hopfield's original architecture used symmetric T and asynchronously executing LTFs [12], although associative memories employing SAFs have since been studied extensively and implemented in analog VLSI.

The task of associative memory is to retrieve a correct and complete memory vector v_i^s from a set of stored memories when provided with an incomplete or imperfect cue for that vector. It is easy to write down an energy function for a Hopfield device that has local minima corresponding to a set of m uncorrelated stored memories v^s, $s = 1, 2, \ldots, m$. One such energy function is given by:

$$E = -\sum_s \left(\sum_{i=1}^{n} (2v_i^s - 1) \cdot (2v_i - 1) \right)^2$$

The connections and biases correspond to the quadratic and linear terms in E; the resulting connection matrix T is given by the Hebbian [8] outer product rule:

$$t_{ij} = -\sum_s (2v_i^s - 1) \cdot (2v_j^s - 1)$$

A number of more complicated encoding or learning algorithms have been applied to the Hopfield associative memory, such as the adaline, geometric, pseudo-inverse, and hetero-associative rules [9,16,23–25]. These tend to provide better network performance in terms of capacity and/or reliability of recall, depending upon the correlations contained in the stored vectors and the order in which vectors are learned. One approach, which is particularly interesting because of its generality, is the so-called master/slave formalism due to Lapedes and Farber [26] in which the t_{ij}s are determined by minimizing an energy function with respect to a fixed set of stored memories $\{v^s\}$:

$$E(T) = \sum_s \sum_{i=1}^{n} (v_i^s - t_{ij} v_j^s)^2$$

The t_{ij}s calculated in this manner are in general assymetric. By placing restrictions upon the t_{ij}s one can use the master/slave approach to program more general architectures. In fact, Lapedes and Farber have shown that the GFF back-propagation algorithm is a special case of master/slave in which the connections t_{ij} are restricted to be feed-forward only.

A very different kind of associative memory based upon internal representations (IRs) has recently been proposed by Baum et al. [27]. Although originally intended for feed-forward networks, it is possible to generalize IRs to networks with symmetric connections. Networks that utilize IRs offer much greater capacity than networks based upon the outer product rule. They are also more easily programmed. An internal representation basically labels each stored memory. In standard digital technology, such labels are called addresses. For associative memory, a unary or a sparse, distributed representation is used. It is desired that the label representation be sparse in order to avoid the interference between memories, which is inherent in the outer product rule.

Memories are stored as connections between the internal representation units R^α and the standard memory vector units v_i. The energy function for such a memory can be expressed in a general form as:

$$-E = \sum R^\alpha S^{\alpha\beta} v_i^\alpha v_i + \text{quadratic constraints on } R^\alpha$$

A particular internal representation is chosen by specifying the sparse representation matrix $S^{\alpha\beta}$ and a set of appropriate constraint terms.

(ii) Computing XOR

The problem of teaching a network to learn how to compute the XOR function has been studied as a "toy" problem by Hinton and Sejnowski

on the Boltzmann machine [13,28]. The network for these problems has one input layer of two units, one intermediate unit, and one output unit. The procedure involves supervised learning in which the network is taught using sets of input–output pairs. During learning, an algorithm based upon statistics and information theory called G-minimization is used to adjust the connections between units. The network gradually develops an internal representation for computing the function correctly.

(iii) Traveling salesman problem

The problem of programming a network to find good, but not necessarily optimal, solutions to the traveling salesman problem (TSP) has been studied by Hopfield and Tank [29], Tesauro [30], and Moody [31]. This problem, which belongs to a class of combinatorially intractable (NP-complete) problems [32], is to find the shortest closed path connecting a set of n destinations. The approach has been to choose first a boolean representation for the problem (assign meaning to the states of the computing units), and then write down a quadratic form whose minima represent appropriate solutions. This quadratic form includes a cost function (tour length) and a set of constraints that insure that only valid tours are found. The t_{ij}s and θ_is are then extracted from this quadratic form.

The TSP solution of Hopfield and Tank has been implemented using continuous dynamics and uses the "tour position" representation. In this representation, n^2 units are required for n destinations. Each unit carries two indices, one for destination d and one for tour position p. A valid tour, where each city is visited once and only once, is given by a permutation matrix in the indices (d, p). Figure 7, adapted from ref. 29, illustrates the matrix of analog output voltages v_i at an intermediate stage in a ten-destination TSP calculation, along with the final tour that the network converged to.

The approach to TSP taken by Tesauro uses the link representation, which offers some advantages for spatial complexity and hardware implementations. The link representation requires at least $n(n-1)/2$ units, where each unit represents one of the possible connections between destinations. A valid tour contains n links. However, the link representation has a serious problem: it often produces disconnected tours. In fact, it is impossible using the link representation alone to enforce the global constraint that tours must be connected using only $O(n^2)$ units.

Moody has developed a hybrid representation, which captures the complexity advantages of the link representation, but also enforces the global constraint that tours be connected in a way analogous to that used in the tour position representation. Moody's solution uses directed

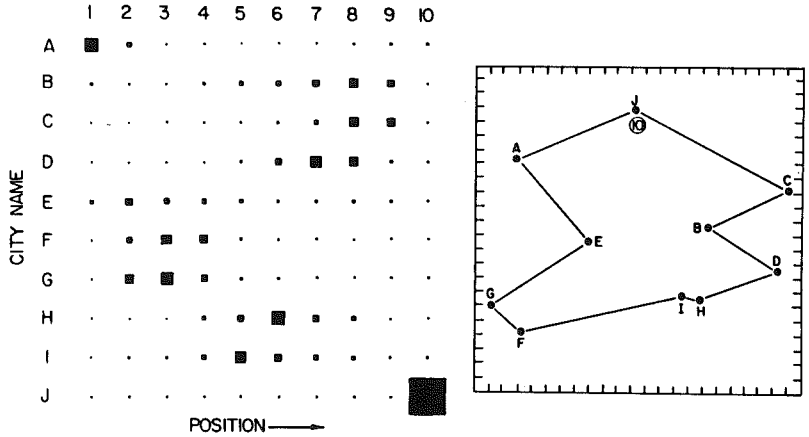


Figure 7 Intermediate analog state and final tour for 10-city TSP

links and a special set of hidden units for a total of $O(n^2)$ units. TSP in the hybrid representation has been implemented using both continuous and stochastic dynamics.

(d) Issues of complexity for Hopfield devices

It is often possible to provide an upper bound on the computational complexity associated with the computation of a given function on some Hopfield device, using the number of units and connections required as the measures of complexity. The computation of a function on a Hopfield device is equivalent to finding a representation for the problem along with a symmetric quadratic form whose minima constitute a solution to the problem. Thus, the complexity of a solution measured in number of units or connections required is very dependent upon the particular network implementation chosen.

(i) For the Hopfield associative memory, the spatial complexity is $O(n)$ units and $O(n^2)$ connections, where n is the number of binary variables in the input string. While spatial complexity is of interest, the real issue for associative memory is storage capacity. The capacity m_u of the network for perfect recall of uncorrelated memories is: $m_u \approx n/(4\log_2 n)$ [33]. If perfect recall is not needed, the capacity for uncorrelated memories is better: $m_u \approx 0.14n$ [12,14,15]. Thus, the network can store only $n \cdot m_u$ bits. However, since each connection must have a logical depth of $(\log_2 m_u + 1)$ bits in order to store m_u memories using the outer product rule, the connections could in principle represent at least $n^2 \log_2 m_u$ bits of information. Thus, the bit efficiency of the memory is rather low.

For the IR associative memory [27], the spatial complexity is $O(n+r)$ units and $O(n \cdot r)$ connections. Here, r is the number of internal units R^α. If the internal representation is unary, then the capacity is $n \cdot r$ for perfect recall; this requires only one trit or two bits per connection. If more bits per connection are allowed, and a sparse, distributed representation is adopted, the storage capacity can be very much greater.

(ii) For the XOR problem, a Boltzmann machine network was able to learn to compute XOR correctly with the minimum required number of units and connections.

(iii) For the traveling salesman problem, Hopfield and Tank [29] have computed approximate solutions using n^2 units and $O(n^3)$ connections, for n destinations. In contrast, the hybrid representation [31] has yielded valid solutions with a network having $O(n^2)$ units and only $O(n^2)$ connections.

As mentioned in the introduction, it is possible to define a simulation complexity for networks that allows comparison of network calculations to standard serial calculations. This has been discussed for Hopfield type networks with particular reference to the TSP [31]. The simulation complexity is proportional to the actual time required to simulate a network calculation on a serial machine. Precisely, it is defined as the number of connections in the network times the number of network updates required for convergence. In the case of discrete networks, one network update time occurs after n asynchronous unit updates, where n is the number of units in the network. For continuous networks, a network update time is the RC time constant of the circuit. While the number of connections in a given network is fixed, the number of network update times required for convergence is not a constant quantity; it depends upon the dynamics used, the initial state of the network, and the quality of the solution desired. For example, simulated annealing requires exponentially slow convergence to guarantee reaching the global minimum [21]. However, quite often, a good, but not globally optimum solution is desired. Under these circumstances, convergence time is typically constant, logarithmic, or linear with problem size. Although time complexity is the most comprehensive measure of network complexity, a full discussion with examples is beyond the scope of this paper.

3. Conclusions

We may draw the following general conclusions from the preceding discussion:

(a) It is possible to define a large class of parallel computational devices, each composed of a massive number of simple processing units,

by prescribing an architecture A for each device and a procedure P by which the system is programmed to compute a given class of functions. The architecture of the device may be specified in terms of a number of parameters that represent a set of constraints concerning the nature of the computational units of the device, the nature of the connections between the units, and the execution protocols of the units. Questions of computational interest concerning such devices include the class of functions that a given device $D = (A, P)$ is capable of computing and the complexity of the associated computations.

(b) With the appropriate use of hidden units, feedback between units and learning procedures, it is now possible to construct devices whose computational power is significantly greater than that of the earlier models.

(c) Classical computational approaches, involving the construction and analysis of sequential algorithms to difficult computational problems such as visual processing and speech understanding, have yet to achieve much success. There is a growing body of evidence, however, that such problems may be amenable to solution on the classes of device discussed in this paper. It is therefore of critical importance that a formal basis for studying both the architectures and procedures that define such devices be established. This basis should have as a major goal the task of understanding the classes of functions computable by such devices and the complexity of the resulting computations.

(d) As both the architectures and the procedures of such devices become increasingly complex, new approaches to the investigation of computational problems will probably be necessary. Some of these approaches will require cross disciplinary analytical tools not generally employed by computer scientists, such as statistical mechanics and dynamical systems. Furthermore, at this stage of development, an emphasis on experimental aspects of computer science also seems necessary.

We believe that such a program of investigation will have profound impacts both on the study of neurophysiological structures and processes and on the design and implementation of massively parallel computing devices.

REFERENCES

1. W.S. McCulloch and W. Pitts, *A logical calculus of ideas immanent in nervous activity*, Bulletin of Mathematical Biophysics **5** (1943), 115–233.
2. F. Rosenblatt, "Principles of Neurodynamics," Spartan, New York, 1962.
3. D.J. Wallace, *Memory and learning in a class of neural network models*, Preprint 86/363, The University of Edinburgh.

4. D.E. Rumelhart, G.E. Hinton and R.J. Williams, *Learning internal representations by error propogation*, "Parallel Distributed Processing," In D.E. Rumelhart, J.L. McClelland and the PDP Research Group, MIT Press, Cambridge, MA, 1986, pp. 318–362.
5. M. Minsky, "Computation: Finite and Infinite Machines," Prentice Hall, Englewood Cliffs, NJ, 1967.
6. S.C. Kleene, *Representation of events in nerve nets and finite automata*, "Automata Studies," Princeton Univ. Press, Princeton, NJ, 1956, pp. 3–42.
7. M. Minsky and S. Papert, "Perceptrons," MIT Press, Cambridge, MA, 1969.
8. D.O. Hebb, "The Organization of Behavior," Wiley, New York, 1949.
9. Y. Le Cun, *A learning procedure for assymetric threshold networks*, Proc. Cognitiva **85** (1985), 599–604.
10. D.B. Parker, *Learning-logic*, TR-47, MIT Center for Computational Research in Economics and Management Science.
11. G. Widrow and M.E. Hoff, *Adaptive switching circuits*, "Institute of Radio Engineers, Western Electronic Show and Convention, Convention Record, Part 4," 1960, pp. 96–204.
12. J.J. Hopfield, *Neural networks and physical systems with emergent collective computational capabilities*, Proc. Natl. Acad. Sci., USA **79** (1982), 2554–2558.
13. G.E. Hinton, T.J. Sejnowski and D.H. Ackley, *Boltzman machines: constraint satisfaction networks that learn*, Tech. Rep. CMU-CS-84–219, Carnegie–Mellon University.
14. J. Amit, H. Gutfreund and H. Sompolinsky, *Spin-glass models of neural networks*, Phys.Rev.A **32** (1985), 1007–2018.
15. D.J. Amit H. Gutfreund and H. Sompolinsky, *Storing infinite numbers of patterns in a spin-glass model of neural networks*, Phys.Rev.Lett. **55** (1985), 1540–2533.
16. I. Kanter and H. Sompolinsky, *Associative recall of memory without errors*, Preprint, Bar–Ilan Univ. Ramat–Gan, Israel.
17. P. Perotto, *Collective properties of neural networks: a statistical physics approach*, Biol.Cybern. **50** (1984), 51–62.
18. P. Peretto and J.Niez, *Stochastic dynamics of neural networks*, IEEE Trans. Systems, Man, and Cybernetics **16** (1986), 73–83.
19. V. Cerny, *A thermodynamical approach to the traveling salesman problem: an efficient simulation algorithm*, Preprint, Inst. of Phys. and Biophysics, Comenius Univ., Bratislava.
20. S. Kirkpatrick, C.D. Gelatt, Jr. and M.P. Vecchi, "Optimization by Simulated Annealing," IBM Thomas J. Watson Research Center, Yorktown Heights, NY, 1982.
21. S. Geman and D. Geman, *Stochastic relaxation, Gibbs distributions, and the Bayesian restoration of images*, IEEE-Pattern Analysis and Machine Intelligence **6** (1984), 721–724.
22. J.J. Hopfield, *Neurons with graded response have collective computational properties like those of two-state neurons*, Proc. Natl. Acad. Sci., USA **81** (1984), 4088–4092.
23. J.S. Denker, *Neural network models of learning and adaptation*, Preprint, AT&T Bell Labs., New Jersey.
24. T. Kohonen, "Self-organization and Associative Memory," Springer-Verlag, New York, 1984.
25. L. Personnaz, I.Guyon and G. Dreyfus, *Collective computational properties of neural networks: analysis of new learning mechanisms for information processing*, Preprint, ESPCI, Paris.

26. A. Lapedes and R. Farber, *Programming a massively parallel, computation universal system*, Preprint, Los Alamos Nat. Lab., New Mexico.
27. E. Baum, J. Moody and F. Wilczek, *Internal representations for associative memory*, Preprint, Inst. for Theoretical Physics, UCSB.
28. G.E. Hinton and T.J. Sejnowski, *Learning and relearning in Boltzmann machines*, "Parallel Distributed Processing," In D.E. Rumelhart, J.L. McClelland and the PDP Research Group, MIT Press, Cambridge, MA, 1986, pp. 282–317.
29. J.J. Hopfield and D.W. Tank, *"Neural" computation of decisions in optimization problems*, Biol. Cybern. **52** (1985), 141–252.
30. G. Tesauro, *Solving optimization problems with analog parallel networks*, Preprint, Inst. for Advanced Study, Princeton, NJ.
31. J. Moody, *Neural networks, computational complexity, and the traveling salesman problem*, Preprint, Inst. for Theoretical Physics, UCSB.
32. M.R. Garey and D.S. Johnson, "Computers and Intractability," W.H. Freeman, New York, 1979.
33. R.J. McEliece, E.C. Posner, E.R. Rodemich and S.S. Venkatesh, *The capacity of the Hopfield associative memory*, Preprint, Jet Propulsion Laboratory.