A new Hopfield-style network for content-addressable memories

Arun Jagota
(jagota@cs.buffalo.edu)
Technical Report 90-02, March 1990
Dept Of Computer Science,
State University Of New York Buffalo, NY 14260

Abstract

Using the binary Hopfield network as a basis, a new model has been developed that has high storage capacity, deterministic non-Hamming distance dynamics, and rapid convergence. It's key properties have been expressed and analysed using graph theory.

Introduction

A content-addressable memory (CAM) model is a model of retrieval of stored memories from incomplete, noisy or ambiguous cues, or multiple competing hypotheses. It can be considered as an attempt to model certain functional aspects of human memory capabilities. A connectionist CAM model, additionally, attempts to model , to some extent, the human-like architecture responsible for such capabilities. The hope is that similar structure will produce similar function , that is, emergent computational behavior. Instead of going from function to structure (problem analysis/computational theory to algorithm/implementation), connectionist research typically goes the other way. This approach has it's merits, because many of the problems that connectionist models are applied to defy exact functional description but yet connectionist models have demonstrated the ability to synthesise solutions from simple and well understood micro-structures. Parenthetically, I might add that what is not always well understood is the macro-behavior that these micro-structures produce, and that is one of the areas of intensive research. Secondly, minor changes to the micro-structure sometimes lead to interesting and diverse macro-behavior, making it possible to apply the same model (with minor variations) to a diverse range of problems. Even without these variations, connectionist models can be applied to a diverse range of problems as has been demonstrated so successfully by models like back-prop [15] and feature maps [16]. And it appears, that there are many not-yet-conceived problems that can perhaps be mapped onto these models, judging by the rate at which new applications keep emerging. The promise is not so much in these models being *optimal* solvers for such problems, rather in that they provide a *unifying* view of computing for certain classes of problems where the class boundaries are still growing and haven't yet been fully established.

The rules that govern which micro-structures should be considered depend on the perspective and background of the modeller. Computer scientists, like I, sometimes relax the physiological plausibility criterion for consideration, if there are strong computational reasons for including a feature.

Memory-intensive connectionist models (like those used for CAM) can also be applied to problems not traditionally solved this way. I'm sure there are researchers that share this view and have known this since quite some time back. As one example, Andy Barto, [4], observes that connectionist approaches to control are special cases of memory-intensive approaches. Adaptive networks can strike a balance between the tradeoffs associated with the extremes of the memory/model continuum. Similar observations could be made for other applications like pattern classification and function learning. Memory-intensive approaches are made possible by the storage capacity and search efficiency of connectionist networks. Multi-layered connectionist networks (like back-prop [15]) can span the extremes of the memory/model continuum. This is of considerable significance, because the same network architecture can be used on a wide range of problems and it's solutions range from *memory-intensive* (for problems that can be solved by memory-intensive solutions) to *model learning* for problems for which the goal is to learn a model. For problems where memory-intensive solutions are feasible, unsupervised connectionist architectures typically have a simpler micro-structure and are capable of very rapid learning. It is with such problems in mind, that I have chosen to focus on unsupervised architectures, highly tailored to memory-intensive applications like CAM. The long-term goal is also to find other problems that can be solved well by such architectures (as an alternative to model learning).

The binary Hopfield network is a particular example of an unsupervised connectionist model with rapid single-shot learning and convergence to *stable* states via local computations, making it especially suitable for CAM applications. But in addition to the above, the appeal of the network lies in its elegant symmetry of lateral connections in a single-layer network. A central question that I have sought to explore is how far one can push a single layer network, specifically, what the limits are on it's capacity and what, if any, are the other problems such a model can be applied to. I have found that incorporation of suitably different learning and energy-descent rules into such a network model result in a new Hopfield-style network with higher capacity and remarkably different but interesting properties. This paper is an attempt to describe this model as best as I can and to map some specific problems onto it.

First, I will qualitatively compare some essential aspects of my model and the classical binary Hopfield network, with respect to storage capacity. The classical binary Hopfield network [1], has two major problems if used as a storage device [3]. The network develops *spurious* memories and can frequently converge to such memories when many memories are stored. By comparison, *spurious* memories can also develop in my model but at a considerably lower rate.

The second problem is that if stored patterns share many features with other stored patterns, the binary Hopfield network can forget some stored patterns, forming new *spurious* patterns instead. Sometimes such patterns are termed *unstable* since if one of them is subsequently presented to the network, the network will converge to some new *spurious* pattern moving away from the original stored pattern. In the model I've developed, any collection of patterns can be stored without losing any stored pattern, so with respect to the second problem, this model has *exponential* capacity. No orthogonalization procedure is needed to keep the stored memories distinct.

The desirable network properties with respect to the above problems are achieved by implicitly incorporating the following bias. Units that are **inactive** don't normally contribute to learning, units that are **active** typically do.

Now, I will compare the dynamics of the binary Hopfield network with mine. The classical binary Hopfield network updates unit states asynchronously. My model updates unit states synchronously, a unit at a time. At a given time, the unit which reduces the energy the most is switched. The network thus performs *steepest descent* in the energy landscape [11]. This difference in energy-descent behavior in conjunction with the weights set by the learning rule impart my model with non-Hamming distance dynamics, as compared to the Hopfield network which behaves like it is using a minimum Hamming distance criterion. Internal resistance (equivalent to a bias term) is associated with each unit in my model, which further guides the process of reaching an energy minimum from an initial state. The internal resistance is an important parameter that has the ability to excercise fine control over energy-descent trajectories, thus opening up the possibility of learning desirable trajectories by learning internal resistances. This makes it possible to use such a model on problems involving *sequential decision making*. Another possible use of internal resistance is to aid in reducing spurious memories that develop in the network when it is used for CAM. Internal resistances might be used to unlearn paths that lead to such spurious states.

Model

I have found that it is convenient to describe the model in the language of domains and sets, although this interpretation is equivalent to that of vectors and vector operations. Let the environment be modelled as a domain D of symbols. A unit is associated with each symbol. The network is fully connected. The number of symbols in D is not fixed. New symbols can be added at any time, which means that the network can *grow* new units and new connections. For purposes of presentation and analysis, however, we assume a fixed size domain D on n symbols, $s_1,\ s_2,\ ..., s_n$. Let k be the maximum size of any stored set, that is $k\ =\ max\ (|S_j|)$ for all stored sets S_j. Then this model makes k more symbols available, named 1 through k. These symbols *represent* sizes of stored sets and are essential for storing subsets of other stored memories [12]. A unit is associated with each *size* symbol and such units are called *size* units. p_i denotes the i_{th} unit and $p_i(t_k)$ denotes its value at time t_k where t_0 and t_f are the initial and final times respectively. w_{ij} denotes the *symmetric* constraint between units p_i and p_j. In a stable state (local energy minimum), each unit, p_i, has the value 0 or 1. $p_i(t_0)$ are allowed values in the interval $[0,1]$. A spatial memory (memory to be trained) is represented by the set

$$S_m = \{s_{i_1}, s_{i_2}, ..., s_{i_j}\},\quad S_m \subset D$$

For training, the network implicitly includes the symbol representing $|S_m|$, in this case, j, in the set S_m. Each spatial memory is represented by a particular set, S_i. Symbols in that set may either be clamped or unclamped. s_i^{*1} denotes a symbol clamped to 1. s_i^{*0} denotes a symbol clamped to 0. It is assumed, unless otherwise mentioned, that all units in a memory are clamped to 1 during training, so that, for notational convenience, they don't have to be shown as clamped.

The network has the following initial state. ρ is kept $< \ -2(n+k)$ for CAM, but can be any value ≤ 0, for other applications. Later on we shall see that the network's behavior can be controlled by choosing different values of ρ.

$$\begin{aligned} w_{ij}(0) &= \rho \quad \forall i,j \quad i \neq j \\ w_{ii}(0) &= 0 \quad \forall i \\ R_i(0) &= 0 \quad \forall i \end{aligned} \tag{1}$$

Learning Rule

Whenever a new spatial input (S_j) is presented, the network learns constraints between *all pairs* of clamped symbols in S_j according to the following rule, some of whose key features were developed in [11].

$$\begin{aligned} w_{ij}(t+1) &= \lambda \qquad\qquad\qquad\qquad\quad if\ p_i^{*1}\ and\ \ p_j^{*1}\ and\ w_{ij}(t) = \rho \\ w_{ij}(t+1) &= w_{ij}(t)\ + \lambda * f(w_{ij}(t)) \quad if\ p_i^{*1}\ and\ \ p_j^{*1}\ and\ w_{ij}(t) \neq \rho \\ w_{ij}(t+1) &= -\lambda \\ & \quad\ if\ p_i\ and\ \ p_j\ are\ clamped\ and\ p_i \oplus p_j\ and\ w_{ij}(t) = \rho \\ w_{ij}(t+1) &= w_{ij}(t)\ - \lambda * f(w_{ij}(t)) \\ & \quad\ if\ p_i\ and\ \ p_j\ are\ clamped\ and\ p_i \oplus p_j\ and\ w_{ij}(t) \neq \rho \end{aligned} \tag{2}$$

$$w_{ii}(t+1) = w_{ii}(t)\ + \lambda * f(w_{ii}(t)) \quad if\ p_i^{*1} \tag{3}$$

$$R_i(t+1) = R_i(t)\ + \lambda_R * f(R_i(t)) \qquad if\ p_i^{*1} \tag{4}$$

$$\begin{aligned} f(x) &= sig(x)/(1+sig(x))^2 \\ sig(x) &= \frac{1-x}{1+x} \end{aligned} \tag{5}$$

The nonlinear function f allows weights (and R_i) to grow in the form of a *sigmoid*, as a function of the individual correlation frequencies. λ, the learning rate, determines the shape of the weight sigmoid, while λ_R, the internal resistance (R_i) learning rate determines the shape of the resistance sigmoid. $\lambda_R < \lambda$. The advantage of such a weight update scheme is that it keeps weights bounded (in the interval $[-1, 1]$), except when they have the value ρ. As importantly, individual synapses are allowed to stabilise after some number of trials because subsequent input don't affect those synapses as much. But, synapses that have as yet to develop are still capable of rapid learning. The second advantage hasn't yet been exploited in any applications, however.

This learning rule exhibits some other important properties. Firstly, it allows the development of local minima separated from each other via strong lateral inhibitions (ρ). Such a scheme *usually* prevents two or more local minima from collapsing to form their union or interfering with each other in any other way. This technique had also been discussed in [10].

Self-weights (w_{ii}) are a non-linear measure of presentation frequency of a symbol (in all preceeding input). When units from different memories are active, the competition is biased in favor of those that occur more frequently in nature (see (6)). Like weights between different units, self-weights can saturate and lose their discriminatory power.

R_i is the internal resistance of a unit (it's resistance to switching). In the learning rule described above, it's primary role is in, given a particular spatial memory, promoting the switching of symbols that are *critical features* in the memory over those that are not, even if the latter were better correlated. There are other uses (and ways of learning R_i) that will be discussed later and have been hinted at earlier.

Energy descent rule

An input to the network is also a set

$$S_m = \{v_{i_1}.s_{i_1}^{*1}, v_{i_2}.s_{i_2}^{*0}, ..., v_{i_j}.s_{i_j}\}, \qquad S_m \subset D \cup \{1, ..k\}$$

but this time the symbols in the set can be optionally *weighted* and/or clamped (to 0 or 1). Additionally, the size symbols can be used in a test set. From a test input S_m, $p_i(t_0)$, the initial states of the units, are calculated as follows.

$$p_{i_k}(t_0) = \begin{cases} v_{i_k} & \text{if } s_{i_k} \in S_m \text{ and was given a confidence value} \\ 1 & \text{if } s_{i_k} \in S_m \text{ but was not given a confidence value} \\ 0 & \text{otherwise} \end{cases}$$

A set can be interpreted as a vector by imposing an order on the symbols in $D \cup \{1,..k\}$, say ordered as $s_1, ...s_{n+k}$. S_m above can be interpreted as the $n+k$ component vector x where $x[i_l] = p_{i_l}$, the value of unit p_{i_l}.

Unit activations are governed by the Hopfield energy-minimization principle [1] but with different dynamics. The state of unit p_i may be switched if it reduces the global energy of the network. The energy descent rule switches the unit that reduces the energy the most, thus performing *steepest descent* in energy space [11]. The model is synchronous and exactly one unit is switched in each cycle. The local energy each unit can lower by switching is calculated as follows

$$n_i = \sum_j w_{ij} * p_j$$

$$V_i = \begin{cases} 1 & \text{if } n_i > 0 \text{ and } p_i < 1 \\ -1 & \text{if } n_i < 0 \text{ and } p_i > 0 \\ 0 & \text{otherwise} \end{cases} \tag{6}$$

$$\Delta E_i = -V_i * n_i + |V_i|(R_i + r) \tag{7}$$

where n_i is the net input to unit p_i and V_i is the direction that p_i must switch. R_i is the internal resistance of p_i and r is a global resistance parameter. Units that are *clamped*, that is held forcibly to 0 or 1 are not allowed to switch. Let $p_{i_{min}}$ be the unit with minimum ΔE. If $\Delta E_{i_{min}} < 0$ then $p_{i_{min}}$ is switched in the direction of $V_{i_{min}}$.

Let us briefly summarise some of the properties of this energy-descent rule. Formal and detailed proofs are given in following sections. Firstly, *Steepest descent* makes a locally optimal decision at each step given the information available (the units that are ON or active). When the collection of initially active units is mutually inhibiting (they are not all in the same stored memory), *steepest descent* is responsible for providing desirable *competition*, switching OFF the active unit inhibited by the largest number of other active units. If there is a tie, then self-weights (and other weights) help determine the unit that switches OFF. Competition can also be controlled by the external bias in the form of initial unit activations (in the interval $[0,1]$). V_i has the value $\{-1,0,1\}$ even if the initial unit values are in the interval $[0,1]$. This ensures that the switching unit is driven to 0 or 1 in one step. Finally, *steepest descent* works well when the stored memories result in an energy landscape in which there is a correlation between the initial *steepness* and *depth* of minima, in other words the basins are steep ravines or uniformly shallow. When the number of stored memories is small and only the units clamped to 1 are involved in learning, the basins that are formed typically have this property.

Now let us look at the other parameters, specifically R_i and w_{ii}. w_{ii} is a nonlinear measure of the occurence frequency of p_i in nature. R_{ii} is the same measure but plays the opposite role. Assuming $\Lambda_R < \Lambda$, after α training steps, $R_i(\alpha) < w_{ii}(\alpha)$. When considering units to switch ON, $w_{ii}(\alpha)$ plays no role since candidate units are currently OFF. However, $R_i(\alpha)$ does play a role. Other terms being equal, the unit with least $R_i(\alpha)$ will switch ON. On the other hand, when considering units to switch OFF, since $w_{ii}(\alpha) > R_i(\alpha)$, $w_{ii}(\alpha)$ dominates. Hence, other terms (except $R_i i, w_i i$) being equal, the unit with smallest $w_{ii}(\alpha)$ will switch OFF.

Some simple examples

Let me now illustrate the above properties with some simple examples. Consider the following domain

$$D = \{a, b, c, d, e\}$$

Training Input : The numbers before each input are the frequency of presentation.

$$10 \ \{a^{*1}, c^{*1}, e^{*1}\} \qquad 6 \ \{a^{*1}, b^{*1}, d^{*1}\} \qquad 1 \ \{a^{*1}, b^{*1}, c^{*1}\}$$

Fig 1.1 shows G_n, the graph underlying the network after training. Vertices that are not adjacent (connected by edges) are mutually inhibiting with strength ρ. Each edge denotes a positive weight between it's incident vertices.

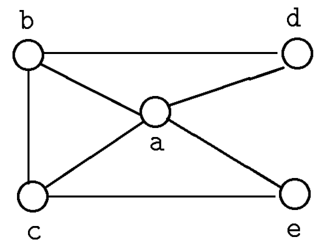

Fig 1.1 - Gn after training

Test Input : The test input is at the left of $-->$, the network output at the right.
Steepest descent:

$$\{a, c\} --> \{a, c, e\}$$

Because *switching* 'e' decreases energy more than switching 'b'.
Self-weights:

$$\{e, d\} --> \{a, c, e\}$$

'e' wins competition over 'd' since it has higher self-weight.
Internal Resistance : Assume units don't have internal resistance. By applying our energy descent rule, the following minimum will be reached.

$$\{b\} --> \{a, b, c\}$$

But it is more desirable to reach $\{a, b, d\}$ from $\{b\}$. If we learn an internal resistance according to our learning rule and then use our energy descent rule,

$$\{b\} --> \{a, b, d\}$$

In other words, 'd' is switched before 'a' since it is more *critical* to $\{a, b, d\}$ whereas 'a' is not.

Internal resistance can sometimes hinder reaching a local minimum. Consider the following example

$$100\ \{a^{*1}, b^{*1}, c^{*1}\} \qquad 100\ \{a^{*1}, b^{*1}, d^{*1}\} \qquad 100\ \{e^{*1}, c^{*1}\} \qquad 100\ \{e^{*1}, d^{*1}\} \qquad \{a^{*1}, e^{*1}, f^{*1}\}$$

From $\{f\}$, we should reach the minimum $\{a, e, f\}$ but the high internal resistances of a and e might prevent this from happening.

A scheme to correct this behavior has not yet been incorporated into the model. One simple solution is to, in every cycle in which there is no switch, gradually decrease, temporarily, the R_i of every unit until a unit switches ON or the R_i of every unit becomes 0. If the R_i of every unit becomes 0 and still no unit switches ON, then the network is already at an energy minimum. By temporarily, I mean that the R_i of every unit must be set to it's original value after the desirable energy minimum is reached. The reasons that this scheme has not *yet* been incorporated into the model are that the dynamics (and the implementation machinery) get needlessly complicated for a phenomenon that might be a rare occurence in real applications and secondly, that this phenomenon seems related to the phenomenon of detectable *inconsistent* states , where the number of units ON in the final state conflicts with the size unit that is ON. Such states are possible, and can be self-detected and avoided by backtracking. It is my goal to try to handle both problems with the same mechanism.

Another simple example showing the role of internal resistance is as follows. The R_i of size units can bias the network against retrieving memories of sizes that occur commonly. In other words, if the distribution of memories in the training set is biased towards particular sizes, then during recall, if a size unit is not initially ON, the network will, in general, be biased towards sizes other than those occuring commonly.

A second use, which has not yet been incorporated into the model, or tried on any application is as follows. We might let $R_i's$ be set externally. An application of this might be to set the R_i of size units in such a way as to bias the network to retrieve memories of particular sizes. As a specific example, consider storing words in the network as memories (see Applications section). We might want to bias the network to retrieve words of large size.

Formal Analysis

In this section, I formally analyse the static and dynamic properties discussed above and present some important results. Graph theory has played a key role in developing the proofs and expressing the network properties in a formal language. Any introductory text on Graph theory will be useful as a reference. [2] is an example of a good text.

The graph underlying the network

We will use G_n to denote the graph underlying our network with the following correspondence. $G_n = (V_n, E_n)$ where $V_n = D \cup \{1, \ldots, k\}$, is the set of vertices of the graph G_n. V_n corresponds 1-1 with the domain of symbols plus the size units. $E_n = \{(x, y) | x \neq y \ and \ w_{xy} \geq 0 \text{in the network}\}$ is the set of edges of G_n. In other words, G_n is the graph obtained by removing all the $-ve\ value$ weights and the self-weights from the network.

Two vertices in a graph are *adjacent* if they are connected by an edge. Two vertices are *connected* if there is a path that connects them. In graphs, two vertices may be *connected* but not *adjacent*. We will use $G_n[S]$, to denote the graph (subgraph of G_n) induced by the units corresponding to the set of symbols S. Intuitively, it is the set of pairwise connections (edges) between units associated with S. A graph is *complete* if every pair of vertices is adjacent. Clearly, if S_m is a stored memory, $G_n[S_m]$ is complete, since the weights of all induced edges (connections between pairs of symbols in S_m) after storing S_m must be positive. A graph G_s (subgraph of G_n) is *maximal* with respect to a property P of G_n if and only if no *proper* supergraph of G_s but subgraph of G_n has that same property. We will also use the concept of *clique*, defined as a *maximal* complete subgraph of the graph G_n. In other words, G_s is a clique of G_n if no proper superset of G_s is also complete. The reader should note that this definition of a *clique* is slightly different from the standard one in graph theory. Finally, we define a connected component of a graph G_n, denoted by CC as a *maximal* connected subgraph of G_n, that is, all pairs of vertices in CC are connected by some path. If G_n is connected then it has exactly one CC, G_n itself. The connected components of a graph are disjoint subgraphs.

Storage properties

Let ρ, the initial lateral inhibition between all pairs of units be $< -n$, the total number of units. The following property, stated and proved formally as Lemma 3 in the section on Dynamic properties then holds. If no unit switches OFF at time t_{i+1}, then all connections between all active units at time t_i must be positive. From this we may conclude that pair-wise connections between all active units in the final stable state must be positive, in other words complete subgraphs in G_n. We can say even more, they must be cliques in G_n. Thus when $\rho < -n$, the storage properties of the network can be analysed by analysing G_n, it's underlying graph, because only the signs of weight values are important, not their magnitude.

On a domain with n symbols there are 2^n possible memories. Ideally, one would hope to be able to store any m of them, without collapsing two or more of them into one, or storing extra memories (spurious minima). Storing a particular memory, say S_i, amounts to constructing a complete sub-graph ($G_n[S_i]$) of $+ve$ weights on the units in S_i (see the learning rule). It is easy to see that, in our model as described so far, neither criterion can be met. [1]

It is easy to find instances of memories $S_i, S_j, ...S_k$ which when stored (trained), form a new complete sub-graph, $G_n[S_u]$, of +ve connections. The distinct stored memories $S_i, S_j, ...$ are lost and the new single (untrained) memory $S_u = S_i \cup S_j \cup ..S_k$ is formed. Also, $G_n[S_u] = G_n[S_i] \cup G_n[S_j] \cup ..G_n[S_k]$ the newly formed clique.

Similarly, spurious memories can develop when there exist collections of memories $S_{i_1}, S_{i_2}, ..., S_{i_k}$ such that storing them in the network by applying the learning rule will also store new memories (spurious), say $S_{i_{k+1}}$ etc.

Some examples of these two cases are:

Consider a 6 unit network, units labelled 1 to 6. Then if we store

```
{1,2} { 2,3} {3,1}
```

these memories are lost and a new memory {1,2,3} is formed. On the other hand, if we store the following memories

```
{1,2,3}, {3,4,5}, {5,6,1}
```

. We see that a spurious memory {1,3,5} develops, even though the original memories are retained.

Storage Capacity

By storage capacity, [2] I mean the number of memories that can be stored in the network, as a function of it's size, without *losing* any stored memory, or said another way, without making any of the fixed points unstable. I will first compute the capacity in some special cases, one of them , interestingly, leading me to consider a new application.

The result I describe uses properties of k-partite graphs, although it can be shown as easily without needing to employ graph theory.

[1] Later on, I will show, that, the first criterion, that is losing no memories can always be met, by a suitable transformation of the domain D.

[2] A harder question to answer is the rate of spurious memory development, as a function of network size and number of stored memories. No theoretical results on this have been obtained.

Notation: We use the term k-ary or sometimes just k- to denote a set with k elements.
A k-partite graph is a graph whose vertex set, V, is divided into k disjoint parts $V_1, \ldots, V_k$ so that there are edges only between parts, that is, no two vertices in the same part are adjacent (connected by an edge).
A *simple* graph is a graph without multiple edges (between a single pair of vertices) and without self-loops.
$T_{k,n}$ denotes the *complete* k-partite graph on n vertices so that the parts are chosen as evenly as possible. In other words, $|V_i| = \lfloor \frac{n}{k} \rfloor$ or $|V_i| = \lceil \frac{n}{k} \rceil$.
$\varepsilon(G)$ denotes the number of edges in the graph G.
Fact: G_n is simple.
Fact: If n is a multiple of k, $T_{k,n}$ has $(\frac{n}{k})^k$ cliques of size k and these are the only cliques $T_{k,n}$ has. $T_{k,n}$ has no complete subgraph of size $k+1$.

I now state and prove the following theorem. For Theorem 1 and Lemma 1, I assume that there are no size units, since none are needed. I also assume that n is a multiple of k.

Theorem 1 *If we wish to store only k-ary sets in an n unit network and are allowed to choose that collection of such sets which maximises storage, then we can choose $(\frac{n}{k})^k$ such sets.*

Proof: If we can show that for any collection of k-ary sets that we choose , a $k+1$-complete subgraph can never form, then we are done. Because if so, we can never lose any stored memory since memories can only be lost by getting subsumed into a larger one.
Consider the graph $T_{k,n}$. $T_{k,n}$ has no $k+1$-clique. $T_{k,n}$ has $(\frac{n}{k})^k$ k-cliques. Each k-clique of $T_{k,n}$ can be made to correspond with a unique k-ary set to be stored, a fact that is proved formally in Lemma 6. Choose the $(\frac{n}{k})^k$ k-ary sets that correspond to the k-cliques and store them in the network. The G_n that forms will be $T_{k,n}$. Hence this collection of k-ary sets can be stored without losing any of them.

It is instructive to look at the special case, $k = 2$, that is storing only *binary* sets. The first reason is rather personal, because thinking about binary sets revealed a new application that, in retrospect, doesn't quite follow from it, but is related. The application is to store graphs in the network by presenting each edge (a binary set) as a memory. The second reason is that the following lemma can be proved easily, for $k = 2$.

From Thm 1 we see that we can store $\frac{n^2}{4}$ binary sets in the network. Strictly speaking this should be $\lfloor \frac{n^2}{4} \rfloor$, to cover the case when n is odd. I wish to show that $\lfloor \frac{n^2}{4} \rfloor$ is also an upper bound on the number of binary sets that can be stored, no matter how they are selected. In order to do this, I invoke Turan's Theorem, a well-known theorem in graph theory. Again, [2] is a good reference. For convenience, I will state the theorem.

Turan's Theorem: *If G is simple and contains no $k+1$-complete subgraph, then $\varepsilon(G) \leq \varepsilon(T_{k,n})$.*
I will now state and prove the following Lemma[3].

Lemma 1 *For $k = 2$, the maximum number of binary sets that can be stored is $\lfloor \frac{n^2}{4} \rfloor$.*

Proof: G_n cannot have any 3-cliques but must have 2-cliques. Hence, By Turan's Theorem, $\varepsilon(G_n) \leq \varepsilon(T_{2,n})$. Now, $\varepsilon(T_{2,n}) = \lfloor \frac{n^2}{4} \rfloor$. Hence, $\varepsilon(G_n) \leq \lfloor \frac{n^2}{4} \rfloor$. But, since the only 2-cliques are the edges in G_n, it follows that G_n has $\leq \lfloor \frac{n^2}{4} \rfloor$ 2-cliques (binary memories).

Now let us see for what value of k (the size of each memory), the network has maximum capacity. In other words, we want the k_m that maximizes $(\frac{n}{k})^k$.
Let $f(k) = (\frac{n}{k})^k$. Then, k_{max} maximises f when $f'(k_{max}) = 0$. Now

$$f'(k) = \frac{n^k}{k^k}(\ln\ n\ -\ (1\ +\ \ln\ k))$$

Since $n > k$, this gives $k = n/e$ for $f'(k) = 0$. Hence,

$$f_{max} \quad = \quad e^{n/e}$$

In other words, the network has exponential capacity. If all the sets we wish to store are of size $n/2$, we can store upto $2^{n/2}$ such sets which is quite close to f_{max} since $n/2 \approx n/e$.

[3]It should be noted that Lemma 1 is simply a restatement of the problem of finding the maximum number of edges a graph can have without forming triangles. Since this proof is simple and I'm merely echoing it, in the interest of presentation, neither my Lemma, nor the following proof, should be considered as my original work.

Again, we must remember that this does not include the effect stored memories have on the development of spurious memories.

Implications: The above theorem shows that it is *likely* that the network exhibits good storage properties for applications with sets resulting in k-partite underlying graphs after storage. Later on we shall see that there are important applications which inherently have such structure. Specifically, their domains have a *combinatorial structure*, a term I shall define later.

I will also show that on a domain D having n symbols, we can store *any* collection of the 2^n possible memories by transforming the domain into a domain with a combinatorial structure. This important result will establish exponential capacity for any domain.

Storing graphs

Binary sets provided a new application for the network, storing *graphs* by presenting the edges (binary sets of vertices) as memories to the network. Any undirected graph without self-loops can be stored this way. Recall that a graph G is a set of vertices V and a set of edges $E = \{(x, y)|x, y \in V\}$. G can be stored in the network as follows. With each vertex in V, a unique unit in the network is associated. Since G is undirected, each edge $e \in E$ is a binary unordered set whose elements are the two vertices incident on e. Each edge in G is then stored as a memory in the network. A positive weight, if any, on an edge of G can be *captured* by repeatedly presenting that edge a number of times proportional to its weight in G. The self-weight of a unit in the network reflects the *weighted* degree of the corresponding vertex in G. Now if G is a tree, a bipartite graph, or any other graph without triangles, then we know from our earlier analysis that the stored binary memories will be retained (no memories will collapse, no spurious memories will develop). In other words, the network's stable states are the edges, that is, the two associated vertices, in G. In other words, if the input to the network is any set of vertices, the network will retrieve exactly two vertices corresponding to one of the stored edges.

Recall that by our definition, *cliques* are *maximal* complete subgraphs of G. If G has cliques of size ≥ 3, that is G has triangles, then clearly, some stored memories (edges) will be lost. In general, for any graph G stored in the network by presenting it's edges, the stable states of the network have a 1-1 correspondence with the *cliques* in G. This also holds when G has no triangles because then it has only 2-cliques, that is edges. Lemma 6 states and proves this property formally.

The network can then be used to store any graph and retrieve it's cliques. The above dynamics holds under the condition that $\rho < -n$. It is also useful to consider the case when $-n < \rho \leq 0$, in particular when $\rho = 0$. It will be shown later, but is fairly intuitive to see even now, that when $\rho = 0$, the network retrieves all connected components in G that contain the vertices in a test input.

An illustrative list of questions that can be formulated and answered about a graph G stored as above is as follows. With the answers, I will also indicate the retrieval time. Let's assume T time units are required to look up a table and map a node name to the corresponding unit. Note that even on a sequential machine, table lookups can be done very rapidly by using *hash* functions. The inverse mapping (from unit to node name) is an array indexing assumed to take negligible time. Having factored out table lookup and indexing times, I will only show the number of steps in which the network answers each of the following questions.

In the following set of questions, $\rho < -n$.

1. List the vertex adjacent to a given node v on it's highest weighted edge. This takes one network time unit to retrieve the correct vertex.

2. Are vertices $v1$ and $v2$ adjacent (connected by an edge)? This takes one time unit to get the answer. If they are not adjacent then one of the vertices has to switch OFF in the first time step.

3. From an input set of vertices S_i, retrieve a clique that contains *some large* subset of S_i. It is not possible, at this time to change the *some* to something more definite. Thm 4 does show that if S_i is a subset of some stored clique, then S_i will also be a subset of the retrieved clique (not necessarily the same one). It also characterises the trajectory during descent, that is the manner in which S_i is successively transformed to reach the final state. Let C_i be the clique found. Then the network takes $\leq |C_i|$ steps to find the answer.

   (a) Special Case: Given a vertex v, find some clique containing it. Again, let C_i be the clique found. The network finds it in $|C_i| - 1$ steps.

To answer the following set of questions, ρ is set to 0.

1. Are nodes $v1$ and $v2$ connected in G. That is, is there a path from $v1$ to $v2$ and vice-versa. This question can be answered by presenting the set $\{v1\}$ to the network and checking if $v2$ gets activated at some time. Let

$|CC_1|$ be the size of the connected component of $v1$. Then the network takes $\leq |CC_1|$ time units to get the answer since within that time every unit in CC_1 is made active. If $v2$ is active then it is connected to $v1$, else not.

2. Given an input set S_i, list all the connected components of G that contain at least one vertex from S_i. Let $CC_1, \ldots, CC_k$ be the k connected components containing at least one vertex from S_i. Then the network will retrieve them in $-|S_i| + \sum_{i=1,k} |CC_i|$ time steps.
   (a) Special case 1: Given a vertex v, list the connected component containing it. Let CC_1 be the retrieved connected component. Then the network takes $|CC_1| - 1$ time steps to retrieve it.
   (b) Special case 2: A set of vertices V_s is called a *path cover* of G if every vertex in G is connected (by a path) to some vertex in V_s. Given a set of vertices V_s, test if V_s is a path cover of G. If the network takes $n - |V_s|$ time units to rest, then V_s is a path cover of G since all the units in the network are active. If the network rests earlier, then V_s is not.

Dynamic properties

Energy Descent

I shall now state and prove some Lemmas and theorems which provide insight into the network's dynamic properties. **Notation:** Recall that D is the domain of symbols or set of units in the network. Sometimes I will use V instead of D since V, the set of vertices in G_n, is equivalent to D. Let n denote the total number of units (including size units), w_{xy} denote the weight between units for the symbols x and y, $n_x(t_i)$ denote the net input to the unit for symbol x at time t_i and $S_a(t_i)$ denote the set of units active at time t_i. Let $S_{a_{inh}}(t_i)$ denote the set of units active at time t_i such that $\forall s_i \in S_{a_{inh}}(t_i), \exists s_j \in S_a(t_i)$, such that $w_{ij} = \rho$. In other words, $S_{a_{inh}}(t_i)$ is the set of all units active at t_i that receive some inhibition from active units.

Let S be a set of units and equivalently, a set of vertices in the underlying graph G_n. Recall that $G[S]$ is the subgraph of G_n induced by the vertices in S. and that $G[S]$ is *complete* if and only if the network weights (w_{xy}) of all the pairwise connections between all units in S ($\forall x, y \in S$) are nonnegative.

Let s_m denote the size of the largest fixed point (stable state), after training, in the network. Let I_m denote the size of the largest input that can be presented to the network. We assume $I_m > s_m$, that is we want to retain the ability of presenting input larger than the size of any stored memory to allow the network to be used for constraint relaxation. We also assume that I_m and s_m are of the same order which may be much less than n. This is a useful assumption, as we shall see later. I_m and s_m can have the value n, the number of units, in the worst case. If on the other hand, I_m (and hence s_m) can be better bounded for specific applications, it allows us to use smaller (in magnitude) values of ρ to achieve the same network properties.
Assumptions: The lemmas and theorems are based on the following assumptions.

1. After training, all remaining negative connections in the network have the value ρ. In other words, $w_{xy} = \{\rho, [0, 1]\}, \forall x, y$.
2. $\rho < 2I_m$. This is really not an assumption, rather what I wish to show given the properties that I want this network to possess, as stated in the following Lemmas/Theorems. It is much easier to prove *all* these Lemmas/Theorems when $\rho < -2n$, by skipping all the steps that invoke the graph-theoretic Thm 1 and Corr 1. I will not do that, however, because I want to put a tighter bound on ρ.
3. Unit states are binary. A unit switches ON if it changes state from 0 to 1. It switches OFF if it changes state from 1 to 0.
4. R_i, the internal resistance of each unit, is 0.
5. Units are not clamped in the test input.

The first theorem sets up an important property of graphs that will be used in subsequent proofs. An existing graph-theoretic result could not be found easily, so it was found convenient to set up a theorem. The theorem was proved by Prof. Xinhe, Dept of Computer Science, SUNY Buffalo, NY.

Theorem 2 *Let G be a graph of n vertices and s_m be the size of the maximum clique in G. Then $\exists$vertex $v \in G$ so that $deg(v) \leq n - \lfloor \frac{n}{s_m} \rfloor$.*

Proof: Suppose not. Then all vertices have $deg(v) > n - \lfloor \frac{n}{s_m} \rfloor$. The following algorithm will construct a clique C of size $> s_m$.

1. $Rem \leftarrow V; C \leftarrow \phi$
2. Pick any vertex $v \in Rem$.
3. $C \leftarrow C \cup \{v\}$.
4. $Rem \leftarrow Rem \; - \; \{v\} \; - \; \{u | u \in Rem \;\; \& \;\; u$ is not adjacent to v in $G\}$.
5. If $Rem = 0$ Stop.
6. Go to 2.

Let $C(k)$ and $Rem(k)$ denote the values of C and Rem respectively after the k_{th} iteration of the loop.
By induction we will show that at any stage i of the execution, $C(i)$ is a complete subgraph and $\forall x \in Rem(i)$, x is connected to all vertices in C_i.
Basis:
$C(0)$ is a complete subgraph trivially.
$\forall x \in Rem(0)$, x is connected to all vertices in $C(0)$. This is true trivially.
Induction step:
Assume $C(k)$ is a complete subgraph.
Assume, $\forall x \in Rem(k)$, x is connected to all vertices in $C(k)$.
Then we have to show that

1. $C(k+1)$ is a complete subgraph.
2. $\forall x \in Rem(k+1)$, x is connected to all vertices in $C(k+1)$

C is only changed in Step 3. By the induction hypothesis, since $C(k)$ is a complete subgraph and $v \in Rem(k)$ is connected to all vertices in $C(k)$, it follows that $C(k+1)$ is a complete subgraph. Rem is changed in Step 5. The only new vertex in $C(k+1)$ is v, and all vertices in Rem not connected to v are removed in Step 4. Hence it follows that $\forall x \in Rem(k+1)$, x is connected to all vertices in $C(k+1)$.

Hence $C(N)$ is a complete subgraph and Rem is empty when the algorithm terminates. Let us now calculate a lower bound on N, the number of iterations. From step 4, we can see that the number of vertices removed from Rem is

$$\begin{aligned} &\leq 1 + (n-1) - deg(v) \\ &= n - deg(v) \\ &< \lfloor \tfrac{n}{s_m} \rfloor \end{aligned}$$

Hence,

$$|Rem(k) - Rem(k+1)| < \lfloor \frac{n}{s_m} \rfloor$$

Since $|Rem(0)| = n$,

$$N > \frac{n}{\lfloor (n/s_m) \rfloor} \geq s_m$$

Since $|C(0)| = 0$, and a new vertex is added to C in every iteration, $|C(N)| \geq s_m + 1$. Hence C is a complete subgraph of size greater than s_m, which is a contradiction.

Corollary 1 *Let $G = (V, E)$ be a graph. For any $V_1 \subseteq V$, let $d_G(x)|V_1 = |\{(x, y) | y \in V_1\}|$. In other words, $d_G(x)|V_1$ is the degree of x wrt to the vertex set V_1. Let $\bar{d}_G(x)|V_1 = |V_1| - 1 - d_G(x)|V_1$. In other words, $\bar{d}_G(x)|V_1$ is the number of vertices in V_1 that x is not adjacent (directly connected) to.*

Let $V' \subset V$ such that $|V'| > s_m$. Then, $\exists x \in V'$ such that $\forall y \in V - V'$, $d_G(y)|V' < (s_m + 1) * (\bar{d}_G(x)|V' + 1)$.

Proof By defn, $\forall y \in V \; - \; V', \quad d_G(y)|V' \leq |V'|$.
Since s_m is the size of the maximum clique, By Thm 2,

$$\begin{aligned} & \exists x \in V', d_G(x)|V' && \leq && |V'| - \lfloor \tfrac{|V'|}{s_m} \rfloor \\ \Leftrightarrow \;& \bar{d}_G(x)|V' && \geq && \lfloor \tfrac{|V'|}{s_m} \rfloor - 1 \\ \Leftrightarrow \;& |V'| && < && (s_m + 1) * (\bar{d}_G(x)|V' + 1) \end{aligned}$$

Hence $\forall y, \quad d_G(y)|V' < (s_m + 1) * (\bar{d}_G(x)|V' + 1)$.

Lemma 2 *$G[S_a(t_k)]$ is incomplete $\Leftrightarrow$ some unit switches OFF at t_{k+1}.*

Proof: "=>". Assume $G[S_a(t_k)]$ is incomplete. We have to show that some unit switches OFF at t_{k+1}.
Consider G_n, the graph underlying the network. Hence $\exists x \in S_a(t_k)$ such that $\bar{d}_G(x)|S_a(t_k) \geq 1$. Pick the $x \in S_a(t_k)$ that satisfies Corr. 1, that is $\forall z \in D - S_a(t_k)$

$$d_G(z)|S_a(t_k) < (s_m + 1) * (\bar{d}_G(x)|S_a(t_k) + 1)$$

Choose any $z \in D - S_a(t_k)$. We have to show that x will switch OFF, and z cannot switch ON. The positive weights have value ≤ 1. Hence,

$$\begin{aligned} n_z(t_{k+1}) &\leq d_G(z)|S_a(t_k). \\ &< (s_m + 1) * (\bar{d}_G(x)|S_a(t_k) + 1) \end{aligned}$$

Now, $\bar{d}_G(x)|S_a(t_k)$ is the number of inhibitory connections from active units ($S_a(t_k)$) to x and each inhibitory connection has the value $\rho = -2I_m$. Hence,

$$\begin{aligned} n_x(t_{k+1}) &\leq -2I_m * \bar{d}_G(x)|S_a(t_k)) + d_G(x)|S_a(t_k) \\ &\leq -2I_m * \bar{d}_G(x)|S_a(t_k)) + I_m \\ &= -2I_m * (\bar{d}_G(x)|S_a(t_k) - 1/2) \end{aligned}$$

Since $\bar{d}_G(x)|S_a(t_k) \geq 1$ and $I_m > s_m$,

$$\begin{aligned} & | -2I_m * \bar{d}_G(x)|S_a(t_k)) \ | &\geq (s_m + 1) * (\bar{d}_G(x)|S_a(t_k) + 1) \\ \Leftrightarrow \quad & |n_x(t_{k+1})| &\geq (s_m + 1) * (\bar{d}_G(x)|S_a(t_k) + 1) \end{aligned}$$

Hence $n_z(t_{k+1}) < |n_x(t_{k+1})|$.
The steepest descent energy rule will hence switch x OFF instead of switching z ON.
"<=". Assume some unit $x \in S_a(t_k)$ switches OFF. We have to show that $G[S_a(t_k)]$ is incomplete.

A unit can switch OFF only if it has at least one $-ve$ connection from other active units. Hence $G[S_a(t_k)]$ is incomplete.

Lemma 3 *$G[S_a(t_k)]$ is complete $\Leftrightarrow$ no unit switches OFF at t_{k+1}.*

Proof
"=>" If $G[S_a(t_k)]$ is complete, then by definition, no unit can switch OFF at t_{k+1}.
"<=". Assume that no unit switches OFF at t_{k+1}. We have to show that $G[S_a(t_k)]$ is complete.

Assume $G[S_a(t_k)]$ is incomplete. By Lemma 2, some unit must switch OFF. This contradicts our assumption. Hence $G[S_a(t_k)]$ must be complete.

Lemma 4 *This lemma has two parts.*

1. *$G[S_a(t_k)]$ is complete but not a clique $\Leftrightarrow$ some unit x switches ON at t_{k+1}.*
2. *If a unit, x switches ON, then $G[S_a(t_k) \cup \{x\}]$ is complete.*

Proof
1.
"=>". Assume $G[S_a(t_k)]$ is complete but not a clique. We have to show that some unit x switches ON.
By Lemma 3, no unit can switch OFF.
Since $G[S_a(t_k)]$ is not a clique, $\exists x \in D - S_a(t_k)$, such that $G[S_a(t_k) \cup \{x\}]$ is complete. Hence, x has $+ve$ connections from all the units in $S_a(t_k)$. Suppose no other unit switches ON. Then x will switch ON.
"<=". Assume some unit $x \in D - S_a(t_k)$ switches ON. We have to show that $G[S_a(t_k)]$ is complete but not a clique.

By Lemma 3, $G[S_a(t_k)]$ is complete. Since s_m is the size of the maximum clique,

$$\begin{aligned} |S_a(t_k)| &\leq s_m \\ &< I_m (\text{By assumption}) \end{aligned}$$

Suppose $G[S_a(t_k)]$ is also a clique. Then $G[S_a(t_k) \cup \{x\}]$ is incomplete. Hence, $\exists y \in S_a(t_k), w_{yx} = \rho$. Now x gets at most $|S_a(t_k) - 1|$ $+ve$ connections and at least one $-ve$ connection (from y). Hence,

$$\begin{aligned} n_x(t_{k+1}) &< |S_a(t_k)| + \rho \\ &< I_m - 2I_m \\ &< 0 \end{aligned}$$

Hence x cannot SWITCH ON at t_{k+1}. This is a contradiction. Hence $G[S_a(t_k)]$ cannot be a clique.
2.
In the above subproof, we arrived at a contradiction when we considered the case when x switches ON and $G[S_a(t_k) \cup \{x\}]$ is incomplete. It follows that if x switches ON, $G[S_a(t_k) \cup \{x\}]$ must be complete.

Lemma 5 *If $G[S_a(t_0)]$ is complete then $G[S_a(t_i)]$ is complete $\forall t_i > t_0$.*

Proof By induction. Let $G[S_a(t_k)]$ be complete. Then at t_{k+1}, no unit can switch OFF. Hence, either one unit $x \in D - S_a(t_k)$ switches ON or no unit switches. If no unit switches, $G[S_a(t_{k+1})]$ is complete. If a unit x switches ON, then by Lemma 4, $G[S_a(t_{k+1})]$ is complete.

Lemma 6 *$S_a(t_k)$ is a resting state of the network $\Leftrightarrow$ $G[S_a(t_k)]$ is a clique. In other words, there is a 1-1 correspondence between the fixed points of the network and the cliques in the underlying graph G_n.*

Proof Let s_f denote the set of units active at a particular fixed point (resting state). Since no unit switches off when the network has s_f as the only active units, by Lemma 3, $G[s_f]$ is complete. Since $G[s_f]$ is complete and no unit switches ON, By Lemma 4, $G[s_f]$ is a clique.

Theorem 3 *If no unit switches OFF at time t_i, then no unit switches OFF at any time $t_j > t_i$.*

Proof Since no unit switches OFF at t_i, by Lemma 3, $G[S_a(t_{i-1})]$ is complete. Hence by Lemma 5, $G[S_a(t_j)]$ is complete $\forall t_j > t_{i-1}$. Again, by Lemma 3, no unit can switch OFF at any $t_k = t_{i+1}, \ldots, t_j$.

Corollary 2 *No unit switches more than twice*

Proof This follows directly from Thm 3. If x switches ON at t_i, then it can never switch OFF at any $t_j > t_i$. If x switches OFF at t_i, then suppose it can switch ON again at some $t_k > t_i$. Then it cannot switch OFF again at any $t_j > t_k$.

Theorem 4 *Let the input, $S_a(t_0)$, be presented to the network at time t_0. The network has the following dynamics in the following order.*

1. *The network switches OFF units successively in the first $k \geq 0$ steps until the first time $G[S_a(t_k)]$ becomes* complete. *At that time, $|S_a(t_k)| = |S_a(t_0)| - k$.*
2. *The network, then switches ON units successively in the next $j \geq 0$ steps until $S_a(t_{k+j})$ is a clique. At that time, $|S_a(t_{k+j})| = |S_a(t_k)| + j$.*

The network rests at $S_a(t_{k+j})$.

Proof: If $G[S_a(t_0)]$ is incomplete, a unit will switch OFF. Apply Lemma 2 as many times as necessary, until the first time, t_k, that $G[S_a(t_k)]$ becomes complete. At each step, a single unit is switched OFF and hence $|S_a(t_{i+1})| = |S_a(t_i)| - 1$. When $|S_a(t_k)| = 1$, $G[S_a(t_k)]$ is always complete. It follows that $k < |S_a(t_0)|$ and $|S_a(t_k)| = |S_a(t_0)| - k$.
At this point, $G[S_a(t_k)]$ is complete. If $G[S_a(t_k)]$ is not a clique, By Lemma 4, some unit switches ON. Apply

Lemma 4 at every step, for j more steps, until $G[S_a(t_{k+j})]$ is a clique. By Lemma 6, $S_a(t_{k+j})$ is a stable state. Let $S_c \supseteq S_a(t_k)$ be the largest clique which contains $S_a(t_k)$. At each of the j steps, a single unit is switched ON. Hence, $j \leq |S_c| - |S_a(t_k)|$. Also, $|S_a(t_{k+j})| = |S_a(t_k)| + j$.

Vector Formulation of Theorem 4

Let $\bar{Y} = (\bar{y}_1 \bar{y}_2 \ldots \bar{y}_n)$ be the matrix of fixed points of the network. Let $\bar{x}$ be an input to the network. Then the trajectory of the network is characterised by $\bar{z}$ and some $\bar{y}_i$ in $\bar{Y}$ such that

$$\bar{z}^T \bar{x} = \bar{z}^T \bar{y}_i = |\bar{z}|$$

The choice of $\bar{z}$ and $\bar{y}_i$ depends on the network weights. The network first reaches $\bar{z}$ from $\bar{x}$ and then $\bar{y}_i$, the resting state, from $\bar{z}$.

Corollary 3 *The network, in general, does not use a* minimum Hamming distance *criterion.*

We only need a counter-example. Let s_i be the initial input set. Let s_j be the only fixed point such that $s_i \subset s_j$. Let s_k be another fixed point closer in Hamming distance to s_i than s_j is to s_i. In other words, $|s_i \Delta s_k| < |s_j \Delta s_i|$ where Δ denotes symmetric difference. Clearly, s_i is not a subset of s_k. By Thm 4, from the initial state s_i, the network will always reach s_j. But a Hamming-distance minimizer would reach s_k.
Example: The following example illustrates this property. Suppose the network is used to store words (see section 'Application') and the words *plug, plagiarize* are stored (among others). Given the input { p1 l2 g4 z9 } to the network, it will retrieve *plagiarize* and not *plug*.

Lemma 7 *If $G[S_a(t_i)]$ is incomplete, then at time t_{i+1}, some unit $s_i \in S_{a_{inh}}(t_i)$ inhibited by the largest number of units in $S_a(t_i)$ will be switched OFF.*

Proof By Lemma 2, a unit must switch OFF at time t_{i+1}. Let s_i be a unit with the largest number of inhibitory connections, say m, from the set of units $S_{a_i}(t_i) \subset S_a(t_i)$. Let s_j be a unit with at most $m-1$ inhibitory connections from the set of units $S_{a_j}(t_i) \subset S_a(t_i)$. We have to show that s_j cannot switch OFF at t_{i+1}.

$$\begin{array}{rcl} n_i & = & m\rho \ + \ \sum_{k \in S_a(t_i) \ - \ S_{a_i}(t_i)} w_{ik} \\ n_j & \geq & (m-1)\rho \ + \ \sum_{k \in S_a(t_i) \ - \ S_{a_j}(t_i)} w_{jk} \end{array}$$

Now, $\rho = -2I_m$. Since $G[S_a(t_i)]$ is incomplete, $|S_a(t_i)| \leq I_m$. Each $+ve$ weight is ≤ 1.
It follows that,

$$\begin{array}{rcl} n_i & < & (m-1)\rho \\ & \leq & n_j \end{array}$$

Hence s_j cannot switch OFF. In effect, the energy descent rule reduces to selecting the s_i which minimizes

$$\Delta E_{i \in S_{a_{inh}}} = \sum_{j \neq i, j \in S_{a_{inh}}} w_{ij} \tag{8}$$

Lemma 8 *The time to reach a fixed point (retrieve a memory) is $\leq s_m + I_m - 2 \leq 2n - 2$ steps.*

Proof Let t_f be the time the network rests. By Thm 4, the number of steps in the trajectory is $|S_a(t_0) - S_s| + |S_a(t_f) \ - \ S_s|$. This sum is maximised by minimizing $|S_s|$ and maximising $|S_a(t_0)|$ and $|S_a(t_f)|$. $|S_a(t_0)| \leq I_m$ since I_m represents the maximum size of any input. $|S_a(t_f)| \leq s_m$ since s_m represents the maximum size of any fixed point. $|S_s| \geq 1$ since at least one unit must always be ON. Hence the number of steps is $\leq I_m - 1 + s_m - 1$. Since $I_m, s_m \leq n$, this is always $\leq 2n - 2$.
The next Lemma overrides the assumption that units are not clamped, by considering, explicitly, the case when they are clamped to 1. Proposition-II of the Lemma requires the additional assumption that $\rho \leq -(I_m + s_m)$.

Lemma 9 *Let $S_a(t_0)$ be the input to the network at time t_0 and $S_{a_c}(t_0) \subseteq S_a(t_0)$ be the set of units clamped to 1 in the input. Let $S_a(t_f)$ be the state at which the network rests at time t_f. Then*

1. *$G[S_a(t_f) \ - \ S_{a_c}(t_0)]$ is complete.*
2. *Each unit in $S_{a_c}(t_0)$ is adjacent to all units in $S_a(t_f) \ - \ S_{a_c}(t_0)$.*

Proof

1. Suppose $S_a(t_f) \ - \ S_{a_c}(t_0)$ is not a complete subgraph. By Lemma 2, some unit in $S_a(t_f) \ - \ S_{a_c}(t_0)$ must switch OFF at time t_{f+1}. This is a contradiction.
2. Suppose some unit $s_1 \in S_{a_c}(t_0)$ is not adjacent to some unit, $s_2 \in S_a(t_f) \ - \ S_{a_c}(t_0)$. Then, $w_{s_1 s_2} = \rho \leq -(I_m + s_m)$. Hence

$$n_{s_2}(t_{f+1}) \leq |S_a(t_f) \ - \ S_{a_c}(t_0)| \ + \ (|S_{a_c}(t_0)| \ - \ 1) \ + \ \rho$$

Since s_m is the size of the largest clique and I_m is the size of the largest input, it follows that

$$|S_a(t_f) \ - \ S_{a_c}(t_0)| \leq s_m$$

and

$$|S_{a_c}(t_0)| \leq I_m$$

Hence,

$$n_{s_2}(t_f+1) \;<\; (I_m + s_m) \;+\; \rho \leq 0$$

It follows that s_2 must switch OFF at t_{f+1}. But this is a contradiction.

Implication: The network, in the resting state, keeps only those units active (other than the clamped units) that are in the intersection of some collection of memories in which all clamped units belong. To say it less formally, if the clamped units are in different memories, then the resting state will be the intersection of those memories, in addition to the clamped units. This property can be used effectively to find the intersection of a set of memories, from their *distinctive* features.

A Probabilistic Perspective on Learning and Recall

Additional insight can be gained into the network's learning and recall behavior by comparing it's learning rule with Bayesian learning and it's recall behavior with Bayesian inference.

Learning

Consider the n symbols in $D \cup \{1,..k\}$ as features in an n-dimensional space. Each unit stores the value of a feature. Since the units are binary, only binary features can be handled. The features, however, can be given different confidence values (in the interval $[0,1]$) in the input. A particular feature vector, denoted by x, is the vector representation of the corresponding set S_x on $D \cup \{1,...k\}$. Let S_p denote the sequence of pattern presentations, including repeated presentations of the same pattern, made to the network during training. S_p is a sequence whose elements are feature vectors, or alternatively, sets on D. Let α denote the total number of presentations, that is $\alpha \;=\; |S_p|$. Let c_i denote the occurence frequency of the i_{th} feature in S_p and c_{ij}, co-occurence frequency of the i_{th} and the j_{th} feature ($j \neq i$), in the same element of S_p. Then, after α presentations, the learning rule (Eqns 2,3) makes the weights take the following values.

$$w_{ii} \;=\; \frac{2}{1+e^{-\lambda c_i}} - 1 \tag{9}$$

$$w_{ij} \;=\; \begin{cases} \frac{2}{1+e^{-\lambda c_{ij}}} - 1 & \text{if } c_{ij} > 0 \\ \rho & \text{otherwise} \end{cases} \tag{10}$$

where w_{ii} is the self-weight of the i_{th} unit and w_{ij} is the weight between the i_{th} and the j_{th} units. Eqn 9 is similar to

$$P(x_i = 1) = c_i/\alpha$$

, the probability of occurence of the i_{th} feature in nature. Specifically,

$$\forall i,j, \quad w_{ii} > w_{jj} \Leftrightarrow \log\ P(x_i = 1) \;>\; \log\ P(x_j = 1)$$

Also, both $w_{ii}(\alpha)$ and $\log\ P(x_i = 1)(\alpha)$ are monotonically increasing but with decreasing slope as a function of α.

The same observations hold, when $c_{ij} > 0$, for Equation 10 and

$$P((x_i = 1)\&(x_j = 1)) = c_{ij}/\alpha \tag{11}$$

, the probability of co-occurence of the i_{th} and j_{th} features in nature. When $c_{ij} = 0$, Equation 11 gives $-\infty$. w_{ij}, however, must always be a finite value. In [14], w_{ij} in such a case is set to $\log\ (1/\ \alpha)$. In my model, the value of w_{ij} is set to a different value for other reasons.

In fact, one of the most important aspects of my model is the value of w_{ij} when $c_{ij} = 0$.

It is found that the value of w_{ij} in such a situation should be selected so as to create strong lateral inhibition between the units, i and j. This is found to give the network high capacity and also, in conjunction with steepest descent, dynamics that is very different than the classical Hopfield network, dynamics that make this model more suitable than the Hopfield network for certain applications.

On the other hand, there is a very good reason why we want the least inhibition required (smallest value of ρ) that will suffice to produce the above properties. In order for analog implementations of this model to remain feasible, the range of values of weights must be tightly bounded. For digital implementations, the range is not so important, but, even there, the range does determine the number of bits required to store each weight. Since $|\rho| >> 1$, the range is clearly determined by the lateral inhibition, ρ.

I have shown that even if the network is large, but if, however, the memories to be stored are of small size, $|\rho|$ can be kept small (see Thm 4 and the Lemmas that preceed it).

Finally, the network also learns R_i, the internal resistance of each unit. After α presentations, the value of R_i is

$$R_i = \frac{2}{1 + e^{-\lambda_R c_i}} - 1 \tag{12}$$

$\lambda_R < \lambda$ so that the internal resistance learning rate is slower than the weight learning rate. Again, Eqn 12 exhibits behavior similar to

$$\log P(x_i = 1)$$

Recall

The network's global behavior can be decomposed into a sequence of local decisions, each local decision using information only from active units to switch some active unit OFF or some inactive unit ON. From Thm 4, we know that local decisions that switch units OFF are made before those that switch units ON. I will compare these two types of decisions individually with their Bayesian analogies.

Local switch OFF decisions

Recall that $S_a(t_i)$ is the set of features ON at t_i and $S_{a_{inh}}(t_i)$ is the subset of $S_a(t_i)$ containing those features that are inhibited by at least one feature in $S_a(t_i)$. Alternatively, $S_{a_{inh}}(t_i)$ is the set of features, such that

$$\forall x_i \in S_{a_{inh}}(t_i), \exists x_j \in S_{a_{inh}}(t_i) \quad \text{such that} \quad c_{ij} = 0$$

By Thm 4, if $|S_{a_{inh}}(t_i)| > 0$, then some unit $x_i \in S_{a_{inh}}(t_i)$ is switched OFF. By Lemma 7, x_i is the unit that is inhibited by the largest number of active units if $\rho \leq -2I_m$. I compare this decision with a Bayesian decision that switches the x_i OFF that minimizes

$$P(x_{i_{ON}} | S_a(t_i) - \{x_i\})$$

the probability that x_i should remain ON given the other units that are currently ON (active). By Bayes rule,

$$P(x_{i_{ON}} | S_a(t_i) - \{x_i\}) = \frac{P(S_a(t_i) - \{x_i\} | x_{i_{ON}}) \;\; P(x_{i_{ON}})}{P(S_a(t_i) - \{x_i\})}$$

The denominator is constant for every x_i and can be ignored. By assuming conditional independence of the features in $S_a(t_i)$,

$$P(S_a(t_i) - \{x_i\} | x_{i_{ON}}) = \prod_{j \in S_a(t_i) - \{x_i\}} P(x_{j_{ON}} | x_{i_{ON}})$$

Hence, the decision reduces to finding the $x_i \in x_{inh}$ that minimizes

$$P(x_{i_{ON}}) \prod_{j \in S_a(t_i) - \{x_i\}} P(x_{j_{ON}} | x_{i_{ON}})$$

which also minimizes

$$\begin{aligned} &\log(P(x_{i_{ON}}) * \textstyle\prod_{j \in S_a(t_i) - \{x_i\}} P(x_{j_{ON}} | x_{i_{ON}})) \\ \Leftrightarrow \quad &\log P(x_{i_{ON}}) \; + \; \textstyle\sum_{j \in S_a(t_i) - \{x_i\}} \log P(x_{j_{ON}} | x_{i_{ON}}) \end{aligned} \tag{13}$$

If $c_{ij} = 0$, let $P(x_{j_{ON}} | x_{i_{ON}})$ be set to ϵ where ϵ is a small positive number. This is done to keep $\log P(x_{j_{ON}} | x_{i_{ON}})$ a large negative value but not $-\infty$.
Clearly, since the *log* terms for anti-correlated features ($c_{ij} = 0$) dominate over the others, for a suitable choice of ϵ, Eqn 14 is minimized by the feature x_i that is anti-correlated with the largest number of active features x_j, that is $c_{ij} = 0$. As mentioned earlier, Lemma 7 shows that the network picks the same x_i.

Now suppose more than one unit receives the same maximum inhibition. Let x_{max} be the set of active units all receiving the same maximum amount of inhibition from other active units. The units in x_{max} will *compete* against switching OFF. The terms (in Eqn 14) that now account for finding the $x_i \in x_{max}$ that minimizes Eqn 14 are

$$\log P(x_{i_{ON}}) \; + \sum_{j \in S_{a_{i^+}}(t_i) - \{x_i\}} \log P(x_{j_{ON}} | x_{i_{ON}}) \tag{14}$$

where $S_{a_i+}(t_i) \subset S_a(t_i)$ is the set of active features positively correlated with x_i. By comparison, the terms that contribute to the energy in the energy-descent rule (Eqn 6) are

$$\Delta E_i \ = \ w_{ii} + \sum_{j \in S_{a_i+}(t_i)} w_{ij} \tag{15}$$

Eqns 14 and 15 are minimized in very similar ways and produce very similar decisions.

Local switch ON decisions

Now consider the case when a feature is switched ON. By Lemma 3, we know that this happens when all pairs of active units are positively correlated, that is $|S_{a_{inh}}| = 0$. The terms that contribute to the energy in the energy-descent rule in this case are

$$\Delta E_i \ = \ -R_i \ + \sum_{j \neq i, j \in S_a(t_i)} w_{ij} \tag{16}$$

The network's decision is to select the x_i that maximises Eqn 16. This decision can be compared with a Bayesian decision which selects the x_i that maximises

$$P(x_{i_{ON}} | S_a(t_i) - \{x_i\})$$

the probability that an inactive unit must switch ON, given the currently active units. By invoking Bayes rule and following the same derivation steps as in the switch-OFF case, the x_i is selected that maximises

$$\log P(x_{i_{ON}}) \ + \sum_{j \in S_a(t_i) - \{x_i\}} \log P(x_{j_{ON}} | x_{i_{ON}}) \tag{17}$$

It may be noted that the second term of Eqn 17 is quite similar to and has a contributing effect similar to that of the first term of Eqn 16. However, Eqn 16 has no term corresponding to $\log P(x_{i_{ON}})$, a measure of the *a priori* probability of x_i being ON. Also, Eqn 16 has the term R_i (whose value is proportional to $\log P(x_{i_{ON}})$), but is missing from Eqn 17. These two differences taken together, achieve exactly the opposite effect that $P(x_{i_{ON}})$ has on Eqn 17. Instead of preferring to switch a feature ON that has a high *a priori* probability of being ON, Eqn 16 has exactly the opposite effect. The reason is as follows. Features with large $P(x_{i_{ON}})$ are usually not critical to particular classes (memories), occuring, rather more commonly. Such features will, in general, correlate well with a larger number of features than those with lower *a priori* probability. By switching them ON too early, they can affect subsequent local decisions more than the features that were ON in the input. The examples after the Section on Energy-descent rule and in the Appendix show that sometimes this leads to undesirable results. On the other hand, if such non critical features are not switched ON immediately, then, by Thm 4, since none of the currently active units can switch OFF in any subsequent cycle, there is still a good chance that they will be eventually switched, if they correlate strongly with the currently active features.

R_i, hence, has the effect of delaying switching of non-critical features over those that are critical, even if the former has higher $P(x_{i_{ON}})$.

Clustering Property

Within the context of my work, I will adopt the following definition of clustering and related concepts. Clustering is a *grouping* of stored memories based on some similarity criterion so that *similar* memories are combined together to form a single new memory called the prototype of that cluster, which somehow represents the collection. The size of a cluster is the number of original memories in it's collection. Given an input, the system can only retrieve the prototype memories (or cluster identifiers), not any of the original memories.

Within this framework, CAM is obviously a special case of clustering where each cluster is of size 1 and has exactly one original memory. At the other extreme, there is only one cluster and all the memories are in that cluster. And, in between, there are interesting *natural* groupings which represent natural clusters at various levels. Given this framework, I have tried to find out if my model can go *smoothly* from CAM to the other extreme of clustering and discover it's grouping behavior. This exploration is driven by the observation that the value of a single parameter, ρ, might make this smooth transition from CAM to clustering. Clearly, when $\rho \leq -min(2(s_m+1), 2I_m)$, it does CAM. At the other extreme, when $\rho = 0$, I will show that it forms clusters such that each connected component of G_n is a distinct cluster. However, if in the input, a unit from each such cluster is ON, then the network will activate all the connected components of G_n, thus effectively, clustering them all together.

I will first describe the necessary condition for activating a single unit, as a function of ρ. We assume there are no size units.

Lemma 10 *Let $S_a(t_i)$ be the set of units ON at t_i. Then at t_{i+1}, if a unit $x \in D - S_a(t_i)$ is switched ON then*

$$\frac{d(x)|S_a(t_i)}{\bar{d}(x)|S_a(t_i)} \quad > \quad -\rho$$

Proof: $\bar{d}(x)|S_a(t_i)$ is the number of inhibitory connections, each of value ρ, x gets from $S_a(t_i)$, the units active at t_i. $d(x)|S_a(t_i)$ is the number of $+ve$ connections, each of value ≤ 1, that x gets from $S_a(t_i)$. Hence

$$n_x(t_{i+1}) \quad \leq \quad \rho \ \bar{d}(x)|S_a(t_i) \ + \ d(x)|S_a(t_i)$$

Since x switches ON, $n_x(t_{i+1}) > 0$. Hence,

$$\frac{d(x)|S_a(t_i)}{\bar{d}(x)|S_a(t_i)} \quad > \quad -\rho$$

Remark 1 *When $\rho = 0$, $d(x)|S_a(t_i) \geq 1$ will satisfy Lemma 10, that is, it is sufficient for x to have a $+ve$ connection to at least one active unit (in $S_a(t_i)$) to meet the criterion set by Lemma 10. This, as we shall see, is the case for connected components.*

Remark 2 *When $rho \leq -min(2(s_m+1), 2I_m)$, it has already been shown that the network does CAM, in particular, By Lemma 4, $d(x)|S_a(t_i)$ must be equal to $|S_a(t_i)|$, in other words, x must have a $+ve$ connection to all the active units. This result falls out also as a special case of Lemma 10.*

For the case, $\rho = 0$, we can now prove the following lemma. Recall that the k connected components (maximal connected subgraphs) of G_n are denoted by $CC_1, CC_2, \ldots, CC_k$.

Lemma 11 *Let G_n have k connected components after training. Let $S_a(t_0)$ be the input to the network and let $\rho = 0$. Then the resting state of the network is $CC_{i_1} \cup CC_{i_2} \ldots \cup CC_{i_j}$ such that*

$$S_a(t_0) \cap CC_{i_m} \neq \phi, 1 \leq m \leq j$$

and

$$S_a(t_0) \cap CC_l = \phi, \quad l \text{ not in } \{i_1, \ldots, i_j\}$$

In plain English, the network activates all and only those connected components that have at least one unit active in the input.

Proof : If no unit in the input is also contained in a connected component, say CC_i, then, no unit in the input has a $+ve$ connection to any unit in CC_i. Hence, no unit in CC_i can be switched ON at any time.

Now we have to show that if at least one unit in the input is also in a CC, say CC_j, then all units in CC_j will be ON in the final state. Suppose not. Let t_f be the time the network rests and let

$$S_{CC_j}(t_f) \ = \ S_a(t_f) \ \cap \ CC_j$$

be the set of units from CC_j that are ON at t_f. Since $S_a(t_0) \cap CC_j \neq \phi$ and when $\rho = 0$, units never switch OFF,

$$|S_{CC_j}(t_f)| \geq 1$$

Also, by our assumption, $S_{CC_j}(t_f) \neq CC_j$. But since CC_j is a connected component, $\exists x \in CC_j - S_{CC_j}(t_f)$ which is adjacent to some $y \in S_{CC_j}(t_f)$. But then $n_x(t_{f+1}) > 0$ and x can switch ON at t_{f+1} which is a contradiction.

Corollary 4 *When $\rho = 0$, if an input, $S_a(t_0)$, is a subset of a connected component, CC_i, the network rests at CC_i.*

This follows directly from Lemma 11. No proof is required.
Under the above condition, we might say that (when $\rho = 0$), the clusters formed are the connected components in G_n, and each cluster represents the collection of memories $S_1, \ldots, S_m$ such that $G[S_1 \cup S_2 \ldots \cup S_m]$ is connected. The prototype memory of each cluster, is then, just

$$S_1 \cup S_2 \ldots S_m$$

Now we go back to the general case of any ρ and prove a property about the final state.

Lemma 12 *Let t_f be the time when the network rests and $S_a(t_f)$ be the set of units ON in the final state. Then* $\forall x \in S_a(t_f)$

$$\frac{d(x)|S_a(t_f)}{\bar{d}(x)|S_a(t_f)} \quad \geq \quad -\rho \tag{18}$$

Proof: $\bar{d}(x)|S_a(t_f)$ is the number of inhibitory connections, each of value ρ, x gets from $S_a(t_f)$, the units active at t_f. Since no unit in G_n has self-weights, this formula also holds if $x \in S_a(t_f)$. $d(x)|S_a(t_f)$ is the number of $+ve$ connections, each of value ≤ 1, that x gets from $S_a(t_f)$.
Suppose, $\exists x$ such that

$$\frac{d(x)|S_a(t_f)}{\bar{d}(x)|S_a(t_f)} \quad < \quad -\rho$$

But then,

$$\begin{aligned} n_x(t_{f+1}) \quad &\leq \quad \rho \; \bar{d}(x)|S_a(t_f) \; + \; d(x)|S_a(t_f) \\ &< \quad 0 \end{aligned}$$

Hence x must switch OFF at t_{f+1}, which is a contradiction.
Note: Eqn 18 is equivalent to

$$\frac{d(x)|S_a(t_i)}{\bar{d}(x)|S_a(t_i) \; + d(x)|S_a(t_i)} \geq \frac{\rho}{\rho - 1}$$

and written in this form indicates that the minimum fraction of active units that each active unit is $+ve$-ly correlated with, in the final state, is $\frac{\rho}{\rho - 1}$.

Self-scaling

An example shows how the network exhibits self-scaling, which is the capability to discriminate between memories as a function of their size. Small memories will be differentiated even on the basis of small differences, large memories require large differences to be considered distinct. This property requires the value of ρ to be in the range when clustering occurs ($-s_m < \rho \leq 0$).
Example: Consider two stored memories S_i and S_j and let $|S_i \; - \; S_j| \; = \; |S_j \; - \; S_i| \; = \; 1$. One example is

```
Si = { a, b, c }    Sj = { b, c, d }
```

We will assume that the weights of $G[S_i], G[S_j]$ are saturated, that is, all the $+ve$ weights S_i, S_j affect have the value 1. This can be easily accomplished by repeatedly presenting S_i, S_j during training.

Given the value of ρ, I will derive the size of S_i at which the network behavior changes. Let $S_i \; - \; S_j = \{x\}$ and $S_j \; - \; S_i = \{y\}$.

$$d(x)|S_i \cup S_j \; = \; d(y)|S_i \cup S_j \; = \; |S_i| \; - \; 1 \quad , \quad \bar{d}(x)|S_i \cup S_j \; = \; \bar{d}(y)|S_i \cup S_j \; = \; 1$$

By Lemma 12, x and y cannot be ON together in the final state if

$$\begin{aligned} -\rho \quad &> \quad \frac{d(x)|S_i \cup S_j}{\bar{d}(x)|S_i \cup S_j} \\ &= \quad |S_i| \; - \; 1 \end{aligned}$$

that is,

$$|S_i| < -\rho + 1$$

In other words, given ρ, memories of size less than $-\rho \; + \; 1$ will always be considered distinct even if they differ from another memory in only one symbol.

Now consider the case when $|S_i| > -\rho + 1$. Since we've assumed that each $+ve$ weight has value 1,

$$\begin{aligned} n_x \quad &= \quad \rho \; \bar{d}(x)|S_i \cup S_j \; + \; d(x)|S_i \cup S_j \\ &= \quad \rho \; + \; (|S_i| \; - \; 1) \\ &> \quad 0 \end{aligned}$$

The same applies to n_y. Hence, both x and y are active in the final state.

Hierarchical clustering

After a collection of (say M) memories is stored in the network, the value of ρ can be changed and the changes in the clusters that form (merge, split or new) can provide useful information about the structure of the memories, or rather , the structure of G_n.

Consider a CC, say CC_i. When $\rho = 0$, CC_i is one cluster. Suppose, when $\rho = -min(2(s_m+1), 2I_m)$, CC_i is still one cluster, then, clearly, the connected component CC_i is a clique, a very dense subgraph of G_n. On the other hand, if CC_i splits up into m clusters where $m \approx |CC_i|$ then clearly, CC_i is a very sparsely connected connected component. Similar analyses can be done by observing how the clusters change based on small changes to ρ. In general, if a small decrease in the value of ρ causes a cluster to break up into many parts, then the subgraph corresponding to the original cluster has pockets (subgraphs) of slightly higher than minimum density.

Implementations of this network might consider allowing dynamic changes to ρ, making it possible to perform hierarchical clustering dynamically. Noting the fact that the only negative weights in the network are of size ρ may make the implementation simpler, in principle.

Stability of clusters

Finally, let me mention a few other properties that accompany the clustering behavior exhibited by this network. Firstly, the number of clusters formed is always finite since it is $\leq$ the number of original memories M which is finite. Secondly, for constant ρ, as new memories are added to the network, the size of clusters and the number of clusters can only increase, neither can ever decrease. This is because new memories can only add additional edges in the underlying graph G_n, causing either an existing cluster to grow in size, or a new cluster to form after every such addition (of a memory). If we now map each cluster prototype to a point in the n-dimensional space of units, then the cluster *prototype vectors* can only move in one direction after new memories are added. Also, the number of points (clusters) can only increase. Hence, the clustering is *stable* in the sense of [9].

Finally, I would like to explore what happens when units are clamped in the input. Clearly, we would like to see if we can generalise Lemma 9 and write it as a function of ρ. We will, however, restrict ourselves to the case when all units in the input are clamped to 1 and they are all mutually inhibitory.

Lemma 13 *Let $S_a(t_0)$ be the input to the network at time t_0 in which all units are clamped to 1 and they are all mutually inhibitory. Let $S_a(t_f)$ be the state at which the network rests at time t_f. Then*

$$\forall x \in S_a(t_f) \; - \; S_a(t_0), \qquad \frac{d(x)|S_a(t_f)}{\bar{d}(x)|S_a(t_f)} \;\; \geq \;\; -\rho$$

Proof

1. Suppose $\exists x \in S_a(t_f) \; - \; S_a(t_0)$,

$$\frac{d(x)|S_a(t_f)}{\bar{d}(x)|S_a(t_f)} \;\; < \;\; -\rho$$

By Lemma 12, x must switch OFF at time t_{f+1}. This is a contradiction.

Implication: Recall, for CAM, that the (unclamped) units active in the resting state are in the intersection of some collection of memories containing all the clamped units, but where the clamped units themselves may lie outside the intersection. But when ρ is in the range when clustering takes place, the (unclamped) units active in the resting state need correlate $+ve$-ly only with a fraction of the other active units. Lemma 11 shows that when $\rho = 0$ the active units are in the union of the connected components. Hence, by varying ρ, we can go *smoothly* from finding the intersection of a set of memories to it's union.

Domains with Combinatorial structure

The Hopfield-style network described above is especially suited to a class of applications whose domain (and the memories to be stored) have a particular structure described below. It is shown that under these circumstances, the network stores all memories perfectly, effectively having exponential capacity. This is not to say, that no *spurious* memories will develop, on the contrary, it has been experimentally shown that spurious memories increase dramatically as the number of stored memories does.

Domains with such structure include lexicons (word dictionaries), phonemic transcriptions of words, finite length ($\leq n$) sequences, and relational data bases. It is also shown that any domain can be transformed into one with such a structure.

Defn 1 *Let D be a domain. Then if D can be partitioned into disjoint sub-domains $D_1, \ldots, D_m$ and $D = D_1 \cup D_2 \ldots D_m$ such that the only stored memories are ordered sets $s_i \in D_1 X D_2 \ldots D_j$, perhaps different j but $1 \leq j \leq m$, then D has a combinatorial structure of partition size m.*

One important special case is when all the D_i are the same domain, D_d. Then D can be represented as a 2-d domain $D_d X D_n$, one dimension for D_d and the other for $D_n = 1, 2, ...n$.

Ordered sets, like s_i can be stored in the network only as unordered sets (since the network cannot preserve order information). This can be done by ensuring that the symbols in all the sub-domains are distinct by renaming them if necessary.

Examples:

```
D = { a1, b1, c1, a2, b2, c2, a3, b3, c3 }
```

If we choose

```
D1 = { a1, b1, c1 } D2 = { a2, b2, c2} D3 = {a3, b3, c3 }
```

then $\{ b1 \}, \{ a1,\ a2 \}$, $\{ a1,\ b2,\ c3\}$ are examples of legal sets that can be stored but $\{ a1,\ b1 \}$, $\{a1,\ c1 \}$ are not.

Lemma 14 *Irrespective of the memories stored, connections between units associated with two symbols from the same domain D_i are always inhibitory.*

Proof : Assume $\exists D_i, 1 \leq i \leq n$ such that $x, y \in D_i$ and w_{xy} is not inhibitory. Then there must be a stored memory , s, which has both x and y. But by definition, s can have at most one symbol from each sub-domain.

Lemma 15 *Let $s_1, s_2, \ldots, s_k$ be some collection of the stored memories. If they don't all have the same size, then in G_n, the graph underlying the network, $s_1 \cup s_2 \ldots \cup s_k$ cannot be a* clique.

Proof : Let $\exists s_i, s_j, 1 \leq i, j \leq k$ such that $|s_i| \neq |s_j|$. Let sz_i, sz_j be size units for s_i, s_j respectively. Since $|s_i| \neq |s_j|$, sz_i and sz_j are different size units. Since no memory can have both sz_i and sz_j, w_{sz_i, sz_j} is always inhibitory, that is **not** adjacent in G_n. Hence, since $sz_i, sz_j \in s_1 \cup \ldots \cup s_k$, $s_1 \cup \ldots \cup s_k$ is not a clique.

Lemma 16 *Let $s_1, \ldots, s_k$ be some of the stored memories. If they are all of the same size, $s_1 \cup s_2 \ldots \cup s_k$ cannot be a clique in G_n.*

Proof : Pick any two memories s_i, s_j. Since they are of the same size, they must have different symbols in at least one domain, say D_l. Let $\Pi_i(D_l), \Pi_j(D_l)$ denote the symbols from D_l in s_i, s_j respectively, $\Pi_i(D_l) \neq \Pi_j(D_l)$. By Lemma 14, $\Pi_i(D_l)$ *and* $\Pi_j(D_l)$ are mutually inhibiting, and hence, not adjacent. But, since $\Pi_i(D_l), \Pi_j(D_l) \in s_1 \cup s_2 \ldots \cup s_k$, it follows that $s_1 \cup s_2 \ldots s_k$ cannot be a clique.

Theorem 5 *If D has a combinatorial structure,then the network doesn't lose any stored memory.*

Proof: We have to show that the only cliques in the network are those that correspond to stored memories. Since training the network with new memories can make connections only more $+ve$, and cliques correspond 1-1 to maximal complete subgraphs of $+ve$ connections in the network, it follows that the only way a network can lose a stored memory is if a larger clique develops that subsumes it. More formally, this can happen only if there exists some collection of stored memories $s_1, s_2, \ldots, s_k$ such that their union is also a clique. But By Lemmas 15,16 this is not possible.

Lemma 17 *The network can develop spurious memories but their size is $\leq (m+1)$.*

proof Suppose a spurious memory, s, forms of size $\geq (m+2)$. Since D has m sub-domains, and one more for size units, it follows that there is some sub-domain D_i or the sub-domain representing size units, that has at least two elements from s. But that is a contradiction since no memory can have more than one symbol from each sub-domain.

Implication : The above, almost trivially obvious Lemma is quite useful because it allows us to bound s_m, the size of the maximum clique, to $m+1$. As we've seen earlier, this allows us to set the value of ρ. It is important to realise that such a bound can only be placed when D has a combinatorial structure. If not, then since collections of memories can collapse to form larger cliques (see examples in Section 'Storage Properties'), such a bound cannot be placed *a priori*.

Lemma 18 *If D has exactly two combinatorial components D_1, D_2 then no spurious memories can develop.*

Proof: For a spurious memory to develop, one of two kinds of $+ve$ edges must be shared by the spurious memory and some other stored memory. $(x,1)$ or $(x,y), x \in D_1, y \in D_2$. The $(x,1)$ kind of edge can never be shared because it uniquely stands for the memory $\{x,1\}$. Similarly, the (x,y) kind of edge cannot be shared by more than one memory because (x,y) is an edge only in one memory $\{x,y\}$. Hence no spurious memory can develop.

It was proved earlier (Thm 1) that if only k-ary sets are to be stored in the network, and we are allowed to choose a collection that maximizes storage, then we can choose $(n/k)^k$ such sets (assuming n is a multiple of k). This also follows directly from combinatorially structured domains. Consider an application in which D has a combinatorial structure and k sub-domains of equal size. Hence $\forall i, \quad |D_i| = n/k$. Since we are storing k-ary sets, the number of possible memories is $|D_1 X D_2 \ldots D_k| = (n/k)^k$ and since D has a combinatorial structure, all memories are stored cleanly.

Relaxing multiple competing hypotheses

Let $D = D_1 \cup D_2 \cup D_3 ... \cup D_m$ be a combinatorially structured domain and consider a test input in which many (0, 1 or many) units are ON from each sub-domain D_i. The task of the network is to settle into a *suitable* final state in which exactly one unit is ON in the first j domains. Since D is combinatorial, units in each sub-domain are mutually inhibiting. By Thm 4, the network will first switch OFF units until all the active units are mutually consistent. When this happens, no sub-domain can have more than one unit ON. This process can be interpreted as the relaxation of multiple competing hypotheses. But some sub-domains may have no unit ON. The 2nd part of Thm 4 tells us that units will be switched ON until the network is at a fixed point. The overall behavior can be characterised as relaxation of multiple hypotheses followed by "filling in" missing details.

Example: The following words are stored. *maple, blvd, buffalo.* The test input is { m1,b1,a2,l2,v3,p3,l4,e5 }. The network will probably retrieve *maple.*

Now, we come to one of the most important results that shows that any domain can be transformed into one with a combinatorial structure. This shows, effectively, that all possible memories, on any domain, can be stored.

Theorem 6 *Any collection of n-dimensional binary vectors can be stored in a network containing $2n$ units without* losing *any stored vector. The, network, therefore, has exponential capacity, 2^n.*

Proof: Construct the domain D as follows.

$$D = \{(v,p) \mid v \in \{0,1\} \ , \ y \in [1,n] \ \}$$

Hence, D has symbols for both values (0,1) at each position of the n-dimensional binary space. D can be partitioned into n sub-domains,

$$D = D_1 \cup D_2 \cup \ldots \cup D_n \tag{19}$$

where

$$D_i = \{(0,i),(1,i)\}$$

Clearly, the set, s, corresponding to any stored vector must be such that

$$s \in D_1 \ X \ D_2 \ X \ \ldots \ X \ D_n$$

D has a combinatorial structure and hence, by Thm 6, all memories can be stored cleanly. Since there are 2^n possible binary vectors, the network capacity is 2^n.

Any domain D containing n symbols can be transformed into one with a combinatorial structure by converting each memory on D to its binary vector form and storing that form as a set as in Thm 6.

Since we know that the domain of Thm 6 has a combinatorial structure, we can in fact even store binary vectors of many dimensions (from 1 to n) all in the same network by using n additional size units.

In the scheme outlined by Thm 6, spurious minima can still develop as shown by the following example. If we store the 3-dimensional vectors

111 100 001

we will get 101 as a spurious memory.

Thm 6 tells us that we can store any collection of n-dimensional binary vectors cleanly. I will now discuss how one might store collections of real-valued n-dimensional vectors cleanly in this network.
Let

$$\bar{x}^T = (x_1, x_2, ..x_n) \qquad x_i \in [0, 1]$$

be a real-valued vector and

without loss of generality, assume that each component takes values in the interval $[0, 1]$ (as shown above). The $[0, 1]$ interval, for each component, say i, is individually quantized to create a domain D_i of values. The issue of quantization is a representation issue that won't be discussed further. We may note that the quantization may be different for different components (that is $|D_i|, |D_j|$ need not be the same), depending on the individual precisions required. It may not even be uniform in the $[0, 1]$ interval for a particular component. Let $D_1, \ldots D_n$ denote the domain of values for the $1^{st} \ldots n^{th}$ components after quantization. We then create a network whose units correspond to the domain

$$D = D_1 \cup D_2 \cup \ldots D_n$$

Clearly, D has a combinatorial structure. The vector $\bar{x}^T$, as shown above, is then stored by first quantizing each component value, denoted by $q_i(x_i)$ and then storing the following set

$$\{q_i(x_1), \quad q_i(x_2), \quad \ldots, \quad q_n(x_n)\}$$

as a memory.

An obvious disadvantage of the above scheme is that a large number of units may be required to represent the domain of values of a component. And since the number of connections is $O(n^2)$, n being the number of units, this can indeed be a problem especially when high precision is required and/or the vector dimension is large. Analog neural networks usually require only one unit per component, a unit having a real-valued state that can directly store any value of a component. Analog networks, however, have limited capacity (typically $O(n)$) and stored real-valued vectors start interfering with each other as this limit is reached. This is not the case with our network, where any collection of such vectors can be stored cleanly. Another advantage, almost as important, is that training is very rapid, typically one-shot (except for unlearning spurious memories). Analog networks typically require many iterations during training. Finally, our network exhibits rapid convergence and well-characterised dynamics (Thm 4). Analog networks typically require many cycles to converge.

One advantage of our network that is not even possible in analog networks is that the network can be used for relaxation of multiple competing hypotheses in the input. Since our network has different units for different domain values, the test input can contain multiple values for the same component. The task of the network is to settle to a *suitable* final state in which each component has exactly one value ON.

Even though it may appear otherwise, it is not my intention to suggest that discrete networks are superior to analog networks, on the contrary, it is my opinion that analog networks are more powerful and general-purpose. Yet, for certain specific applications, I claim that my model has very desirable properties that analog networks do not possess.

Structure of G_n

Finally, it is insightful to comment on the relationship between a domain with a combinatorial structure and the underlying graph in the network that it produces. I state, without proof, that if D has a uniform combinatorial structure, that is $|D_i| = D/m$, m being the partition size of D, then if **all** possible memories on D are stored, the underlying graph G_{n+m} has a subgraph that is $T_{m,n}$. G_{n+m} is $m+1$-partite, the first m partitions corresponding to the partitions in $T_{m,n}$ and the last partition (of size m) containing the m size units.

Applications

In this section, I present a brief summary of the results of extensive experimentation on one application of the model. For more details of the application and the experiments, see [12]. I also present results of short experiments on other applications. These have not been published elsewhere. These results provide additional insight into the network's properties.

Lexicon

In this application, we wish to store words in the network and recall a correct word from incorrect, imprecise or ambiguous information. This domain has a combinatorial structure, so any number of words can be stored cleanly.

That this was the case was also experimentally verified by storing upto $25,000$ words cleanly.
Example: Store the words, *cat, car, dim.*

```
{ c1 a2 t3 }     { c1 a2 r3 }    { d1 i2 m3 }
```

An example of test input is

```
{ c1 a2 }
```

That spurious memories can develop is illustrated by the following simple example. Store the words, *car, con, for.* A spurious memory *cor* will develop.

One of the main goals of this experiment was to ascertain the number of spurious memories that would develop, as a function of the number of stored words, and to test the effectiveness of the spurious memory unlearning procedure reported in [12]. Table 1.1 also gives an indication of how the network size scales for this application. The number of spurious memories is a relative measure, , since a simple procedure was used to detect them, not guaranteeing that all were detected. They are used only for relative comparisons. Table 1.1 shows, very roughly, that we can store upto 500 words quite reliably but as we move in the range of 1000 or more words, the number of spurious memories increases dramatically. λ was set to 0.15, λ_R to 0. ρ was in the CAM range ($[-40, -30]$).

Words	Units	# Spurious (Before)	# Spurious (After unlearning)
150	110	30	9
353	248	84	40
5873	489	4774	~4000
25622	533	19802	~18000

Table 1.1

Phonemic transcriptions of words

In this application, I wish to store and retrieve phonemic transcriptions of words in the network. This domain also has a combinatorial structure, so the network guarantees that any collection of memories will be stored cleanly.

The results shown below should not be interpreted as a serious attempt to do speech recognition, but rather as an example of how phonemic transcriptions can be mapped onto this network as a CAM application and an indication of how many spurious memories develop.

A database of phonemic transcriptions of words was obtained. Some examples of phonemic transcriptions from this database are as follows. The last field on each line shows how the transcription was mapped as a CAM memory.

```
ABASHED       AX  B  AE   SH   TD           {  AX1 B2 AE3 SH4 TD5 }
ABHORRENCE    AX  B  HH   AO   R    EN  S   {  AX1 B2 HH3 AO4 R5 EN6 S7 }
```

The network was trained with 1021 phonemic transcriptions of words, requiring 458 units. The training was very fast, about one minute (real-time, not cpu-time) on a sequential machine. The network was then tested by removing the phoneme at a particular position from each phonemic sequence. The total number of spurious memories reached from the test set was counted. Table 1.2 indicates the approximate number of spurious memories and the effectiveness of the unlearning procedure. Each column heading indicates the position of the phoneme that was removed from all the phonemic transcriptions that were used during training. The first row indicates the # of spurious memories before unlearning, the second row after unlearning. After the experiment was done, it was felt that the number of spurious memories was significantly less than those for the lexicon application for a similar training set size. To confirm this observation, the 1021 words from which the phonemic transcriptions were generated were stored in a separate network (requiring 338 units) and the spurious memories counted after the same number of unlearning cycles. The 3rd and 4th rows of Table 1.2 show the results. It appears that the phonemic application creates $15 - 20\%$ less spurious memories than the lexicon application both before and after the same number of unlearning cycles. The difference in the size of the networks may account for some, if not most of this difference.

Phonemes	Pos 1	Pos 2	Pos 3	Pos 4	Pos 5	Pos 6
	446	328	355	292	196	116
	237	193	160	122	82	47
Words	Pos 1	Pos 2	Pos 3	Pos 4	Pos 5	Pos 6
	592	578	511	450	412	317
	410	377	407	358	260	184

Table 1.2

In all of the experiments in Table 1.2, λ, λ_R, ρ were the same as in Table 1.1.

Network as a filter

If the network is used for CAM, spurious memories become a problem as the training set size is increased. We may observe, however, that the network's decisions can be interpreted at a finer level, that of individual units. Intuition would indicate that even when the network is at a spurious memory, many of the *desired* units are active. In other words, even though its macro-decision is completely wrong, a good fraction of it's micro-decisions are probably correct. If this is indeed true, the network will perform better as a filter that transforms an input set into a *better*, but not always correct, output set. And if the domain has a combinatorial structure, the filter has the desirable property that any correct input is passed through the network without any change.

To test the above theory, the following experiment was set up. The 1021 words associated with the phonemes in the phonemic application were used as the training set. 3 experiments were conducted, each row of Table 1.3 summarising the results of each. In the first experiment, the first 3 vowels from each word were removed and that constituted the test set. Table 1.3 shows that about 61% of the time, the network settled at a spurious memory. However, it filled in only 950 letters incorrectly out of 2421 missing in the test input, giving it a micro-decision error rate of 40%. In the second experiment, multiple hypotheses were added at each letter position to words in the training set and this constituted the test set. The network was used to *relax* constraints in each test input in an attempt to settle to a single letter at each position. This time the percentage of spurious memories reached was 31%, much lower than the previous case. By comparison, the micro-decision error rate ($\frac{442}{10607-7022}$) was 12 %. The third experiment had exactly the same setup except that the network's energy-descent was stopped before any unit could switch ON. Thm 4 shows that the network always switches units OFF before switching any unit ON. It would seem that one could take advantage of this property to provide an option in which only noise would be removed, no filling in would be done. Clearly, this makes the filter more conservative. This option can be trivially implemented by incorporating a single conditional check into the algorithm: Stop the energy-descent just before the first switch-ON decision (see algorithm in the *Implementation* section). The third experiment shows that the micro-decision error percentage drops to 9.5 % but now 7 % of the letter positions in all the words were left uncommitted (vacant).

Exp Type	# spurious	# hypotheses (initially)	#letters correct (finally)	#letters in error (finally)	#letters uncommitted (finally)
Vowels	621	-	1467	950	4
Multiple hypotheses	309	10607	7022	442	7
Multiple hypotheses (switch-OFF only)	-	10607	6820	362	289

Table 1.3

Again, λ, λ_R, ρ were as in Table 1.1.

Relational Database

This section is not intended to show a serious application of my model towards database management, only that certain kinds of databases have combinatorial structure and so can be stored in my network without losing any memories. If database management is to be seriously considered, we will need a two layer network (as discussed briefly below) a detailed description of which is beyond the scope of this paper.

The Relational database model is a very popular database model. A database is a collection of relations, each n-ary relation R is a set of n-tuples on n domains, $D_1, \ldots D_n$. More compactly,

$$R = \{t | t \in D_1 X D_2 \ldots D_n\}$$

A **key** of a relation R is a set of sub-domains on which R is defined whose values taken together *uniquely* identify any tuple of R.

Let

$$R \subseteq \underline{D_{k_1}} X D_2 ... X \underline{D_{k_j}} ... D_n$$

where $\underline{D_i}$ denotes that D_i is a domain in the key. Then construct a new relation R_k by projecting out only the key values of the tuples of R, that is

$$R_k = \Pi_{\underline{D_{k_1}}, \ldots, \underline{D_{k_j}}}(R)$$

Store R_k in our single layer network (denoted L_1) as follows. Let

$$D_k = D_{k_1} \cup D_{k_2} \cup .. D_{k_j}$$

denote the domain of distinct values of each element of the key. Clearly, D_k has a combinatorial structure. By Thm 5, any set of tuples in R_k can be stored cleanly (without losing any memory) in a network (L_1) requiring $|D_k|$ units. Tuples of R_k uniquely determine tuples of R. The non-key fields of R are stored in a second layer, L_2, which receives connections from L_1 that represent the functional dependencies from the key fields to the non-key fields. In other words, these connections are used to uniquely determine, given a tuple in R_k , the rest of the field values of the corresponding tuple in R. The reason for the breakup into two layers is to reduce the number of connections required at L_1, and hence, effectively for R, since R can be reconstructed from active units at L_1. To store R_k only at L_1, we need $|D_k|^2$ connections in L_1 whereas to store R at L_1, we would need $|D|^2$ connections in L_1. The issue of the connections required at L_2 and between L_1 and L_2 and the inter-layer operation of the network are beyond the scope of this paper.

We wish to discuss under what circumstances storing a relational database in our neural network (by storing R_k at L_1) is a desirable thing to do. Such a scheme should be considered for storage only if the key sub-domains constituting D_k are finite and fairly small, that is $|D_k|$ is well bounded, which is, however, not typically the case in practical database applications. Also, the network will have been utilised well if $|R_k| = O(\prod_i |D_{k_i}|)$, that is, R_k has a large number of tuples, of the order of the maximum possible, which is again typically not the case. There probably are some database applications, however, for which these two conditions are met. Even, if so, it must be noted that the retrieval characteristics of our network are quite different than those required of most database systems. Since we are confining our discussion to L_1, we will assume that the task (for comparison purposes) is to retreive tuples of R_k that match a *partial* specification of the key. In other words, given some values of parts of the key, to retrieve all those complete keys that match. Database systems typically retrieve all the tuples (0 to many) that match the input. Our network retrieves exactly one tuple that matches the input conditions (even if there are many or none that do). The network can , however, be suitably extended to retrieve all the tuples that match the input. It can also be easily modified to not filter out noise, in other words, if no tuple matches the input, then return none.

A more fundamental problem that can't be rectified that easily is that of spurious memories. In other words, there is no guarantee that the network will retrieve a tuple that was stored, unless the input matches that tuple exactly. The probability of *spurious* retrieval can be characterised experimentally but for most applications, it is not enough to do so.

If the goal, on the other hand, is, to retrieve the missing values of sub-domains that best correlate with the values in the input, based on the set of tuples that are stored in the network, then, of course, the network performs rather well. Another advantage, of course, is that retrieval is massively parallel and fast, within $O(n)$ steps, n being the number of sub-domains.

Implementation of Network Dynamics

Analog implementations have been considered for the Hopfield network and many lessons from those experiences can be transferred to this model. The main advantages if such a technology were successful would be the direct implementation of the *multiply* and *add* operations (in Eqns 6,7) by physical characteristics of analog devices , the possibility of incorporating learning directly into the analog elements that form the synapses and the potential for implementing the entire spectrum from continuous to discrete networks and from deterministic to stochastic networks (by adding analog noise) in a single architecture. This would provide enormous speed-ups and incorporation of the entire network into a single chip and allow the entire gamut of continuous/discrete units, step-wise/continuous/stochastic energy-descent behavior. The main problems appear to be the difficulty of constructing units that can take n inputs for large n, say (> 500), the difficulty of managing the total connectivity on a plane and the difficulty of building compact synapses capable of learning. Other possible problems include the inherent instability associated with analog circuits.

On the other hand, digital implementations are quite feasible and simpler to construct, but still suffer from n^2 connectivity requirements. If we limit ourselves to implementing only the energy-descent behavior, we can take advantage of the fact that our units are binary and energy-descent is step-wise to construct an algorithm that can be directly realised in specialised hardware, directly implemented on a vector machine or directly implemented on a sequential machine.

Assume an abstract vector machine capable of adding n-dimensional vectors, finding the position of the largest component and other vector operations in one step each. n is the dimensionality of our network. Let $S_a(t_0) = \{v_{i_1}.s_{i_1}, s_{i_2}^{*1} \ldots s_{i_k}\}$ be the set of features ON in the input. Features could have weights (v_{i_j}) or be clamped to 1 ($*1$) or clamped to 0 ($*0$). Let $\bar{p}$ be the vector of the values of each unit (p_i has a value in the interval $[0,1]$ or has the value $c = 1 + \epsilon$ to indicate that it is clamped to 1 or 0.) Assume the operation, *init*, initialises these values from the set $S_a(t_0)$ depending on whether the corresponding s_k has a weight v_k or if s_k is clamped to 1 or 0. Let $\bar{n}$ be the vector of the net inputs to all the units, $\bar{e}$, the vector of local energies of all units, $\bar{r}$, the vector of their internal resistances and $\bar{v}$ ($v_i = \{-1, 1, 0\}$), the vector of the directions in which they can switch. Also let $\bar{w}_i$ be the column vector of weights between s_i and the other units.

Energy-descent Algorithm:

$\bar{n} \leftarrow 0$; $\bar{p} \leftarrow \text{init}(S_a(t_0))$
<u>for</u> $s_i \in S_a(t_0)$ <u>do</u>
  $\bar{n} \leftarrow \bar{n} + p_i * \bar{w}_i$
<u>end</u> <u>for</u>
<u>loop</u>

$$j:1,n \quad v_j \leftarrow \begin{cases} 1 & n_j > 0 \& p_j < 1 \\ -1 & n_j < 0 \& p_j > 0 \\ 0 & p_j = c \\ 0 & \text{otherwise} \end{cases}$$

$j:1,n \quad e_j \leftarrow v_j * n_j - |v_j| * r_j$
  $k \leftarrow \text{position\_of\_max}(\bar{e})$
  <u>if</u> $e_k = 0$ <u>then exit</u>
  <u>if</u> $v_k = 1$ <u>then</u>
   $\Delta_p \leftarrow 1 - p_k$; $p_k \leftarrow 1$; $S_a(t_{m+1}) \leftarrow S_a(t_m) \cup \{s_k\}$
  <u>else</u>
   $\Delta_p \leftarrow 0 - p_k$; $p_k \leftarrow 0$; $S_a(t_{m+1}) \leftarrow S_a(t_m) - \{s_k\}$
  <u>endif</u>
 $\bar{n} \leftarrow \bar{n} + \Delta_p * \bar{w}_k$
<u>end</u> <u>loop</u>

Running Time There is 1 vector instruction before any loop, 1 inside the for loop and 4 vector instructions in the second loop. We assume that all except one such vector instructions can be done in one step. The exception is position_of_max, which we assume can be done in *logn* steps by using a binary tree circuit. Since $|S_a(t_0)| \leq I_m$, the assumed maximum size of any input, the for loop is repeated at most I_m times. By Lemma 8, the network converges in $\leq I_m + s_m - 2 < 2n - 2$ steps, hence the second loop is repeated at most $I_m + s_m - 2$ times. It follows that the running time during energy-descent is $\leq 1 + I_m + (3 + logn) * (I_m + s_m - 2)$. This is $O(nlogn)$, n being the number of units, and even better ($O(I_m logn)$) when the stored memories are of small size, that is, $s_m, I_m << n$.

Clearly, the above algorithm could be directly realised in specialised hardware capable of supporting the required vector operations in one cycle each. Since only vector operations are performed, the size of the circuitry would be linear in n. The for loop could be implemented by a shift register which stores only the *indices* of the units that are ON.

Similarly, the above algorithm can be directly implemented on a sequential machine by storing $S_a(t_k)$ in a linked list and serialising all the vector operations. Since serialising any vector operation should introduce $O(n)$ more steps, the total number of steps is on the *order* of $\leq n * (5I_m + 4s_m - 5)$ and since $I_m \approx s_m$, $O(I_m * n)$. Even on a sequential machine this algorithm is very fast, from linear to n^2 depending on whether $I_m << n$ or $I_m \approx n$.

Minimizing Spurious Memories

It would be fair to say that the most fundamental deficiency of my model (and single layer networks in general) is that spurious memories can develop and their rate increases dramatically with the number of stored memories. There may be many ways to attack this problem. One way is to use multiple layers, higher layers representing more complex features. Another approach is to extract and use higher order statistics by creating units for them in the same layer, so that one still needs only one layer. The following experiment was done to show the effectiveness of the latter approach. The network was trained with the 1021 words associated with the phoneme application. Unlearning was done, both of weights and internal resistance until the results of Table 1.5 were obtained. Then, the network was retrained on the words but this time, the training input comprised of the letters in a word plus the first few transitions. The word *cat*, for instance, was now presented to the network as follows.

```
{ c1 a2 t3 c1a2 a2t3 3 }
```

The size unit was explicitly included in the training input and the network was told not to compute it implicitly. Table 1.4 summarises the results. The first two rows should be compared with the first two columns of the second row of Table 1.5. The number of units used was 999, 338 (letter,position) units as before and 661 new transition units. The last row of Table 1.4 should be compared with the 1st row, 1st column of Table 1.3 (corresponding to Vowels). The test input was the same as used for that row in Table 1.3 except that transitions were added as appropriate. Adding transitions in this case reduces spurious memories from 621 to 492, an improvement of about 20%.

Words	Pos 1	Pos 2
	420	441
	258	279
	492	498

Table 1.4

The first row shows when the transitions were not included in the test input. For instance, the test input after Pos 2 was removed from the word *card* was { c1 r3 d4 }. The second row shows when the transitions were included. In the above example, the test input would be { c1 r3 d4 r3d4 }. The above experiment suggests that at least 10-15 % improvement is possible by including transitions both during training and testing.

The main advantage of such a representation is that it shows that spurious memories can be reduced by increasing the network size, without having to change the network architecture. The main disadvantage is that the increase in network size required may be prohibitively large for many applications. The transitions (and higher order correlations, in general), can be moved up to higher layers in a multi-layered version of this network, reducing considerably, the connectivity of the network. This approach will be pursued seriously. There are many generalisations of this approach that can be considered, including a self-organising network (single or multi-layered) that has a pool of units (in one or many layers) and assigns units to detect specific higher-order features based on a dynamically-adjustable estimate of their *utility*.

The above scheme and its variants are methods of *minimizing* the number of spurious memories that develop for a given application by representing higher order features and increasing network size and/or architecture to accommodate them. A more *fundamental* question is, given a fixed network size, *if* and to *what* extent the number of spurious memories can be reduced. In the following approaches, I used only a single layer network and simple feature representations and addressed the question of how and to what extent spurious memories could be unlearnt.

Reinforcing and Unlearning weights

The first approach I took was to *unlearn* weights. The results of some experimentation with this approach are summarised in the section on Applications (Table 1.2). The *unlearning* algorithm is rather straight-forward and is discussed in detail in [12]. It is quite effective when there are between 100-1000 stored memories. A brief description of the algorithm (in the context of the lexicon application) is as follows. Test the network with the trained words with the P_{th} letter removed from each word. Note all the spurious words reached, in other words, the words in which the P_{th} letter was filled in incorrectly. Unlearn the connections between the incorrectly filled in P_{th} letter in each spurious word and the rest of the letters of that word. This is done to ensure that unlearning does not affect all connections of the spurious word, only those connections that could possibly be responsible for the spurious state (since the network made only one wrong decision, filling in the P_{th} letter incorrectly). For each spurious word reached, reinforce the connections between the correct P_{th} letter and the rest of it's letters. Again, only those connections that could have switched ON the correct unit (P_{th} letter) are reinforced. Doing reinforcement in addition to unlearning is very important and gives much better results than unlearning alone.

Reinforcing and Unlearning internal resistance

The above algorithm is reasonably effective for moderate size training sets. This unlearning/reinforcement algorithm doesn't change R_i, the internal resistance of units. To consider the effect of incorporating it, the following experiment was performed in which the above unlearning/reinforcement algorithm was used first to reduce the number of spurious memories. λ was set to 0.15 and λ_R to 0. Then, the same algorithm was used but this time only the internal resistance R_i of the unit that switched in the last decision was changed. If the wrong unit was switched ON, it's internal resistance was increased and the internal resistance of the unit that should have been switched ON was decreased. R_i was allowed to take values in the range $[0, 15]$ and λ_R was set to 0.4 in this second part. The experiment was conducted on the 1021 words associated with the phonemes experiment (see Table 1.2, 1.3). Table 1.5 summarises the results showing that learning internal resistance achieved 5-10 % additional reduction of spurious memories. The rest of the parameters in this experiment were as in the second part of Table 1.2.

Words	Pos 1	Pos 2	Pos 3	Pos 4	Pos 5	Pos 6
	409	390	394	371	302	143
	366	377	331	266	248	90

Table 1.5

Learning desirable trajectories

The network's behavior can be characterised as a sequence of decisions which transform an initial state to a final state. The task of the network is to reach a desired final state from some initial state. This goal can be accomplished by penalising *responsible* decisions when an *undesirable* final state is reached and *reinforcing responsible* decisions when a *desirable* final state is reached. A network that has learnt desirable trajectories (of sequential decisions) can also be used for sequential decision making. Within this framework, the reinforcement and unlearning algorithms described above are special cases of trajectory learning that unlearn or reinforce only the *last* network decision on a trajectory. Let us examine the general case within our network architecture. We will restrict ourselves to considering only switch ON decisions. Let $S_a(t_i)$ be the set of units active at t_i and x some candidate unit to switch ON at t_{i+1}. Let $w_{xS_{a(t_i)}}$ be the set of weights the unit x gets from units in $S_a(t_i)$ and R_x, the internal resistance of x. Then the decision made at time t_{i+1} is a function of

$$< w_{xS_{a(t_i)}}, R_x >$$

specifically, it is based on summing all it's arguments and thresholding. A particular decision can be penalised by reducing this sum, which entails reducing some of the argument values, that is weights and/or internal resistance. A decision can be reinforced by doing exactly the opposite.

As the number of stored memories increases, many path segments will probably lie in undesirable as well as desired paths. This could lead to the problem that unlearning a segment of an undesirable path will also unlearn the segment from a desired path (and vice versa). An unlearning/reinforcement rule will have to be devised which recognises this problem and attempts to mitigate it within the confines of the network architecture. Extensive experimentation will then have to be done on large training sets to determine it's effectiveness.

Reinforcement learning might provide the general framework within which to do reinforcement and unlearning of the kind we want [5], [6]. A rough sketch of a possible algorithm is as follows. Whenever a unit switches ON, keep track of this switch and all the units that were active when it switched. If the network reaches a spurious state, then increase this unit's internal resistance and decrease the weights from the units that were active at that time.

Comparisons with other models

I would like to make very brief comparisons with other neural network models that have been applied to (or proposed for) CAM. These comparisons are by no means in-depth, I focus only on the bare essentials. Our model exhibits clustering behavior that could be useful for many applications. Certainly at both extremes, one of which is CAM, the other is finding *connected components* (by setting ρ to 0). Most other models don't exhibit both properties, so my comparisons will be limited to CAM behavior. I will assume that the dimensionality of input vectors (or training input) is n, and M is the number of memories stored.

The Hamming net, [3], can be used for content-addressible memories. Whereas the Hamming net minimizes Hamming distance, the model that I've proposed does not (Corr. 3). The Hamming net requires at least $n * M$ connections (ignoring the connections for the winner-take-all effect in $L2$), whereas my model requires $n(n-1)/2$ connections. Hence, my model requires less connections for $M > n$, assuming M is in the range in which spurious memories are still not a problem.

SDM is another model that can be embedded into a 3-layer neural network and can be used for CAM [13]. Retrieval is typically based on a Hamming distance criterion, while in my model it is not. SDM allows storage of pairs of patterns (address, data). SDM, is more general, since it is a true pattern associator whereas my model (and the original Hopfield network) can be used only for CAM. CAM is a special case of pattern association where the two patterns being associated are the same. SDM, however, requires n units in the 1st and 3rd layers and about $20M$ units in the second layer (personal communication with Douglas G. Danforth, RIACS, Nasa). This means that it would require about $20Mn$ binary connections (input-to-hidden layer) and about $20Mn$ real-valued connections (hidden-to-output layer). Using arguments similar to those in the Hamming net comparison, for $M > n$, the storage requirements of SDM are considerably higher than those of my model.

Finally, I would like to compare my model with a one-layer Bayesian network developed in [14], which is similar (in architecture). Firstly, the design goals were quite different and hence the resulting network properties are different.

The parameters of my network (weights, thresholds) were chosen to maximize capacity, that being the highest priority objective, whereas the parameters of the Bayesian network appear to have been chosen to impart it with Bayesian decision making capabilities. It is my belief that, for content-addressable memories, the lack of good storage capacity is a bigger problem in single layer networks than the lack of suitable statistical decision making capabilities.

Secondly, the dynamics of my model and the Bayesian network are quite different. The Bayesian network performs something like a maximum *a posteriori* (MAP) decision as to which of the patterns in the training set is the most likely one to have generated the input [14]. This is quite different from my dynamics, which is a sequence of local binary decisions. My dynamics maps quite well to some applications and provides the network with extremely rapid convergence properties. Since units are switched OFF before others are switched ON, it also allows the network to be used only for noise removal, if that is desired. Finally, it opens up potential use in sequential decision making by allowing the network's state to be output after each decision instead of waiting till the final state (energy minimum) is reached.

Further Work

The first major problem that must be addressed in my model is that of spurious memories. In particular, a theoretically motivated design of unlearning/reinforcement of decisions must be incorporated and it's effectiveness shown through extensive experimentation. If successful, this will also extend the applications of this model to those requiring learning of sequential decisions.

The clustering property of this network has been characterised to some extent. Specific applications must be found that can take advantage of such clustering behavior.

Multi-layer networks based on this model (already developed in [11]) will be considered for applications requiring *hierarchical* or *multi-scale* representation and/or those requiring *selective attention* and *segmentation*. Multi-layered networks are also useful for reducing spurious memories and keeping the network connectivity low.

Conclusion and Acknowledgements

A Hopfield-style network has been proposed that has high capacity, suffers from low interference between memories and exhibits other desirable properties which can be controlled by a single parameter (ρ).

I wish to acknowledge the contribution of Prof. Oleg Jakubowicz, who taught two courses on Neural networks which developed my background in this exciting field. Prof. Oleg Jakubowicz also provided useful comments during the initial development of this model. I also wish to acknowledge the contribution of Prof. Xinhe who proved Thm. 2 and showed a willingness to discuss some of my ideas expressed in graph-theoretic form. Finally, Prof. David Sher has constantly encouraged me to express the network properties in probabilistic terms.

Appendix

In this section, I show some short examples that are quite insightful but don't deserve the status of full-fledged applications. It is hoped that some might lead to full-scale applications. All the following examples were tested and the examples contain editted versions of the actual network training and test sets/output.

Micro-features in Rooms

This example was derived from [15] and is discussed in more detail in [11]. It is repeated here to show certain aspects of network dynamics (during recall) not discussed in [11], specifically, the use of internal resistance, self-weights and clamping. ρ was set to -15, λ to 0.15 and λ_R to 0.6.

Training Input:

```
10 { ceiling walls small  frig stove    oven }
10 { ceiling walls small  tub  toilet   sink     towels }
10 { ceiling walls medium bed  dresser  curtains windows }
10 { ceiling walls large  sofa arm-chair }
5  { ceiling walls small  desk lamp }
2  { ceiling walls large  bed  sofa }
```

```
1  { ceiling walls small  sofa bed      frig }
```

Selected Test Input:

The network output is preceeded by >>.

```
{ bed }
>> { ceiling walls medium bed dresser curtains windows }

-- Without the internal resistance, ceilings and walls would
-- be switched ON before dresser and the network would reach
-- { ceiling walls small sofa bed frig } as the final state.

-- By using (and learning) internal resistance, ceilings and walls
-- have high internal resistance and so dresser switches ON
-- before them leading to the desired final state (bed-room)

{ bed desk }
>> { ceiling walls medium bed dresser curtains windows }
-- bed has higher self-weight than desk and wins the competition.
-- desk switches OFF.

{ bed*1 desk*1 }
>> { ceiling walls small bed desk }
-- Since, the input units are clamped, as shown in Lemma 9,
-- the active units are in the intersection of the memories.
```

Resolving word meanings

This example is derived from [7] and illustrates how our network can resolve word meanings from context. In our example, word order information is not used, so the example in [7] is clearly superior. However, our network does converge provably faster, because the unit states are binary. The value of ρ was set to -0.3, that is in the clustering range. This setting was crucial to the operation of the network for this example.

Training Input

```
{ bucks money } { bucks deer } { shot bullet } { shot waste } { shot fire }
{ shot tired }  { hunt deer }  { hunt bullet } { hunt fire }  { gamble money } { gamble waste }
```

Testing Input

```
{ shot bucks hunt }
>> { fire bullet deer hunt bucks shot }

{ shot bucks gamble }
>> { waste money gamble bucks shot }

-- The above examples show that the context 'hunt' or 'gamble' influence
-- the decisions of the network for the same input.
```

Retrieving closure of binary relations

Consider a binary relation R which is *symmetric*. The task is to store binary sets that satisfy R and retrieve larger sets, all pairs of whose elements satisfy R. The retrieved sets are termed as the *closure* of R. No size units are used in this application.
Case 1: When the relation is also *transitive*. By setting $\rho = 0$, we can retrieve the connected component in G_n that represents the transitive closure of the specified test input.

Relation : Brother.

Training Input

```
{ jim jack } { john jim } { george jeff } { george will } { george he }
```

Testing Input

```
{ jim }
>> { john jack jim }
{ he }
>> { george will jeff he }
```

Case 2: When the relation is not *transitive*. By setting $\rho < -n$, we can retrieve some *clique* in G_n that represents its closure given a specific test input. Note that, given a test input, the closure is not, in general, unique. In other words, there may be more than one clique in G_n that is a superset of the test input.
Relation : Acquaintance.

Training Input

```
{ jim jack } { john jim } { george jeff } { george will } { george he }
{ john jack } { jim rich } { rich jack } { rich john }
```

Testing Input

```
{ jim }
>> { john jack jim rich }
{ he }
>> { george he }
```

References

[1] Hopfield, J.J. (1982). Neural networks and physical systems with emergent computational properties. *Proceedings of the National Academy of Sciences, USA, 79, 2554-2558.*

[2] J.A. Bondy, U.S.R Murty (1976). Graph Theory with Applications, *North-Holland.*

[3] Lippmann, Richard P. An Introduction To Computing with Neural Nets. *IEEE ASSP Magazine, April 1987.*

[4] Andrew G. Barto (1989). Connectionist Learning for Control: An Overview *COINS Technical Report 89-89, Dept Of Computer and Information Science, University Of Massachusetts.*

[5] R.J. Williams, J. Peng (1989). Reinforcement Learning Algorithms as Function Optimizers. *Proceedings of the International Joint Conference on Neural Networks, 1989.*

[6] Andrew G. Barto (1989). Learning and Sequential Decision Making *COINS Technical Report 89-89, Dept Of Computer and Information Science, University Of Massachusetts.*

[7] Waltz, David L., Pollack Jordan B. (1985). Massively Parallel Parsing: A Strongly Interactive Model of Natural Language Interpretation, *Cognitive Science 9, 51-74 (1985).*

[8] Carpenter Gail A. and Stephen Grossberg (1987). A massively parallel architecture for a self-organizing neural pattern recognition machine *Computer Vision, Graphics and Image Processing, 1987, 37, 54-115.*

[9] Barbara Moore (1988). Art 1 and Pattern Clustering *Proceedings of the 1988 Connectionist Models Summer School.*

[10] Tony Bell (1988). Sequential Processing using Attractor Transitions *Proceedings of the 1988 Connectionist Models Summer School.*

[11] Arun Jagota and Oleg Jakubowicz (1989). Knowledge Representation in a Multi-layered Hopfield network *Proceedings of the International Joint Conference on Neural Networks, 1989.*

[12] Arun Jagota and Y.T. Hung (1990). A neural lexicon in a Hopfield-style network, *Proceedings of the International Joint Conference on Neural Networks, 1990.*

[13] Pentti Kanerva (1984). Self-propagating Search: A Unified Theory Of Memory. *Report No CSLI-84-7, Center for the study of Language and Information, Stanford University.*

[14] Lansner A., Ekeberg O. (1989). A one-layer feedback artificial neural network with a Bayesian learning rule. *Tech Report TRITA-NA-P8910, Dept Of Numerical Analysis and Computing Science, Royal Institute of Technology, Stockholm, Sweden.*

[15] McClelland, Rumelhart, and the PDP Research Group. (1986). *Parallel Distributed Processing - Vol 2, Schemata and Sequential Thought.*

[16] T. Kohonen, *Self-Organization and Associative Memory*, Springer-Verlag, Berlin, 1984.