Neural Networks and Optimization Problems

A Case Study: The Minimum Cost Spare Allocation Problem

Michail G. Lagoudakis
The Center for Advanced Computer Studies
University of Southwestern Louisiana
P.O. Box 44330, Lafayette, LA 70504
mgl4822@cacs.usl.edu

Abstract

We are interested in finding near-optimal solutions to hard optimization problems using Hopfield-type neural networks. The methodology is based on a basic property of such networks, that of reducing their 'energy' during evolution, leading to a local or global minimum. The methodology is presented and several different network models usually employed as optimizers (Analog Hopfield net with Simulated Annealing, Boltzmann Machine, Cauchy Machine and a hybrid scheme) are applied on the Minimum Cost Spare Allocation Problem (or equivalently Vertex Cover in bipartite graphs). The experimental results (compared with a conventional exact algorithm) demonstrate the advantages and limitations of the approach in terms of solution quality and computation time.

Keywords: Hopfield Network, Boltzmann Machine, Cauchy Machine, Optimization, Optimal Spare Allocation.

Neural networks were first used to solve optimization problems by Hopfield and Tank around 1985 ([HoTa85], [HoTa87]). Since then the field has grown and two recently published volumes ([ChUn93], [WaTa96]) contain most of the work in the area. Although neural optimization is not a panacea for any optimization problem, there are cases where it applies successfully with respect to conventional search methods. After providing the necessary background, we proceed into such a case study with considerable results, which for that matter consists the main contribution of this work.

1 Hopfield Neural Networks

The *Hopfield Neural Network Model* ([Hopf82], [Hopf84]) consists of a fully connected network of *units* (or *neurons*). The connections between the units are weighted; w_{ij} is the weight of the connection from unit *j* to unit *i*. The model assumes symmetrical weights ($w_{ij} = w_{ji}$) and, in most cases, zero self-coupling terms ($w_{ii} = 0$). A connection with positive weight is *excitatory*, as opposed to an *inhibitory* connection which has negative weight. A unit *i* is characterized by its *output* (or *state*, or *activation*) v_i, the *net input* u_i it receives from the other units and a *bias* θ_i. The *network state* at any time step is given by the output (or state) vector $\vec{v} = (v_1, v_2, \ldots, v_n)$.

Each unit receives input from all the other units and forwards its output to all the other units. The way the output of a unit is updated over time is defined by the network *dynamics*[1]. In general, the output behavior is a function of the *net input*, which is the weighted sum of the inputs and the bias. The McCulloch-Pitts [McPi43] dynamic rule, usually employed in Hopfield networks, has the form:

$$v_i(t+1) = \Phi(u_i(t)) = \Phi\left(\sum_{j=1}^{n} w_{ij} v_j(t) + \theta_i\right) \quad (1)$$

The *activation function* $\Phi(x)$ can take several forms. It can be the unit *step* (or *Heaviside*) function

$$\Theta(x) = \begin{cases} 1 & x > 0 \\ 0 & x \leq 0 \end{cases} \quad (2)$$

in which case we have a discrete Hopfield network with binary states {0, 1}. If the *sign* function

$$Sign(x) = \begin{cases} 1 & x \geq 0 \\ -1 & x < 0 \end{cases} \quad (3)$$

is used, the network is discrete but with values {1, –1}. Finally, a network with continuous-valued units (*analog* Hopfield network), where the values fall in the range [0, 1] or [-1, 1], can be obtained by employing the *sigmoid* (or *logistic*) function

$$g_\beta(x) = \frac{1}{1 + e^{-2\beta x}} \quad (4)$$

or the *hyperbolic tangent* function

$$\tanh(x) = \frac{e^x - e^{-x}}{e^x + e^{-x}} \quad (5)$$

[1] We consider discrete time dynamics only.

respectively. Note that

$$Sign(x) = 2\Theta(x) - 1, \quad \tanh(\beta x) = 2g_\beta(x) - 1 \quad (6)$$

so the {0, 1} (or [0, 1]) model can be easily transformed to {-1, 1} (or [-1, 1]) and vice-versa. In the limit $\beta \to \infty$ the analog units become discrete.

All the networks above are *deterministic* in the sense that the next state is an explicit function of the previous state and the characteristics of the network. For *stochastic* Hopfield networks, the probability of a unit to be in a particular state is drawn from a probability distribution, e.g. Boltzmann or Cauchy. Alternatively, the stochastic network can be viewed as a deterministic one, where the bias of each unit is variable and is drawn from a probability density.

Another distinction comes from the updating policy. It can be *synchronous*, in which case all the units are updated simultaneously at each time step, or *asynchronous*, where either the units are updated in sequence, one unit per time step (*sequential* asynchronous update) or one randomly chosen unit is updated each time step (*random* asynchronous update).

The *energy function* of a Hopfield network is defined over the state space of the network and has the form:

$$E(\vec{v}) = -\frac{1}{2}\sum_{i=1}^{n}\sum_{j=1}^{n} v_i v_j w_{ij} - \sum_{i=1}^{n} v_i \theta_i \qquad (7)$$

The main property of the function above is that it decreases (not necessarily monotonically) during the dynamic evolution of the network. Alternatively, the energy function can be viewed as defining an energy landscape and the dynamics can be thought of as the motion of a marble on the energy surface under the influence of gravity and friction. Consequently, the network performs as a minimizer of its energy function and will be trapped soon, during its evolution, into a local or global minimum of this function. This property is crucial for optimization purposes.

Whenever a unit *i* changes state, the *energy difference* ΔE_i is given by:

$$\Delta E_i = -u_i \Delta v_i = -\left(\sum_{j=1}^{n} w_{ij} v_j + \theta_i\right)\Delta v_i \qquad (8)$$

Consider the {0, 1} network dynamics given by (1) and (2). If the net input u_i is positive, then either Δv_i=0 (1→1) or Δv_i=1 (0→1) and thus $\Delta E_i \leq 0$. Similarly, if the net input is negative, then either Δv_i=0 (0→0) or Δv_i=–1 (1→0) and again $\Delta E_i \leq 0$. The argument is similar for the other models.

The main property of the energy function described above holds only if the connection weights are symmetric and the self-coupling terms are zero or positive. The Hopfield network, as defined previously, fits these requirements and thus its energy function is a, so called, *Lyapunov* function.

2 Mapping Problems on Networks

An *optimization problem* can be defined as an objective function $f(x_1,x_2,x_3, \ldots ,x_n)$ with a set of constraints *C* on the function variables. The goal is to find values for the variables x_i that lead to an optimal value (either minimum or maximum, depending on the problem) of the function *f*, while satisfying all the constraints in *C*. Here we consider only discrete optimization problems, where the x_i's are constrained to discrete values.

Given that a Hopfield network minimizes its energy function, the application of such networks to optimization is straightforward: if the problem can be coded as a function, with the property that the better the solution, the lower the energy level, then a network that encodes this function as its energy function can be used to minimize it (locally or globally) and thus provide an optimal or near-optimal solution. So, the procedure begins with the construction of the energy function and then the network parameters (number of units, weights of connections, biases, update policy or even the dynamics) are adjusted to reflect the problem. The network is initialized to some initial state and is allowed to run until it comes to equilibrium from where a solution can be drawn.

Constructing the appropriate energy function is not, in general, an easy task. The basic constraint is that it must be quadratic in order to meet the form of (7). The most common approach is the *penalty* (or *cost*) *function* approach [LLHZ93]. The energy function is initialized to the objective function of the problem and for each constraint a penalty term is added. Thus, the problem from a constrained optimization form is reduced to an unconstrained minimization problem. If $f(\vec{x})$ is the objective function of the problem and *C* the set of constraints, then the energy function will take the form:

$$E(\vec{x}) = mf(\vec{x}) + \sum_{c \in C} a_c P_c(\vec{x}) \qquad (9)$$

where *m*=+1 for minimization problems or *m*=-1 for maximization problems. The *penalty term* $P_c(\vec{x})$ for

each constraint $c \in C$ has the property that it is zero if and only if the corresponding constraint *c* on $\vec{x}$ is satisfied, otherwise it has a positive value (where possible, it is desirable for this value to increase proportionally to the degree of violation). Finally, the constants α_c define the relative weight concerning the satisfaction of some constraints against some other and/or the relative weight between satisfying all the constraints or minimizing $mf(\vec{x})$. The task of tuning these parameters is generally hard, but does provide a means of customizing the function according to current needs. In the extreme, it can be $\alpha_c = \alpha_c(t)$, so the parameters can be adjusted dynamically during the system's evolution.

The constraints in *C* are in general equality or inequality constraints. An *equality constraint* has the general form

$$c_k : \quad \sum_j \mu_{kj} x_j = \gamma_k \tag{10}$$

where μ_{kj}, γ_k are constants. Since the energy function should be quadratic (at most), an appropriate penalty term for such constraints is the following:

$$P_{c_k}(\vec{x}) = \left(\sum_j \mu_{kj} x_j - \gamma_k \right)^2 \tag{11}$$

The penalty term is zero if and only if the constraint is satisfied, otherwise the penalty is proportional to the degree of violation.

An *inequality constraint* has the general form

$$c_k : \quad \sum_j \mu_{kj} x_j \leq \gamma_k \tag{12}$$

where μ_{kj}, γ_k are constants. The penalty term for such constraints has the following general form:

$$P_{c_k}(\vec{x}) = \Omega \left(\sum_j \mu_{kj} x_j - \gamma_k \right) \tag{13}$$

The function $\Omega(y)$ should penalize only configurations where $y > 0$. One possible choice for $\Omega(y)$ is the unit step function $\Theta(y)$ (eq. 2), with the disadvantage of having the same penalty (=1) independently of the degree of violation. A better alternative, proposed in [OhPS93] is

$$\Omega(y) = y\Theta(y) \tag{14}$$

The penalty term is zero if and only if the constraint is satisfied, otherwise the penalty is proportional (linear) to the degree of violation. The slope of $\Omega(y)$ is implicitly given by the strength of the constraint α_c.

As soon as the energy function is constructed, the network elements (number of units, weights, biases) can be derived. Generally, the number of units depends on the dimension of $\vec{x}$. If $\vec{x}$ is binary, one unit for each component of $\vec{x}$ is needed. For weights and biases, equation (8) is used. The *energy difference* ΔE_i for each unit *i* is calculated (derivatives can be used for this purpose) and is transformed into a form similar to (8). By analogy, the coefficient of v_j will give the weight w_{ij}, whereas the constant terms will give the bias θ_i. In most cases, these values can be verified by common reasoning with respect to the problem under investigation.

One difficulty, at this point, arises from the fact that as problem size grows, the network size (units and connections) may become intractably large, and this is one of the main factors that determine the suitability of a neural optimization scheme for a particular problem. Other factors are evolution time and solution quality.

An important issue is the initialization of the network state. If the network is initialized to a state that corresponds to a possible solution, it will stay there since all the constraints are satisfied, although this solution may not be optimal. Such states correspond to local minima and are not desirable as initial states. Moreover, different initializations may lead to different solutions, with different quality and/or computation time. Usually, the initial state is chosen randomly and the network is allowed to run several times with different initializations. The best of the obtained solutions is returned. The evolution time depends on the initial state and the architecture of the network.

3 The Optimizers

This section presents six different network models used for optimization. The first two are simple and have many disadvantages. Nevertheless, they serve as a smooth introduction to the remaining four sophisticated models whose performance is assessed in the experimentation part of this work.

Discrete Synchronous Hopfield Network

This is the simplest model. The dynamics of the system are given by (1) and either (2) or (3), and all the units are updated in parallel. However, the evolution of the network is not continuous over the edges of the state hypercube leading to oscillation phenomena,

where the network is trapped oscillating between two states. Moreover, the energy is not guaranteed to decrease at each step and since the system may fail to come into equilibrium a solution cannot be always derived. On the other hand, it has the advantage of being fully parallelizable.

Discrete Asynchronous Hopfield Network

This model is similar to the previous one with the difference that the units are updated sequentially. The evolution is continuous over the edges of the state hypercube, since at most one unit changes state at each time step. As a result, the oscillation phenomena are eliminated and the energy function is guaranteed to decrease during evolution. The network will eventually come into equilibrium, from where a solution can be drawn. However, it can be easily trapped in a local minimum depending on the initial state. Finally, the strictly sequential dynamics prevent any parallelization.

Analog Hopfield Network

The dynamics of this model [Hopf84] are given by (1) and either (4) or (5). In the context of optimization, synchronous, asynchronous or continuous updating can be applied. Although the outputs of such networks are continuous they can be used to solve discrete optimization problems. In the equilibrium state the outputs closer to 1 are taken as 1 and the outputs closer to 0 (or -1) are taken as 0. The potential disadvantage of this network is that it may be trapped in local minima inside the hypercube without reaching any corner.

The parameter β of the sigmoid function is usually taken as $\beta = \frac{1}{T}$ where *T* is a virtual temperature. The temperature *T* adjusts the sharpness of the sigmoid (or hyperbolic tangent) function and at the limit $T \to 0$ (absolute temperature) the output becomes discrete. If we start the network at the temperature where we want to measure the outputs, it may take a long time to come to equilibrium. Usually, the *simulated annealing* technique is applied. We start the network at a relatively high temperature and gradually cool it down. This helps the system converge faster avoiding most local minima. Also, as the temperature is lowered the outputs tend to be more discrete. In the case of synchronous dynamics parallelization can be applied.

Boltzmann Machine

The Boltzmann machine [AcHS85] integrates the dynamics of the discrete asynchronous Hopfield model with the simulated annealing technique. At each step *t*, a unit *i* of the network is selected randomly and the energy difference ΔE_i that will be caused by a change of its state is calculated using (8). If ΔE_i is negative (i.e. the energy is decreased) the change is definitely accepted, otherwise it is accepted with probability $P_i^B(t)$ that depends on the quantity $e^{(\Delta E_i / T_B)}$. T_B is a temperature that decreases according to some annealing schedule.

Usually, the *Metropolis criterion* is used as the acceptance criterion, given by

$$P_i^B(t) = \begin{cases} 1 & \text{if } \Delta E_i(t) < 0 \\ \frac{1}{1+e^{\left(\frac{\Delta E_i(t)}{T_B(t)}\right)}} & \text{if } \Delta E_i(t) \geq 0 \end{cases} \quad (15)$$

The *logarithmic schedule* can be used to decrease the temperature:

$$T_B(t) = \frac{T_B(t-1)}{1 + t \log(1+r)} \quad (16)$$

where *r* is a parameter that adjusts the speed of the schedule.

The Boltzmann machine can be effective when used with the appropriate annealling schedule and is able to perform a wide exploration of the problem state space. However, it is strictly sequential and cannot be parallelized. Attempts to parallelize its operation (e.g. group updates [LiPS95]) must cope with a trade-off between solution quality and actual speedup.

Cauchy Machine

The Cauchy machine [ShuH86] extends the discrete synchronous Hopfield model. The behavior of a unit *i* at each time step *t* is given by

$$v_i(t) = \Theta(u_i(t)) \quad (17)$$

During operation, the net input is updated using the following motion equation:

$$\frac{du_i(t)}{dt} = -\frac{\partial E(\vec{v})}{\partial v_i} \quad (18)$$

For simulation purposes, the first-order approximation of the dynamics above is used:

$$\frac{\Delta u_i}{\Delta t} = -\frac{\Delta E_i}{\Delta v_i} \quad \text{and} \quad u_i(t+\Delta t) = u_i(t) + \Delta u_i \quad (19)$$

Note that the net input plays the role of an accumulator, a kind of a memory that stores the cumulative net input of the unit during the network's operation time. This way, the probability of two units to change state

simultaneously is reduced significantly, thus oscillation phenomena are avoided and the system will eventually come to equilibrium. This is strengthened also by the fact that each unit follows *gradient descent* dynamics (see (18), (19)). When the energy function is given by (7), then using (8) and (19), we can derive the motion equation for each unit *i*:

$$\frac{\Delta u_i}{\Delta t} = \sum_{j=1}^{n} w_{ij} v_j + \theta_i \tag{20}$$

$$u_i(t+\Delta t) = u_i(t) + \left(\sum_{j=1}^{n} w_{ij} v_j + \theta_i \right) \Delta t \tag{21}$$

A state vector $\vec{v}$ constitutes an equilibrium state for the network if for all *i* the following condition is satisfied

$$(v_i = 1 \text{ and } \Delta u_i \geq 0) \text{ or } (v_i = 0 \text{ and } \Delta u_i \leq 0) \tag{22}$$

In order to provide the system with stochastic hill-climbing capabilities, the *Distributed Cauchy Machine* [TaSh89] was developed, where Cauchy color noise is added in the updating procedure. The output of unit *i* at each time step *t* is stochastically updated with probability $P_i^C(t)$ which depends on the Cauchy distribution:

$$P_i^C(t) = \begin{cases} s_i(t) & \text{if } v_i(t) = 0 \\ 1 - s_i(t) & \text{if } v_i(t) = 1 \end{cases} \tag{23}$$

$$s_i(t) = P\{v_i(t) = 1\} = \frac{1}{2} + \frac{1}{\pi} \arctan\left(\frac{u_i(t)}{T_C(t)} \right) \tag{24}$$

The virtual temperature $T_C(t)$ is usually given by a *fast annealing* schedule:

$$T_C(t) = \frac{T_C^0}{1+\beta t} \tag{25}$$

where T_C^0 is the initial temperature and β is a real parameter in the range [0, 1] that controls the speed of the schedule. In the extreme case, where T_C = 0, the network becomes deterministic.

The Cauchy machine generally provides solutions of lower quality than the Boltzmann machine, however it has the advantage of being fully parallelizable. Also, since the net input is cumulative, changes at the state of a unit can be done only after many time steps, for the net input must change sign. The lack of this flexibility becomes more obvious when the system has operated for a long time and the net inputs have large absolute values.

Hybrid Scheme

The hybrid update scheme presented here is suggested in [PaLS97] and attempts to combine the advantages of both the Boltzmann and the Cauchy machine. It extends the synchronous discrete Hopfield model and the stochastic update rule is based on a convex combination of the Boltzmann and Cauchy machine update rules. At each time step *t*, the probability of accepting a state change concerning unit *i* is given by

$$P_i^H(t) = \alpha P_i^C(t) + (1-\alpha) P_i^B(t) \tag{26}$$

where α is a control parameter in the range [0, 1]. A large value of α will result in a typical Cauchy machine, whereas a small value will result in a synchronous Boltzmann machine destroying the convergence property. The two temperatures do not have to be equal. A good choice is to update $T_C(t)$ according to some annealing schedule and then take $T_B(t) = \lambda T_C(t)$, where λ is an adjustable parameter.

The above stochastic update rule accepts changes suggested by the energy difference (through $P_i^B(t)$) but such changes should be reflected on the net input $u_i(t)$, otherwise at the next time step the unit will return to the previous value, since this is still suggested by the net input. The following rule[2] ensures that this will not happen:

$$\begin{aligned} &\text{if } (u_i(t+\Delta t) <> u_i(t)) \text{ and } \left(P_i^C(t) < 0.25\right) \text{and} \\ &\left(P_i^B(t) > 0.75\right) \text{ then } (u_i(t+\Delta t) = -u_i(t)) \end{aligned} \tag{27}$$

Experimental results show that solutions produced by this hybrid scheme are similar to solutions produced by the Boltzmann machine but the convergence time is larger, similar to that of the Cauchy machine. However, the hybrid scheme is fully parallelizable, and a parallel implementation will provide solutions similar to those of the Boltzmann machine but in considerably less time.

4 A Case Study

4.1 Minimum Cost Spare Allocation

The Minimum Cost Spare Allocation (MCSA) problem [Kufu87] appears in the context of fault tolerant computing and can be described as follows: Given a 2–dimensional array *M* with dimensions (*R*×*C*), a

[2] The condition $\left(P_i^B(t) > 0.75\right)$ in the rule is not included in the original paper. However, our experimental results suggest that it is necessary for the network to converge.

set of positions in the array *FC*={ (fr_k, fc_k) : 1≤fr_k≤*R*, 1≤fc_k≤*C*, *k*=1, 2, ..., *NFC* } which represent faulty cells (processors, memory cells, VLSI components), a number of *SR* spare rows that can replace any row of the array with a cost *SRC* associated with each one of them and a number of *SC* spare columns that can replace any column of the array with a cost *SCC* associated with each one of them, the objective is to repair all the faulty cells, by assigning spare rows and columns over *M*, with a minimum overall cost. Let r_i, *i*=1, 2, ..., *R*, be a binary variable with a value of 1, if a spare row is assigned to row *i* and 0 otherwise. Similarly, let c_j, *j*=1, 2, ..., *C*, be a binary variable with a value of 1, if a spare column is assigned to column *j* and 0 otherwise. Then the problem can be described as an optimization problem. Minimize

$$f(\vec{r},\vec{c}) = SRC\sum_{i=1}^{R} r_i + SCC\sum_{j=1}^{C} c_j \qquad (28)$$

subject to the following constraints:

$$\sum_{i=1}^{R} r_i \le SR,\ \sum_{j=1}^{C} c_j \le SC,\ \sum_{k=1}^{NFC}(1-r_{fr_k})(1-c_{fc_k}) = 0. \qquad (29)$$

The MCSA problem is equivalent to the Vertex Cover (VC) problem in bipartite graphs, where each row is a node at one side and each column is a node at the other side of the graph. The faulty cells are represented as non–directed links between row-nodes and column-nodes. A cost is assigned to all the row-nodes and similarly to all the column-nodes. The problem now is to find a vertex cover with minimum cost, where the number of the row- and column-nodes in the cover are constrained by some maximum values.

The problem, in either form, is proven to be NP-HARD [Kufu87]. We can drop the constraints on the number of spares so that a solution is always possible (trivial: assign one spare row per row, or one spare column per column). Even in this 'relaxed' form, the problem is still intractable (NP-HARD). Thus, it is difficult (or practically impossible) to find an optimal solution for large instances of the problem. Sometimes, a fast near-optimal solution is acceptable. In the sequel, a network that provides such solutions to the 'relaxed' version of the problem is presented.

4.2 The Energy Function

From section 2 it is straightforward to construct the appropriate energy function for the 'relaxed' form (the constraints on the number of spares have been dropped):

$$E(\vec{r},\vec{c}) = SRC\sum_{i=1}^{R} r_i + SCC\sum_{j=1}^{C} c_j + \alpha_1\sum_{k=1}^{NFC}(1-r_{fr_k})(1-c_{fc_k}) \qquad (30)$$

We have dropped the square in the last term of the function since the quantity under it is always positive. The minimum of this function corresponds to the optimal solution. The last term will be zero and the first two will give the optimal cost. The parameter α_1 gives the relative weight between the objectives of repairing all the cells or achieving the minimum cost. An appropriate value that satisfies both constraints can be found by experimentation.

4.3 The Network

The network will contain (*R*+*C*) units in total, one for each row (row-units) and one for each column (column-units). If a row-unit is firing then a spare row should be assigned to this row of the array. Similarly, firing column-units correspond to spare column assignments.

In order to derive weights and biases we must calculate the energy difference caused by a state change of a unit. Since there are two types of units, we handle the two cases separately. We have

$$\Delta E_{r_i} = \left(SRC + \alpha_1\sum_{k=1}^{NFC}(-1)\delta(i,fr_k)(1-r_{fc_k})\right)\Delta r_i \qquad (31)$$

$$\text{or } \Delta E_{r_i} = -\left(\sum_{k=1}^{NFC}(-\alpha_1)\delta(i,fr_k)r_{fc_k} - SRC + \alpha_1\sum_{k=1}^{NFC}\delta(i,fr_k)\right)\Delta r_i \qquad (32)$$

where $\delta(x,y) = \begin{cases} 1 & x = y \\ 0 & \text{o/w} \end{cases}$

By comparing (32) with (8) it is easy to derive the following for each row-unit *i*, *i*=1,2, ..., *R* and column-unit *j*, *j*=1, 2, ..., *C*:

- $\theta_i = -SRC + \alpha_1(\#\text{ of faulty cells in row } i)$
- $w_{ij} = \begin{cases} -\alpha_1 & \text{if } (r_i, r_j) \text{ is a faulty cell} \\ 0 & \text{otherwise} \end{cases}$

Similarly for the column-units we can derive

$$\Delta E_{c_j} = -\left(\sum_{k=1}^{NFC}(-\alpha_1)\delta(fc_k,j)r_{fr_k} - SCC + \alpha_1\sum_{k=1}^{NFC}\delta(fc_k,j)\right)\Delta rc_j \qquad (33)$$

and for each column-unit *j*, *j*=1, 2, ..., *C* and row-unit *i*, *i*=1,2, ..., *R* we have

- $\theta_j = -SCC + \alpha_1(\#\text{ of faulty cells in column } j)$
- $w_{ji} = \begin{cases} -\alpha_1 & \text{if } (r_i, r_j) \text{ is a faulty cell} \\ 0 & \text{otherwise} \end{cases}$

The weights are symmetric and there are no connections within the row-units or within the column-units but only from row-units to column-units and vice-versa.

It is easy to interpret the values above. The negative weight means that if a unit is firing then it covers all the faulty cells in its row or column, so it will try to prevent the units that could cover the same cells from firing. The positive term in the bias will give firing priority to units with many faulty cells in their row or column, whereas the negative term will prevent units with high cost.

It is easy to see that in the worst case the memory requirements of the network grow like $O(R \times C)$ after taking advantage of the particular structure of the network.

4.4 Experimentation

Simulation experiments (using the C programming language) have been conducted to assess the performance of the four sophisticated optimizers. The number of parameters involved in the MCSA problem is large (namely five), leading to a plethora of different instances. Ten sample instances (with randomly arranged faulty cells) were selected to demonstrate the approach. All the experiments were conducted on a SUN SPARCstation 4 and the results compared with an optimal A* search algorithm for the same problem [Lago97].

The following decisions were made concerning the parameters of the network.

Analog Hopfield with Simulated Annealing The initial temperature was taken equal to 10 and a simple annealing schedule with decreasing rate equal to 0.996 was used, $T(s+1) = rate \times T(s)$.

Boltzmann Machine The initial temperature was taken equal to 5 and the annealing followed the logarithmic schedule (16) with rate equal to 0.000001.

Cauchy Machine The initial temperature was taken equal to 2, following a fast annealing schedule (25) with rate equal to 1. The time step Δt was taken relatively small, equal to 0.005, for close approximation.

Hybrid Scheme The value of α which determines the relative contribution gave best results, when it was taken equal to 0.75[3], i.e. when the contribution was 75% Cauchy Machine and 25% Boltzmann machine. The temperatures were the same as above and the parameter λ was taken equal to 2.5.

For all four models, the termination condition was either 3 consecutive iterations with the same stable state, or a maximum of 2,000 iterations. Each iteration corresponds to a full course of updates (all units updated).

A crucial decision is the value of α_1 (see (30)), that determines the relative strength of the constraints. A small value leads to an 'optimal' cost, but probably with some cells uncovered (!), whereas a large value covers all the cells but not necessarily with the optimal cost. After many experiments, a good value for α_1 was found to be:

$$\begin{aligned} &\alpha_1 = 0.2 \times Min + 0.8 \times Max + \delta(Min, Max) \times Min \\ &Min = \min(SRC, SCC), \quad Max = \max(SRC, SCC) \end{aligned} \tag{34}$$

4.5 Discussion

Table 1 summarizes the performance results. One can figure out easily when the neural network approach is most appropriate. The exact algorithm guarantees the optimal solution, but it is out of the question for large instances. Its time and space requirements grow like $O\left(2^{TFC}\right)$, where *TFC* is the total number of faulty cells, and is highly dependent on the values of the two costs. The size of the array has little impact on it.

On the other hand, the neural network cannot guarantee the optimal solution, but only a near-optimal solution, without any indication of the error. However, the space requirements, which are substantially less, and the time requirements depend solely on the size of the array. The number of faulty cells, as well as the two costs, seem to have no impact on its performance. For these reasons, large instances of the problem can be handled, where the exact algorithm fails.

In case there are instances of a problem that have no solution, the network, as defined here, has no means to indicate this. Neural optimization is most appropriate for optimization problems where at least one solution exists and this is the reason for studying the 'relaxed' version of the MCSA problem.

Parallelization can be easily applied on the parallelizable models. It can decrease the total computation time significantly achieving maximum speedup, and it is one of the main advantages of the approach. Also continuous updating implies the possibility of VLSI implementations of the network.

[3] There seems to be a contradiction here, since in the original paper [PaLS97] its best value was determined as 0.25. However, it may be problem dependent (they study the Minimum Independent Set problem).

Instance	Faulty Cells	Rows	Columns	Row Cost	Column Cost	α_1	Analog Hopfield	Boltzmann Machine	Cauchy Machine	Hybrid Scheme	Exact Algorithm
1	5	5	5	1	9	6.0	**9** (17,700) 1.15 sec	**9** (2,190) 0.32 sec	**9** (3,010) 0.11 sec	**9** (4,120) 1.83 sec	**9** 0.35 sec
2	12	10	10	8	15	13.6	**32** (30,500) 2.67 sec	**32** (1,760) 0.95 sec	**32** (4,200) 1.42 sec	**32** (3,880) 0.66 sec	**32** 0.28 sec
3	50	20	20	3	3	6.0	**48** (70,800) 4.01 sec	**48** (3,040) 1.76 sec	**48** (20,840) 2.01 sec	**48** (22,960) 2.79 sec	**48** 6.00 sec
4	100	20	20	3	3	6.0	**60** (70,800) 5.22 sec	**60** (3,120) 0.27 sec	**72** (25,240) 2.86 sec	**60** (24,800) 3.04 sec	**60** 5.48 sec
5	100	10	30	7	2	6.0	**70** (74,840) 6.46 sec	**70** (3,440) 1.31 sec	**70** (24,640) 1.69 sec	**70** (36,440) 2.11 sec	**58** 0.61 sec
6	30	100	100	3	15	12.6	**78** (354,000) 92.81 sec	**78** (3,800) 1.23 sec	**78** (159,000) 32.26 sec	**78** (208,200) 44.73 sec	**78** 0.07 sec
7	60	100	100	3	3	6.0	**117** (354,000) 97.28 sec	**120** (4,000) 1.70 sec	**117** (218,400) 44.13 sec	**117** (304,200) 62.79 sec	**117** 1200.00 sec
8	60	100	100	1	9	7.4	**45** (400,000) 94.59 sec	**45** (4,600) 1.27 sec	**45** (248,800) 50.23 sec	**45** (400,000) 84.83 sec	**45** 0.29 sec
9	1000	250	250	10	10	20.0	**2520** (735,000) 428.48 sec	**2830** (4,000) 2.27 sec	**2720** (431,500) 219.51 sec	**2490** (322,500) 171.16 sec	out of memory
10	1000	500	500	10	10	20.0	**3960** (1,470,000) 1951.01 sec	**4410** (7,000) 1.64 sec	**3970** (1,327,000) 1484.94 sec	**3920** (1,029,000) 1617.16 sec	out of memory

Table 1 Experimental Results. For each model, the first number is the cost found, the number in parentheses is the number of cycles (individual updates) and the third is the execution time. For the exact algorithm the optimal cost and the execution time are provided.

5 Conclusion

In this paper we studied the application of neural networks on optimization problems and more particularly on the MCSA problem. Although the neural network model was not inspired by optimization, the results presented here and elsewhere prove its success in areas, other than the ones it was intended for. Neural Optimization provides an alternative to conventional methods and, if nothing else, it is another example of how advances in science can be realized by interdisciplinary research and study.

6 Acknowledgments

I would like to thank Dr. Anthony Maida for his invaluable help, the Center for Advanced Computer Studies and the Lilian-Boudouri Foundation in Greece for the financial support.

7 References

[AcHS85] Ackley, D., Hinton, G. & Sejnowski, T. "A Learning Algorithm for Boltzmann Machines", *Cognitive Science*, **9**, 1985, 147–165.

[ChUn93] Cichocki, A. & Unbehauen, R. *Neural Networks for Optimization and Signal Processing*, John Wiley & Sons, 1993.

[Hopf82] Hopfield, J. "Neural Networks and Physical Systems with Emergent Collective Computational Capabilities", *Proceedings of the National Academy of Science, USA*, **79**, 1982, 2254–2558.

[Hopf84] Hopfield, J. "Neurons with Graded Response have Collective Computational Properties like those of the two-state neurons. ", *Proceedings of the National Academy of Science, USA*, **81**, 1984, 3088–3092.

[HoTa85] Hopfield, J.J & Tank, D.W. "Neural Computation of Decisions in Optimization Problems", *Biological Cybernetics*, **52**, 1985, 141–152.

[HoTa87] Hopfield, J.J & Tank, D.W. "Computing with Neural Circuits: A Model", *Science*, **233**, 1987, 625–633.

[KuFu87] Kuo, S. & Fuchs K. "Efficient Spare Allocation for Reconfigurable Arrays", *IEEE Design & Test*, **4**, February 1987, 24–31.

[Lago97] Lagoudakis, M. "An Algorithm for Optimal Spare Allocation: Parallel Implementation for Distributed Memory Machine using TreadMarks™", Submitted to IPPS 98.

[LiPS95] Likas, A., Papageorgiou & Stafylopatis, A." Parallelizable Operation Scheme of the Boltzmann Machine Optimizer Based on Group Updates ", *Neural, Parallel & Scientific Computations*, **3**, 1995, 451–466.

[LLHZ93] Lillo, W., Loh, M., Hui, S., Zak, S. "On Solving Constrained Optimization Problems with Neural Networks: A Penalty Method Approach", *IEEE Transactions on Neural Networks*, **4**, 6, 1993, 931–940.

[McPi43] McCulloch, W.S. & Pitts, W. "A Logical Calculus of Ideas Immanent in Neurous Activity", *Bulletin of Mathematical Biophysics* **5**, 1943, 115–133.

[OhPS93] Ohlsson, M., Peterson, C. & Soderberg, B. "Neural Networks for optimization problems with inequality constraints: the knapsack problem", *Neural Computation*, **5**, 2, 1993, 331–339.

[PaLS97] Papageorgiou, G. Likas, A. & Stafylopatis, A. "A Hybrid Neural Optimization Scheme Based on Parallel Updates", to appear in *International Journal of Computer Mathematics*.

[ShuH86] Shu, H. "Fast Simulated Annealing", in *Neural Networks for Computing*, ed. Denker, J., American Institute of Physics, Snowbird, 1986, 420–425.

[TaSh89] Takefuji, Y. & Shu, H. "Design of Parallel Distributed Cauchy Machines", *International Joint Conference on Neural Networks*, vol. **1**, Washington, D.C., 1989, 529–532.

[WaTa96] Wang, J. & Takefuji, Y. *Neural Computing for Optimization and Combinatorics*, World Scientific Pub Co, 1996.