ECE/CS 559 - Neural Networks Lecture Notes #8: Associative Memory and Hopfield Networks

1 Associative memory

1.1 Introduction

- We learn and recall memories by association: "This is similar to something that I have seen before."
- Even if you see a cat that you have never seen before, you can immediately recognize it as a "cat" by associating it with what you already know about cats (appearance, meowing, etc).
- Computer memory vs. human memory:
  - Computer memory: We feed the address, we get the data in that address.
  - Associative memory: We cannot feed an address (the images that we see do not directly tell us the address that we shall use to understand what that image is/contains). Instead, we utilize the images that we see as stimulants to our neurons, we feed an actual content, and the response we get is an association "This is a cat" or even more "This is a black cat," etc. Meanwhile, we also remember things that we do not see (things that we learned before about cats), e.g. "Cats like milk."
  - Also, another important difference that will be apparent is that: The memory is distributed in associative memory: In a computer memory, a particular data (say a bit) is in one particular physical location. The information in the associative memory is however distributed across all the building blocks (neurons) of the memory.
- Also, mathematically speaking, in the previous sections, we had: A continuous range of inputs $\rightarrow$ A continuous range of outputs. Squared-error function to measure the different between the desired response and the response of the neural network.
- Here, we consider a scenario where we design a neural network that can memorize a certain number of patterns (memories). What we want is that if input stimulus pattern/vector x is mapped to an output vector/response pattern y, then the neighborhood of x should also be mapped to y.
- Example: Even if the cat is missing a tail, we should still recognize it as a cat.
- Example: Suppose the network has learned "Michael Jordan." Then, if the input is Michael Jordan, the output is also Michael Jordan. We may also consider correction of noisy input: Mickhael Joredan $\rightarrow$ Michael Jordan. Or, one should even be able to recover "Michael Jordan" from just "Jordan."
- We may not be able to store however an arbitrarily large number of patterns/names just as in our ordinary memories, or they may be confusions/conflicts between the patterns that we wish to store. Jordan the county, or Michael Jordan the basketball player?
- So, to summarize, the associative memory has the following properties:
  - Stimulus pattern (e.g. face of the person) $\rightarrow$ Response pattern (e.g. the name of the person + other information about him/her).
  - Resistance to noise.

– Distributed nature of both storage and recovery.

– Errors/confusions during recovery due to "closeness" of stored patterns.

1.2 Mathematical model

Say we have m input neurons and n output neurons. The weight going from input neuron j to output neuron i is w_{ij} as usual. Also, suppose for now identity activation functions, so that the input output relationships are

$$y_i = \sum_{j=1}^{m} w_{ij} x_j,\ i = 1, \ldots, n.$$

Or as usual, $\vec{y} = W\vec{x}$, where $\vec{y}$ is the column vector of y_is, $\vec{x}$ is the column vector of x_is, and W is the $n \times m$ matrix with w_{ij} in the ith row, jth column. In general, we want to associate input vectors $\vec{x}_1, \ldots, \vec{x}_q \in \mathbb{R}^m$ with the output vectors $\vec{y}_1, \ldots, \vec{y}_q$. There are three cases:

- Heteroassociative memory: The general case; we want $\vec{x}_i \mapsto \vec{y}_i$, $i = 1, \ldots, q$. Also, if $\widetilde{x}$ is close to $\vec{x}_i$ for some $i \in \{1, \ldots, q\}$ (e.g. in the Euclidean sense), then we want $\widetilde{x} \mapsto \vec{y}_i$.
- Autoassociative memory: Special case where $m = n$ and $\vec{y}_i = \vec{x}_i$, $i = 1, \ldots, q$. Useful for error correction applications.
- Any pattern recognition problem can be considered in the context of heteroassociative memory. Output space dimension is 1, consisting of the indices of the patterns we wish to memorize.

Now let Y denote the $n \times q$ matrix of $\vec{y}_i$s, and X denote the $m \times q$ matrix of x_is. Then, we want $Y = WX$, or in the autoassociative case, we want $X = WX$. At least in the case where $m = n$, there is a solution to our problem when X is invertible, then the weights we want are given by $W = YX^{-1}$.

Now this network will give us what we want at least for the case where we choose the inputs as the patterns we wish to memorize. But it has no mechanism to correct errors if the input is even close a memorized pattern. One way to make the network recover a memorized pattern is to use feedback. Let us for example, connect all the outputs back to the input, and consider a "synchronous update" scenario. That i is to say, we begin with an initial input $\vec{x}^{(1)}$ at discrete time instance 1. Calculate output $\vec{y}^{(1)} = W\vec{x}^{(1)}$, which becomes the input $\vec{x}^{(2)} = \vec{y}^{(1)}$ for discrete time instance 2 and so on.

Then, we feed an input pattern, wait for convergence of the synchronous update scenario, and look at the response. The network converges to the outputs $\lim_{t\to\infty} W^t\vec{x}^{(0)}$ if it exists. The limits can be determined by looking at the Jordan decomposition $W = PJP^{-1}$ of W, where J is a block diagonal matrix, with each block depending on the eigenvalues of W, P is an invertible matrix whose columns are generalized eigenvectors of W. For example:

- If all eigenvalues are magnitude less than 1, then the matrix converges to 0.
- There may be oscillations: E.g. for the 90 degree rotation matrix $W = [0\ 1\ ;\ -1\ 0]$, we have cycles of length 4: $(2,1) \to (1,-2) \to (-2,-1) \to (-1,2) \to (2,1) \to \cdots$.
- An interesting case is when the eigenvalues are all real distinct, with ordering, say $\lambda_1 > \lambda_2 > \cdots$ with $\lambda_1 > 1$. Then, any input vector converges to the direction of the eigenvector corresponding to the largest eigenvalue λ_1. I.e., we have $W^t = PJ^tP^{-1} = P\text{diag}(\lambda_1^t, \cdots, \lambda_n^t)P^{-1}$, where the columns $v_1, \ldots, v_n$ of P are the eigenvectors of W and form a basis for $\mathbb{R}^n$. Then, $x^{(1)}$ can be written as $\sum_{i=1}^n \alpha_i v_i$. So, $W^t x^{(1)} = \sum_{i=1}^n \alpha_i \lambda_i^t v_i \simeq \alpha_1 \lambda_1^t v_i$ for large t provided that $\alpha_1 \neq 0$. Hence, the direction of the output vector converges to the direction of the largest eigenvector of W. The direction v_1 can be considered to be an "attractor" for all initial vectors with non-zero first component.

Regarding the last example, in fact, we can normalize the network output to have unit norm before feeding back, resulting in what is called a Power iteration, or von Mises iteration (for estimation of the largest eigenvalue). In any case, the largest eigenvector (or its direction) acts as an "attractor" of the iterations. We shall design our network in such a way that the attractors are precisely the response patterns that we wish

to store. The last example is not very useful in the sense that we can have only one attractor in the system (we can only store one vector). The way to store more vectors is to introduce non-linearity to the system.

We will work with the following non-linearity. Suppose that the activator is $\phi(x) = 1$ if $x \geq 0$ and $\phi(x) = -1$ if $x < 0$. Hence, we now have

$$y_i = \phi\left(\sum_{j=1}^{m} w_{ij}x_j\right), \; i = 1, \ldots, n.$$

Or as usual, $\vec{y} = \phi(W\vec{x})$. As to how to create memories, we will use Hebbian learning as follows. Assume for a little while that we have no feedback.

1.3 Hebbian Learning

We consider an m-input n-output neural network with the $\phi = \text{sgn}$ activation function. Recall that if neuron j is connected to neuron i with weight w_{ij}, the update equation with Hebbian learning is $\Delta w_{ij} = \eta x_j y_i$. The weight matrix is set to $W = 0$ before the learning is started. Set input to $\vec{x}_1$ and output to $\vec{y}_1$ (the first patterns to be memorized) to get $W = \eta\vec{y}_1\vec{x}_1^T$. Set $\eta = 1$, and do this for all patterns to be memorized to get

$$W = \sum_{i=1}^{q} \vec{y}_i\vec{x}_i^T \in \mathbb{R}^{n\times m}$$

For only one pattern, we obtain $W = \vec{y}_1\vec{x}_1^T$. Now, let us test the network by providing it with an input of $\vec{x}_1$. We have $W\vec{x}_1 = \vec{y}_1\vec{x}_1^T\vec{x}_1 = \|\vec{x}_1\|^2\vec{y}_1$, and if $\vec{x}_1 \neq 0$, after the activation functions, we obtain the output $\phi(\|\vec{x}_1\|^2\vec{y}_1) = \phi(\vec{y}_1) = \vec{y}_1$, provided that the components of $\vec{y}_1$ are $\{-1, +1\}$. So, we are able to successfully associate the input pattern $\vec{x}_1$ to output pattern $\vec{y}_1$. In general, for many patterns, let's test by putting some input pattern $\vec{x}_i$, we obtain

$$W\vec{x}_i = \sum_{i=1}^{q} \vec{y}_j\vec{x}_j^T\vec{x}_i = \vec{y}_i\|\vec{x}_i\|^2 + \sum_{i=1, i\neq j}^{q} \vec{y}_j\vec{x}_j^T\vec{x}_i$$

The summation is called the cross-talk/interference (up to possible normalizing constants). It is equal to zero if all the input patterns are orthogonal, in which case perfect reconstruction is possible as in the single-pattern memory case.

Note that in general, we need

$$\vec{y}_i = \phi(W\vec{x}_i) = \phi\left(\vec{y}_i + \sum_{i=1, i\neq j}^{q} \vec{y}_j \frac{\vec{x}_j^T\vec{x}_i}{\|\vec{x}_i\|^2}\right)$$

.

This will be equal to $\vec{y}_i$ if the components of summation have absolute value less than 1. For that $\vec{x}_j^T\vec{x}_i$ must be small whenever $i \neq j$.

Can we estimate the number of patterns we can store so that later we can recall them with high probability? Note that the crosstalk for bit ℓ is

$$\frac{1}{m}\sum_{i=1, i\neq j}^{q} y_{j\ell}(\vec{x}_j^T\vec{x}_i).$$

Suppose now the components of y_i and x_i are iid Bernoulli($\frac{1}{2}$) random variables on $\{-1. + 1\}$. Then the crosstalk is the sum of $(q-1)m$ Bernoulli random variables. The expected value of the crosstalk is zero. For large qm, we can approximate the crosstalk via a normal distribution with variance $\frac{q}{m}$. If we set the one bit failure probability to 0.01, we obtain $q = 0.18m$ roughly. Hence, the network can store approximately $0.18m$ memory patterns reliably. Other recovery criteria, of course, will result in different bounds on the number of patterns that can be stored.

1.4 Recalling patterns

Let $W = \sum_{i=1}^{q} \vec{y}_i \vec{x}_i^T$, and for an initial vector $\mathbf{x} \in \mathbb{R}^{m\times 1}$, consider the iterates $\phi(\mathbf{Wx}), \phi(\phi(\mathbf{Wx}))$ and so on. We expect this sequence to converge to one of the stored memory patterns $y_1, \ldots, y_q$ (or one of the negations of the stored patterns). In general, the sequence may not converge (oscillations), or it may converge to what is called a spurious state. These are the "unwanted" memory patterns that we inadvertently create when designing $\mathbf{W}$.

2 Hopfield Networks

I work with the same activator as before $\phi(x) = 1$ if $x \geq 0$ and $\phi(x) = -1$ if $x < 0$. I have $\vec{y} = \phi(W\vec{x} + \vec{\theta})$, where y is $n \times 1$, W is $n \times n$, with w_{ij} representing the weight going from the output of neuron j into the input of neuron i, $\vec{\theta}$ is the bias at the neurons. Suppose that W is symmetric with non-negative diagonals.

Let me define the energy function

$$\begin{aligned} E &= -(x^T W x + 2x^T \theta) \\ &= -\sum_{i=1}^{n}\sum_{j=1}^{n} w_{ij}x_i x_j - 2\sum_{i=1}^{n} x_i\theta_i \end{aligned}$$

In the asynchronous update scenario, we update the neurons one by one. Here, we consider a specific update scenario where we update the neurons in the order neuron 1, neuron 2, so on up to neuron n at pass 1, neuron 1, neuron 2, and so on up to neuron n at pass 2, and so on... Then, **each update step results in a non-increasing energy, and the iterations eventually converge to a stable state.**

To see this, say, we updated the output of neuron i, the state x_i has transitioned to state x_i'. First let me write the energy function as

$$E(\vec{x}) = -\left(w_{ii}x_i^2 + x_i \sum_{j=1,\, j\neq i}^{n} w_{ij}x_j + x_i \sum_{j=1,\, j\neq i}^{n} w_{ji}x_j + 2x_i\theta_i \right) + (x_i \text{ independent terms})$$

so that

$$E(\vec{x}') - E(\vec{x}) = -\left(w_{ii}((x_i')^2 - x_i^2) + (x_i' - x_i) \sum_{j=1,\, j\neq i}^{n} w_{ij}x_j + (x_i' - x_i) \sum_{j=1,\, j\neq i}^{n} w_{ji}x_j + 2\theta_i(x_i' - x_i) \right)$$

or

$$\begin{aligned} E(\vec{x}') - E(\vec{x}) &= -\left(w_{ii}((x_i')^2 - x_i^2) - 2w_{ii}x_i(x_i' - x_i) + (x_i' - x_i)\sum_{j=1}^{n} w_{ij}x_j + (x_i' - x_i)\sum_{j=1}^{n} w_{ji}x_j + \theta_i(x_i' - x_i) \right) \\ &= -\left(w_{ii}(x_i' - x_i)^2 + (x_i' - x_i)\sum_{j=1}^{n} w_{ij}x_j + (x_i' - x_i)\sum_{j=1}^{n} w_{ji}x_j + 2\theta_i(x_i' - x_i) \right) \end{aligned}$$

Further, when W is symmetric, we get

$$E(\vec{x}') - E(\vec{x}) = -w_{ii}(x_i' - x_i)^2 - 2(x_i' - x_i)\left(\sum_{j=1}^{n} w_{ij}x_j + \theta_i \right)$$

Now obviously if $x_i' = x_i$, then the energy difference is 0 or ≤ 0. Otherwise, the state has changed. Call the term in the final parenthesis as v_i so that

$$E(\vec{x}') - E(\vec{x}) = -w_{ii}(x_i' - x_i)^2 - 2(x_i' - x_i)v_i$$

If $v_i \geq 0$, we have a transition from $x_i = -1$ to $x_i' = 1$. This makes $2(x_i' - x_i)v_i = 4v_i \geq 0$. By the assumption of $w_{ii} \geq 0$, we have $w_{ii}(x_i' - x_i)^2 = 4w_{ii}^2 \geq 0$, as a result the energy difference is $E(\vec{x}') - E(\vec{x}) \leq 0$. If $v_i < 0$, we have a transition from $x_i = 1$ to $x_i' = -1$, and similarly we have energy $E(\vec{x}') - E(\vec{x}) < 0$ (note the strict inequality).

On the other hand, for any $\vec{x}$, we have

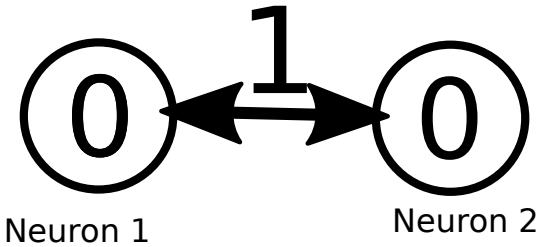
$$\begin{aligned} E(\vec{x}) &= -\left(\sum_{i=1}^{n}\sum_{j=1}^{n} w_{ij}x_i x_j + 2\sum_{i=1}^{n} x_i\theta_i\right) \\ &\geq -\left(\sum_{i=1}^{n}\sum_{j=1}^{n} w_{ij} + 2\sum_{i=1}^{n} |\theta_i|\right) \end{aligned}$$

so that the energy is always bounded from below.

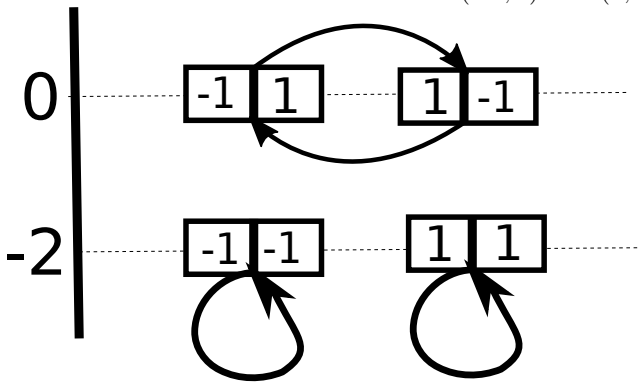
Consider updating neurons one by one, neuron 1, neuron 2, so on up to neuron n at pass 1, neuron 1, neuron 2, and so on up to neuron n at pass 2, and so on... Let k be the number of times that the energy is strictly decreased. Then k should be finite as there are at most a finite number of energy states $\leq 2^n$ each with a finite energy level. Let t_k, $k = 1, \ldots, K$ be the times where the energy is strictly decreased, where K is now for sure finite. The network then converges at time $\max_{k=1,\ldots,K} t_k$ to some energy level and stays there.

Now do the states itself converge? Note that after the energy is converged, the only possible state transition (that does not change the energy is) $-1 \to 1$. After the energy has converged, It will not take more than n^2 steps for all such transitions to occur. Hence, the states also converge.

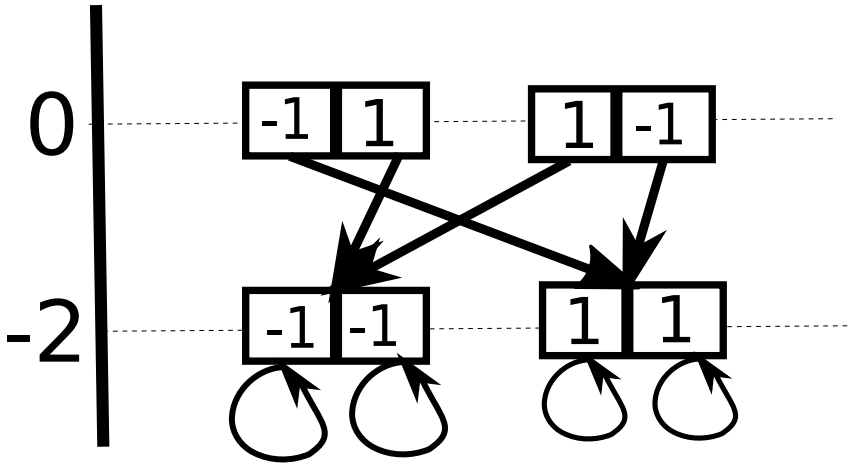
In the synchronous case, we first calculate the new states for all neurons without changing any of the states. We then update the states of all neurons at once. In this case, the system is not guaranteed to converge to a stable state and in fact, there may be oscillations. For example, consider the following two-neuron network. The corresponding weight matrix is $W = \begin{bmatrix} 0 & 1 \\ 1 & 0 \end{bmatrix}$. The numbers "0" within the circles represent that each neuron has 0 bias.



With a synchronous update rule, one then gets the following state transition diagram. The energy levels for each state is also indicated. Note that $(-1, 1)$ and $(1, -1)$ are oscillatory states.



On the other hand, with an asynchronous update rule, we get the following state transition diagram:



All state transitions are clear. For example, starting from state $(-1, 1)$, if we update the first neuron, we follow the arrow that originates below -1 to find out that the next state for the first neuron is 1, so that the entire state of the network transitions to $(1, 1)$. States $(-1, -1)$ and $(1, 1)$ are stable states, as they can only transition to themselves. States $(-1, 1)$ and $(1, -1)$ are referred to as "urstates" or "Garden of Eden" states as they can only appear as the first initial state of the network (they have no ancestral states including themselves). Of course, given different weights/biases, there may be states that are neither urstates nor stable states.

2.1 Applications

2.1.1 Memory and restoration of patterns

Given memory patterns $\mathbf{x}_1, \ldots, \mathbf{x}_q \in \mathbb{R}^{m \times 1}$ we wish to store, we design our weight matrix as $\mathbf{W} = \sum_{i=1}^{q} \mathbf{x}_i \mathbf{x}_i^T$. Consider now a Hopfield network with weights $\mathbf{W}$ and 0 biases. To correct/restore an input pattern $\mathbf{z}$, we consider $\mathbf{z}$ as the initial state of the network, use the asynchronous update rule until we reach a stable state. The stable state will be the corrected/restored version of $\mathbf{z}$.

2.1.2 The Min-Cut Problem

Hopfield networks can be used to provide a locally optimal solution to the min-cut problem (MC) of graph theory.

Let G be an undirected weighted graph of n nodes with $n \times n$ weight matrix W and no self-loops, i.e. W_{ij} represents the weight between Node i and Node j. The undirected nature of the graph implies that W is symmetric. Since the graph has no self-loops, the diagonal elements of W are zero.

We assign a binary state $x_i \in \{-1, +1\}$ to each Node i of the graph. A cut can then be characterized by the variables x_i, $i = 1, \ldots, n$. The set of nodes on one side of the cut would be characterized by $x_i = 1$, and the set of nodes on the other side of the cut would be characterized by $x_i = -1$. The size of the cut is then the sum of weights that connect a -1 node to a $+1$ node. In other words, let W^{+-} denote the sum of the weights of edges one of whose end points have state $+1$ and the other end point has state -1. The size of the cut is W^{+-}. In a similar vein, define the variables W^{++} and W^{--}.

Again, considering $\mathbf{x}$ to be the vectorized version of x_is, we obtain

$$\begin{aligned}
\mathbf{x}^T W \mathbf{x} &= \sum_{i,j} W_{ij} x_i x_j \\
&= 2(W^{++} + W^{--} - W^{+-}) \\
&= 2(W^{++} + W^{--} + W^{+-}) - 2W^{+-} \\
&= \sum_{i,j} W_{ij} - 4(\text{size of the cut})
\end{aligned}$$

Hence, minimizing the size of the cut is equivalent to maximizing $\mathbf{x}^T W \mathbf{x}$, or, minimizing $-\mathbf{x}^T W \mathbf{x}$. But, this is what the Hopfield network does (at least locally-optimally)! So, you may run a Hopfield network with some initial states and weight matrix W and no biases to find a locally-optimal solution to the min-cut problem (Note that there are algorithms such as the Stoer-Wagner algorithm that can solve min-cut exactly for graphs with non-negative weights)... In this context, one can also interpret the Hopfield iterations as shown in the following figure:

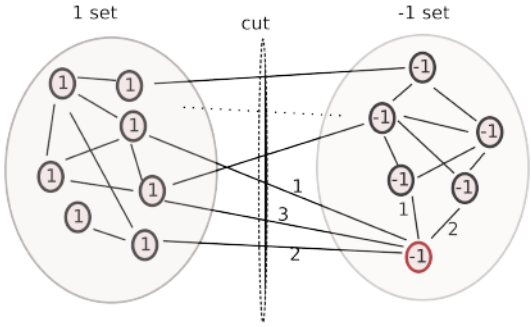


FIGURE 2: A small example

For example, here, for the node circled in red, the outside attraction is higher than the inner attraction (out : 6, in : 3). So the node will change state ($-1 \to 1$) and the size of the cut will decrease (-3).

Image borrowed from Prof. Thierry's notes http://perso.ens-lyon.fr/eric.thierry/Graphes2010/alice-julien-laferriere.pdf)

2.1.3 A toy example

In preparation for the next two applications, we consider the following toy example. We have n boxes and have a number of balls. We would like to find a configuration where we have exactly one ball in one of the boxes and no balls in the other boxes.

There are obviously n solutions. The first is to put the ball in the first box, and then no balls Boxes 2 to n. The other solutions are similar.

For the fun of it, we will design a Hopfield network that finds one of these solutions. Different from the previous sections, we will use here the step activation function $u(\cdot)$. The convergence theorem remains exactly the same.

Suppose the states of the boxes are given by $x_i \in \{0, 1\}$, $i = 1, \ldots, n$. Our design goal is then equivalent to minimizing $(\sum_{i=1}^n x_i - 1)^2$. If we expand this, we get

$$\begin{aligned}\left(\sum_{i=1}^n x_i - 1\right)^2 &= \left(\sum_{i=1}^n x_i\right)^2 - 2\sum_{i=1}^n x_i + 1 \\ &= \sum_{i=1}^n \sum_{j=1}^n x_i x_j - 2\sum_{i=1}^n x_i + 1 \\ &= -\left(-\sum_{i=1}^n \sum_{j=1}^n x_i x_j + 2\sum_{i=1}^n x_i\right) + 1\end{aligned}$$

Now consider choosing $W = -(n \times n$ all-1 matrix), and $b = [2 \cdots 2]^T$. The above derivation shows that minimizing $(\sum_{i=1}^n x_i - 1)^2$ is equivalent to minimizing $-(\mathbf{x}^T W \mathbf{x} + 2b^T \mathbf{x})$, which is the energy function of the Hopfield network. Hence, we just consider an n-neuron Hopfield network with weights W and biases n, and that solves the balls and boxes problem we have.

One complication is that for the Hopfield network to converge the weight matrix should have non-negative diagonals, whereas W has negative diagonals. The way to fix this is by adding the terms $x_i^2 - x_i$, which are

equal to 0 when x_i are either 0 or 1 and thus do not change the value of the objective function. In particular, adding $\sum_{i=1}^{n}(x_i^2 - x_i)$ to the final equality above, we obtain a weight matrix with all -1 except with zeros on the diagonal, and a bias vector of all 1s. The corresponding Hopfield network is now guaranteed to converge.

2.1.4 n rooks

Suppose now that we have n rooks and would like to place them on an $n \times n$ chess board such that no two rooks can take one another. Again, the solution is trivial, but for the fun of it, we use a Hopfield network.

Let x_{ij} denote the state whether the (i, j)th cell of the chess board contains a rook or not. For our constraint to be satisfied (no two rooks should take one another), no two rooks should be on the first row (otherwise they can take one another). Equivalently, we need $(\sum_{j=1}^{n} x_{1j} - 1)^2 = 0$. A similar condition should hold for all rows, resulting in the condition $\sum_{i=1}^{n}(\sum_{j=1}^{n} x_{ij} - 1)^2 = 0$. Running the same constraints for the columns, the configurations we need are the minimizers of $\sum_{i=1}^{n}(\sum_{j=1}^{n} x_{ij} - 1)^2 + \sum_{i=1}^{n}(\sum_{j=1}^{n} x_{ji} - 1)^2$. We can similarly expand this expression to design a Hopfield network to solve our problem. The network in this case will have n^2 neurons.

2.1.5 The traveling salesman problem

We can also approximately solve the traveling salesman problem using Hopfield networks and the n rook idea in the previous subsection.

(Images below From Prof. Thierry's notes http://perso.ens-lyon.fr/eric.thierry/Graphes2010/alice-julien-laferriere.pdf):

4.2.1 The travelling salesman

The Travelling Salesman Problem (TSP) is an NP-hard problem in combinatorial optimization. Given a list of cities and their pairwise distances, the task is to find a shortest possible tour that visits each city exactly once. The path has to pass through n cities : S_1, S_2,...S_n. Each city should only be visited once, the salesman has to return to his original point of departure and the length of the path has to be minimal.

The network can be represented as :

	1	2	...	n
S_1				
S_2				
$\vdots$				
S_n				

Where rows are the city and columns are the time. The units are the entry of the matrix. We suppose that the (n+1) column is the first one to keep the cycle. Also only one 1 should appears for each row and column because the salesman only visit a city once and cannot visit two city during the same time step.

The distance between the city S_i and the city S_j is d_{ij}. To find the shortest path, the network will minimize the function :

$$E_{tsp} = \underbrace{\frac{1}{2}\sum_{i,j,k}^{n} d_{ij}x_{ik}x_{j,k+1}}_{L} + \underbrace{\frac{\gamma}{2}(\sum_{j=1}^{n}(\sum_{i=1}^{n} x_{ij} - 1))^2 + \sum_{i=1}^{n}(\sum_{j=1}^{n} x_{ij} - 1)^2))}_{Condition} \tag{7}$$

The first part of the equation, L, is the total length of the trip. $x_{i,k} = 1$ means that the city S_i was visited at time k(otherwise $x_{i,k} = 0$), so if $x_{ik} = 1$ and $x_{i,k+1} = 1$ (S_i visited at time k and S_j visited at time k+1), the distance will be added to the length L.

The second part represent the condition : one time in each city. $\sum_{i=1}^{n} x_{ij} - 1)^2$ will be minimal (=0) if for every i (here city), the sum over j (here time) is 1, meaning that a city can only be visited once. $\sum_{j=1}^{n}(\sum_{i=1}^{n} x_{ij} - 1)^2$ will be minimal if for every time (k=1...n), the salesman only visit one city.

γ is a parameter. Unfortunately, there is no other choice than using ≪ trial and error ≫ to determine the value we want to set. If γ is small, the condition will not be respected. If γ is really big, the system will be constraint a lot and L will be hidden (so the minimization of E_{tsp} will not consider the distance between cities).

The weights between units becomes :$w_{ik,jk+1} = -d_{ij} + t_{ik,jk+1}$ where

$$t_{ik,jk+1} = \begin{cases} -\gamma & \text{if the units belong to the same column or row} \\ 0 & \text{otherwise} \end{cases}$$

and the threshold should be set to $-\frac{\gamma}{2}$.

The network will not be able to find the good path. But still, the solutions are approximation. If the number of cities is important (> 100), there is several minima so we cannot prove if the solution given by the system is a good approximation.