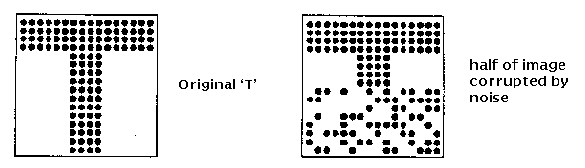
**HOPFIELD NEURAL NETWORKS (HNN)**

* These are special kinds of NN, where the output is different from other neural networks.
* These networks collect and retrieve memory like the human brain.
* Invented by John Hopfield in 1982 and is commonly used for auto-association and optimization tasks.

### The purpose of a Hopfield net is to store one or more patterns and to recall the full patterns based on partial input.

* These networks have a single layer of neurons, which are fully connected recurrent neurons.
* No. of neurons are relative to the size of the input and output, which are the same.
* A HNN is first trained to store various patterns or memories.
* Once trained, even if we give some corrupted input pattern, it is capable of restoring/retrieving the closest pattern.
* Ex : Training a HNN to retrieve alphabets with one neuron per pixel in the image of an alphabet.

****

* If it is trained with ‘original T’, even if we input a distorted image of ‘T’, it is capable of retrieving the closest pattern of ‘original T’.
* Here, a neuron is either on or off. The state of a neuron (on +1 or off 0) will be restored, relying on the input it receives from the other neuron.
* A Hopfield network is at first prepared to store various patterns or memories. Afterward, it is ready to recognize any of the learned patterns by uncovering partial or even some corrupted data about that pattern, i.e., it eventually settles down and restores the closest pattern.
* Thus, similar to the human brain, the Hopfield model has stability in pattern recognition.

Architecture

* A Hopfield network is a single-layered and recurrent network in which the neurons are entirely connected, i.e., each neuron is associated with other neurons.
* No self-connection
* If there are two neurons i and j, then there is a connectivity weight **w**ij lies between them which is symmetric **wij = wji**.
* The output of each neuron should be the input of other neurons but not the input of self.
* With zero self-connectivity, **Wii =0** is given below. Here, the given three neurons having values **i = 1, 2, 3** with values **Xi=±1** have connectivity weight **Wij**.



# Hopfield Networks

|  |
| --- |
| **So here's the way a Hopfield network would work.**  **You map it out so that each pixel is one node in the network. You train it (or just assign the weights) to recognize each of the 26 characters of the alphabet, in both upper and lower case (that's 52 patterns).**  **Now if your scan gives you a pattern like something on the right of the above illustration, you input it to the Hopfield network, and it chugs away for a few iterations, and eventually reproduces the pattern on the left, a perfect "T".**  **Note that this could work with higher-level chunks; for example, it could have an array of pixels to represent the whole word.**  **It could also be used for something more complex like sound or facial images. The problem is, the more complex the things being recalled, the more pixels you need, and as you will see, if you have N pixels, you'll be dealing with N2 weights, so the problem is very computationally expensive (and thus slow).** |

|  |  |
| --- | --- |
| All the nodes in a Hopfield network are both inputs and outputs, and they are fully interconnected.That is, each node is an input to every other node in the network.You can think of the links from each node to itself as being a link with a weight of 0.Here's a picture of a 3-node Hopfield network: | http://web.cs.ucla.edu/~rosen/161/notes/hop3.gif |

|  |  |
| --- | --- |
| http://web.cs.ucla.edu/~rosen/161/notes/bp1.gif | In the previous neural models you've seen, the processing (ignoring training) only goes in one direction:you start from the input nodes, do a sum & threshold of those values to get the outputs of the first layer, and possibly pass those values to a second layer of summing & thresholding, but nothing gets passed back from layer 2 to layer 1 or even passed between the nodes in layer 1: |

|  |
| --- |
| In a Hopfield network, all the nodes are inputs to each other, and they're also outputs. As I stated above, how it works in computation is that you put a distorted pattern onto the nodes of the network, iterate a bunch of times, and eventually it arrives at one of the patterns we trained it to know and stays there. So, what you need to know to make it work are:How to "train" the networkHow to update a node in the networkHow the overall sequencing of node updates is accomplised, andHow can you tell if you're at one of the trained patterns |

## How to "train" a Hopfield network

|  |
| --- |
| In the original article where Hopfield described the net, he provided a formula with which you can calculate the values of the weights without any training.Suppose we wish to store the set of states Vs, s = 1, ..., n. We use the storage prescription: |

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
|  | Wij = | http://web.cs.ucla.edu/~rosen/161/notes/sigma.gif | (2Vsi - 1)(2Vsj - 1) |  | [1] |

|  |
| --- |
| but with Wij = 0. **Note that if you only have one pattern, this equation deteriorates to:** |

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
|  | Wij = | (2Vi - 1)(2Vj - 1) |  | [2] |

|  |
| --- |
| For example, say we have a 5 node Hopfield network and we want it to recognize the pattern (0 1 1 0 1). Since there are 5 nodes, we need a matrix of 5 x 5 weights, where the weights from a node back to itself are 0. The weight matrix will look like this: |

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
|  | 0 | W12 | W13 | W14 | W15 |
|  | W21 | 0 | W23 | W24 | W25 |
|  | W31 | W32 | 0 | W34 | W35 |
|  | W41 | W42 | W43 | 0 | W45 |
|  | W51 | W52 | W53 | W54 | 0 |

|  |
| --- |
| One thing you might notice from equation [2] is that if you switch the i and j indexes, you get the same result. That is,Wij = (2Vi - 1)(2Vj - 1) = (2Vj - 1)(2Vi - 1) = Wji Since the weights are symmetric, we only have to calculate the upper diagonal of weights, and then we can copy each weight to its inverse weight. In this case, V is the vector  (0 1 1 0 1), so V1 = 0, V2 = 1, V3 = 1, V4 = 0, and V5 = 1. Thus the computation of the weights is as follows: |

|  |  |
| --- | --- |
|  | W12 = (2V1 - 1)(2V2 - 1) = (0 - 1)(2 - 1) = (-1)(1) = -1 W13 = (2V1 - 1)(2V3 - 1) = (0 - 1)(2 - 1) = (-1)(1) = -1 W14 = (2V1 - 1)(2V4 - 1) = (0 - 1)(0 - 1) = (-1)(-1) = 1 W15 = (2V1 - 1)(2V5 - 1) = (0 - 1)(2 - 1) = (-1)(1) = -1 W23 = (2V2 - 1)(2V3 - 1) = (2 - 1)(2 - 1) = (1)(1) = 1 W24 = (2V2 - 1)(2V4 - 1) = (2 - 1)(0 - 1) = (1)(-1) = -1 W25 = (2V2 - 1)(2V5 - 1) = (2 - 1)(2 - 1) = (1)(1) = 1 W34 = (2V3 - 1)(2V4 - 1) = (2 - 1)(0 - 1) = (1)(-1) = -1 W35 = (2V3 - 1)(2V5 - 1) = (2 - 1)(2 - 1) = (1)(1) = 1 W45 = (2V4 - 1)(2V5 - 1) = (0 - 1)(2 - 1) = (-1)(1) = -1 |

|  |
| --- |
| So now our weight matrix looks like this:W12 = (2V1 - 1)(2V2 - 1) = (0 - 1)(2 - 1) = (-1)(1) = -1W21=(2V2-1)(2V1-1)= |

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
|  | 0 | -1 | -1 | 1 | -1 |
|  | W21 | 0 | 1 | -1 | 1 |
|  | W31 | W32 | 0 | -1 | 1 |
|  | W41 | W42 | W43 | 0 | -1 |
|  | W51 | W52 | W53 | W54 | 0 |

|  |
| --- |
| By reflecting about the diagonal, we get the full weight matrix: |

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
|  | 0 | -1 | -1 | 1 | -1 |
|  | -1 | 0 | 1 | -1 | 1 |
|  | -1 | 1 | 0 | -1 | 1 |
|  | 1 | -1 | -1 | 0 | -1 |
|  | -1 | 1 | 1 | -1 | 0 |

|  |
| --- |
| For completeness' sake, you'll remember that the original formula was set up to allow you to have n patterns. So let's consider the case where we want our 5 node Hopfield net to store both the pattern V1 = (0 1 1 0 1) and another pattern V2 = (1 0 1 0 1). One way you could go about calculating the weights is on a weight by weight basis. For example, W12 could be calculated as: |

|  |  |  |  |
| --- | --- | --- | --- |
|  | W12 = | http://web.cs.ucla.edu/~rosen/161/notes/sigma.gif | (2Vs1 - 1)(2Vs2 - 1) |
|  | = | (2V11 - 1)(2V12 - 1) + (2V21 - 1)(2V22 - 1) | |
|  | = | (2\*0 - 1)(2\*1 - 1) + (2\*1 - 1)(2\*0 - 1) | |
|  | = | (0 - 1)(2 - 1) + (2 - 1)(0 - 1) | |
|  | = | (-1)(1) + (1)(-1) | |
|  | = | -1 + -1 | |
|  | = | -2 | |

|  |
| --- |
| You could go through each weight like this and calculate the new weight matrix. To do this, you would calculate the matrix for the first pattern (which we did above), then calculate the value for the second matrix and finally add the two matrices together. Here's the weight matrix for the pattern (1 0 1 0 1): |

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
|  | 0 | -1 | 1 | -1 | 1 |
|  | -1 | 0 | -1 | 1 | -1 |
|  | 1 | -1 | 0 | -1 | 1 |
|  | -1 | 1 | -1 | 0 | -1 |
|  | 1 | -1 | 1 | -1 | 0 |

|  |
| --- |
| Now add this to the previous weight matrix and we get: |

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
|  | 0 | -2 | 0 | 0 | 0 |
|  | -2 | 0 | 0 | 0 | 0 |
|  | 0 | 0 | 0 | -2 | 2 |
|  | 0 | 0 | -2 | 0 | -2 |
|  | 0 | 0 | 2 | -2 | 0 |

## How to update a node in a Hopfield network

## This allows the net to serve as a content addressable memory system, that is to say, the network will converge to a "remembered" state if it is given only part of the state

## The net can be used to recover from a distorted input to the trained state that is most similar to that input. This is called associative memory because it recovers memories on the basis of similarity.

## For example, if we train a Hopfield net with five units so that the state (1, -1, 1, -1, 1) is an energy minimum, and we give the network the state (1, -1, -1, -1, 1) it will converge to (1, -1, 1, -1, 1).

|  |
| --- |
| So now we have a weight matrix for a 5 node Hopfield network that's meant to recognize the patterns (0 1 1 0 1) and (1 0 1 0 1). One thing you might notice is that these patterns only differ by 2 bits. As you might imagine, patterns this close together might be difficult to tell apart. By analogy, you might have trouble discriminating a lower case "c" from "e" or an upper case "O" from "Q" if they were mangled badly enough. Let's start from the pattern (1 1 1 1 1), which only differs from each of these patterns by 2 bits, and see what happens. **Updating a node in a Hopfield network is very much like updating a perceptron. If you are updating node 3 of a Hopfield network, then you can think of that as the perceptron, and the values of all the other nodes as input values, and the weights from those nodes to node 3 as the weights. In other words, first you do a weighted sum of the inputs from the other nodes, then if that value is greater than or equal to 0, you output 1. Otherwise, you output 0. In formula form:** |

|  |  |  |  |
| --- | --- | --- | --- |
|  | Viin = | http://web.cs.ucla.edu/~rosen/161/notes/sigma2.gif | WjiVj |
|  | Vi -> 1 if Viin >= 0 | |  |
|  | else Vi -> 0 | |  |

|  |
| --- |
| So assuming we have the final weight matrix from the previous section, and we start from the state (1 1 1 1 1), for the 3rd node, we have: |

|  |  |  |  |
| --- | --- | --- | --- |
|  | V3in = | http://web.cs.ucla.edu/~rosen/161/notes/sigma3.gif | Wj3Vj |
|  | = | W13V1 + W23V2 + W43V4 + W53V5 | |
|  | = | 0\*1 + 0\*1 + -2\*1 + 2\*1 | |
|  | = | 0 | |
|  | since 0 >= 0, | |  |
|  | V3 = 1 | |  |

|  |
| --- |
| In this case, the value of V3 doesn't change. It's worth noticing that since the weight from node 3 to itself is 0, we could have just calculated the dot product of the 3rd column out of the weight matrix and the current state to calculate the weighted sum: |

|  |  |
| --- | --- |
|  | V3in = (0 0 0 -2 2) \* (1 1 1 1 1) = -2 + 2 = 0 |

## Sequencing of node updates in a Hopfield network

|  |
| --- |
| You might have noticed by now that sequencing the updates of the nodes in a Hopfield network is somewhat tricky. How can you update a neuron if the values of its inputs are changing? Well, there are two approaches. The first is synchonous updating, which means all the nodes get updated at the same time, based on the existing state (i.e. not on the values the nodes are changing to). To update the nodes in this method, you can just multiply the weight matrix by the vector of the current state. **This isn't very realistic in a neural sense, as neurons don't all update at the same rate. They have varying propagation delays, varying firing times, etc., so a more realistic assumption would be to update them in random order. This was the method described by Hopfield, in fact. You randomly select a neuron, and update it. Then you randomly select another neuron and update it. You keep doing this until the system is in a stable state (which we'll talk about later).**  **In practice, people code Hopfield nets in a semi-random order. They update all of the nodes in one step, but within that step they are updated in random order. So it might go 3, 2, 1, 5, 4, 2, 3, 1, 5, 4, etc. This is just to avoid a bad pseudo-random generator from favoring one of the nodes, which could happen if it was purely random: 3, 2, 1, 2, 2, 2, 5, 1, 2, 2, 4, 2, 1, etc.** |

## How to tell when you can stop updating the network

|  |
| --- |
| The main reason to want to cycle through all the nodes each step is that it's the only way you can tell when to stop. Basically, if you go through all the nodes and none of them changes, you can stop. If you're updating them in a fixed sequence (e.g. 1, 2, 3, 4, 5, 1, 2, 3, 4, 5, 1, 2, 3, etc.), this means that if you go through 5 consecutive neurons without changing any of their values, then you're at an attractor so you can stop. |

## Finishing up the example

|  |
| --- |
| given the weight matrix for a 5 node network with (0 1 1 0 1) and (1 0 1 0 1) as attractors, start at the state (1 1 1 1 1) and see where it goes. To keep it simple, I'm going to update the nodes in the fixed order 3, 1, 5, 2, 4, 3, 1, 5, 2, 4, etc. I already did the first update of node 3, and it didn't change, so continuing:update node 3 - did it, no changeupdate node 1 - V1in = (0 -2 0 0 0) . (1 1 1 1 1) = -2 since -2 < 0, V1 = 0 *(it changed)*update node 5 - V5in = (0 0 2 -2 0) . (0 1 1 1 1) = 0 since 0 >= 0, V5 = 1 *(it didn't change)*update node 2 - V2in = (-2 0 0 0 0) . (0 1 1 1 1) = 0 since 0 >= 0, V2 = 1 *(it didn't change)*update node 4 - V4in = (0 0 -2 0 -2) . (0 1 1 1 1) = -4 since -4 < 0, V4 = 0 *(it changed)*update node 3 - V3in = (0 0 0 -2 2) . (0 1 1 0 1) = 2 since 2 >= 0, V3 = 1 *(it didn't change)*update node 1 - V1in = (0 -2 0 0 0) . (0 1 1 0 1) = -2 since -2 < 0, V1 = 0 *(it didn't change)*update node 5 - V5in = (0 0 2 -2 0) . (0 1 1 0 1) = 2 since 2 >= 0, V5 = 1 *(it didn't change)*update node 2 - V2in = (-2 0 0 0 0) . (0 1 1 0 1) = 0 since 0 >= 0, V2 = 1 *(it didn't change)*update node 4 - V4in = (0 0 -2 0 -2) . (0 1 1 0 1) = -4 since -4 < 0, V4 = 0 *(it didn't change)*Now we've updated each node in the net without them changing, so we can stop.Notice where the network ends up: at the attractor (0 1 1 0 1). Now why don't you try to do the same problem, but use this order for updating the nodes: 2, 4, 3, 5, 1, 2, 4, 3, 5, 1. Hopfield network  Input pattern: 11011, 10101 calculate weight matrix  Find the correct pattern of distorted pattern (11110)  Wij=(2Vi-1)(2Vj-1)  Sequence: 5 3 4 1 2  Weight matrix for 11011=>  0 1 -1 1 1  1 0 -1 1 1  -1 -1 0 -1 -1  1 1 -1 0 1  1 1 -1 1 0  Weight matrix for 10101 =>  0 -1 1 -1 1  -1 0 -1 1 -1  1 -1 0 -1 1  -1 1 -1 0 -1  1 -1 1 -1 0  Final weight matrix =>  0 0 0 0 2  0 0 -2 2 0  0 -2 0 -2 0  0 2 -2 0 0  2 0 0 0 0  After calculating weight matrix, update the node values. (11011)  Vi=weight\*input  If Vi>=0 then vi=1  Else vi=0  Distorted input: 11110  Order of updation= 5 3 4 1 2 , 1 2 3 4 5  Update node 5:  [2 0 0 0 0]\*[11110]=2 2>=0 then v5=1 (node5 value is changed)  Update node 3:  [0 -2 0 -2 0]\*[11111]= -4 -4<0 v3=0 (node3 value is changed)  Update node 4:  [0 2 -2 0 0]\*[11011]=2 2>0 v4=1 (no change in node 4)  Update node 1:  [0 0 0 0 2]\*[11011]=2 2>0 v1=1 (no change in node 1)  Update node 2:  [0 0 -2 2 0]\*[11011]= 2 2>0 v2=1 ( no change in node 2)  Second iteration: sequence is 5 3 4 1 2  Update node 5:  [2 0 0 0 0]\*[11011]= 2 2>=0, v5=1 (no change)  Update node 3:  [0 -2 0 -2 0]\*[11011]= -4 -4<0, v3=0 (no change)  Update node 4:  [0 2 -2 0 0]\*[11011]=2 2>0 v4=1 (no change in node 4)  Update node 1:  [0 0 0 0 2]\*[11011]=2 2>0 v1=1 (no change in node 1)  Update node 2:  [0 0 -2 2 0]\*[11011]= 2 2>0 v2=1 ( no change in node 2)  Corrected pattern is [11011] |