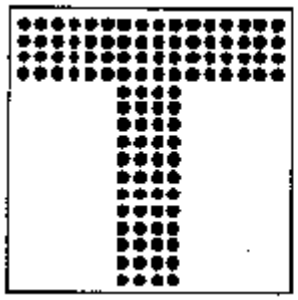
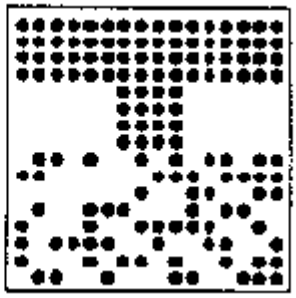
Hopfield Networks (Notes)

(notes are generously borrowed Prof. Bruce Rosen who borrowed some illustrations from Kevin Gurney's notes, and some descriptions from "Neural networks and physical systems with emergent collective computational abilities" by John Hopfield)

The purpose of a Hopfield net is to store 1 or more patterns and to recall the full patterns based on partial input. For example, consider the problem of optical character recognition. The task is to scan an input text and extract the characters out and put them in a text file in ASCII form. Okay, so what happens if you spilled coffee on the text that you want to scan? Now some of the characters are not quite as well defined, though they're mostly closer to the original characters than any other character:



Original 'T'

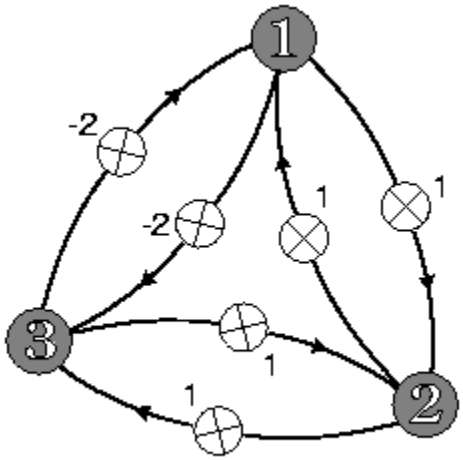


half of image corrupted by noise

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".

Structure of a Hopfield Network

A Hopfield network is a recurrent network. 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:



3 node Hopfield net

In a Hopfield network, all the nodes are inputs to each other, and they're also outputs. 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 network
- How to update a node in the network
- How the overall sequencing of node updates is accomplished, and
- How can you tell if you're at one of the trained patterns

How to "train" a Hopfield network

There are ways of training 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. This is called **one-shot training.**

Suppose we wish to store the set of states V^s, s = 1, ..., n. We use the storage prescription:

$$W_{ij} = \sum_{s=1}^{n} (2V_s^i - 1)(2V_s^j - 1)$$

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	W_{12}	W_{13}	W_{14}	W_{15}
W_{21}	0	W_{23}	W_{24}	W_{25}
W_{31}	W_{32}	0	W_{34}	W_{35}
W_{41}	W_{42}	W_{43}	0	W_{45}
W_{51}	W_{52}	W_{53}	W_{54}	0

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 $V_1 = 0$, $V_2 = 1$, $V_3 = 1$, $V_4 = 0$, and $V_5 = 1$. Thus the computation of the weights is as follows:

$W_{12} = (2V_1 - 1)(2V_2 - 1) = (0 - 1)(2 - 1) = (-1)(1) = -1$
$W_{13} = (2V_1 - 1)(2V_3 - 1) = (0 - 1)(2 - 1) = (-1)(1) = -1$
$W_{14} = (2V_1 - 1)(2V_4 - 1) = (0 - 1)(0 - 1) = (-1)(-1) = 1$

$W_{15} = (2V_1 - 1)(2V_5 - 1) = (0 - 1)(2 - 1) = (-1)(1) = -1$
$W_{23} = (2V_2 - 1)(2V_3 - 1) = (2 - 1)(2 - 1) = (1)(1) = 1$
$W_{24} = (2V_2 - 1)(2V_4 - 1) = (2 - 1)(0 - 1) = (1)(-1) = -1$
$W_{25} = (2V_2 - 1)(2V_5 - 1) = (2 - 1)(2 - 1) = (1)(1) = 1$
$W_{34} = (2V_3 - 1)(2V_4 - 1) = (2 - 1)(0 - 1) = (1)(-1) = -1$
$W_{35} = (2V_3 - 1)(2V_5 - 1) = (2 - 1)(2 - 1) = (1)(1) = 1$
$W_{45} = (2V_4 - 1)(2V_5 - 1) = (0 - 1)(2 - 1) = (-1)(1) = -1$

Note this this is very similar to Hebbian Learning – weights are strengthened (changed to 1) when nodes fire synchronously (e.g., W_{14} and W_{23}, W_{25}, and W_{35}) – otherwise they are weakened (changed to -1).

So now our weight matrix looks like this:

0	-1	-1	1	-1
W_{21}	0	1	-1	1
W_{31}	W_{32}	0	-1	1
W_{41}	W_{42}	W_{43}	0	-1
W_{51}	W_{52}	W_{53}	W_{54}	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 $V^1 = (0\ 1\ 1\ 0\ 1)$ and another pattern $V^2 = (1\ 0\ 1\ 0\ 1)$. One way you could go about calculating the weights is on a weight by weight basis. For example, W_{12} could be calculated as:

$$W_{12} = \sum_{s=1}^{n} (2V^s_1 - 1)(2V^s_2 - 1)$$

$= (2V^1_1 - 1)(2V^1_2 - 1) + (2V^2_1 - 1)(2V^2_2 - 1)$
$= (2*0 - 1)(2*1 - 1) + (2*1 - 1)(2*0 - 1)$
$= (0 - 1)(2 - 1) + (2 - 1)(0 - 1)$

= (-1)(1) + (1)(-1)
= -1 + -1
= -2

After completing all weight computations, 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

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). 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:

$$V_i in = \sum_{j \neq i} W_{ji} V_j$$

V_i -> 1 if $V_i in >= 0$
else V_i -> 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:

$$V_3 in = \sum_{j \neq 3} W_{j3} V_j$$

$= W_{13}V_1 + W_{23}V_2 + W_{43}V_4 + W_{53}V_5$
$= 0*1 + 0*1 + -2*1 + 2*1$
$= 0$

since 0 >= 0,
$V_3 = 1$

In this case, the value of V_3 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**:

$V_3in = (0\ 0\ 0\ -2\ 2) * (1\ 1\ 1\ 1\ 1) = -2 + 2 = 0$

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 synchronous 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

Now let's finish the example I started. In other words, 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 change

- update node 1 -
  V_1in = (0 -2 0 0 0) . (1 1 1 1 1) = -2
  since -2 < 0, V_1 = 0 *(it changed)*
- update node 5 -
  V_5in = (0 0 2 -2 0) . (0 1 1 1 1) = 0
  since 0 >= 0, V_5 = 1 *(it didn't change)*
- update node 2 -
  V_2in = (-2 0 0 0 0) . (0 1 1 1 1) = 0
  since 0 >= 0, V_2 = 1 *(it didn't change)*
- update node 4 -
  V_4in = (0 0 -2 0 -2) . (0 1 1 1 1) = -4
  since -4 < 0, V_4 = 0 *(it changed)*

After one round of updates Vin is now (0 1 1 0 1). Start updating again

- update node 3 -
  V_3in = (0 0 0 -2 2) . (0 1 1 0 1) = 2
  since 2 >= 0, V_3 = 1 *(it didn't change)*
- update node 1 -
  V_1in = (0 -2 0 0 0) . (0 1 1 0 1) = -2
  since -2 < 0, V_1 = 0 *(it didn't change)*
- update node 5 -
  V_5in = (0 0 2 -2 0) . (0 1 1 0 1) = 2
  since 2 >= 0, V_5 = 1 *(it didn't change)*
- update node 2 -
  V_2in = (-2 0 0 0 0) . (0 1 1 0 1) = 0
  since 0 >= 0, V_2 = 1 *(it didn't change)*
- update node 4 -
  V_4in = (0 0 -2 0 -2) . (0 1 1 0 1) = -4
  since -4 < 0, V_4 = 0 *(it didn't change)*

Vin is still (0 1 1 0 1) 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) – a previously stored memory.