Introduction to Neural Networks
U. Minn. Psy 5038
Spring, 1998

"Energy" and neural networks
Discrete Hopfield network
Constraint satisfaction problems

Introduction

This notebook introduces the discrete two-state Hopfield net with an application to a constraint satisfaction problem--the random dot stereogram. In the next notebook, we will study the generalization to graded or continuous valued neural responses, which will lead us to stochastic models of neural networks.

Discrete two-state Hopfield network

■ Background

We've seen how learning can be characterized by gradient descent on an error function e(**W**), in weight-space. Hopfield (1982) showed that for certain networks composed of threshold logic units, the *state vector* , evolved through time in such as way as to decrease the value of a function called an "energy function". Note that now we are holding the weights fixed, and following the value of the energy function as the neural activities evolve through time. This function is associated with energy because the mathematics in some cases is identical to that describing the evolution of physical systems with declining free energy. The Ising model of ferromagnetism developed in the 1920's is, as Hopfield pointed out, isomorphic to the discrete Hopfield net. We will study Hopfield's network in this notebook.

One can define a function that depends on a neural state vector in almost any way one would like, e.g. E(**V**), where **V** is vector whose elements are the neural activites at time t. Then starting with this function, we could compute the time derivative of E(**V**) and set it equal to the gradient of E with respect to **V**. Then as we did for an error function over weight space, we could define a rule for updating **V** (in state space) over time such that we descend E(**V**) in the direction of the steepest gradient at each time step. But this rule would not necessarily correspond to any reasonable neural network model. In two influential papers, Hopfield went the opposite direction. He started off with a reasonable model of neuron processing (the threshold logic unit or TLU), and posited an energy function for it. That is, he showed that with certain restrictions, state vectors for TLU networks descended this energy function as time progressed. The state vectors don't necessarily proceed in the direction of steepest descent, but they don't go up the energy surface. One reason this is useful, is that it shows that these networks converge to a stable state, and thus have well-defined properties for computation.

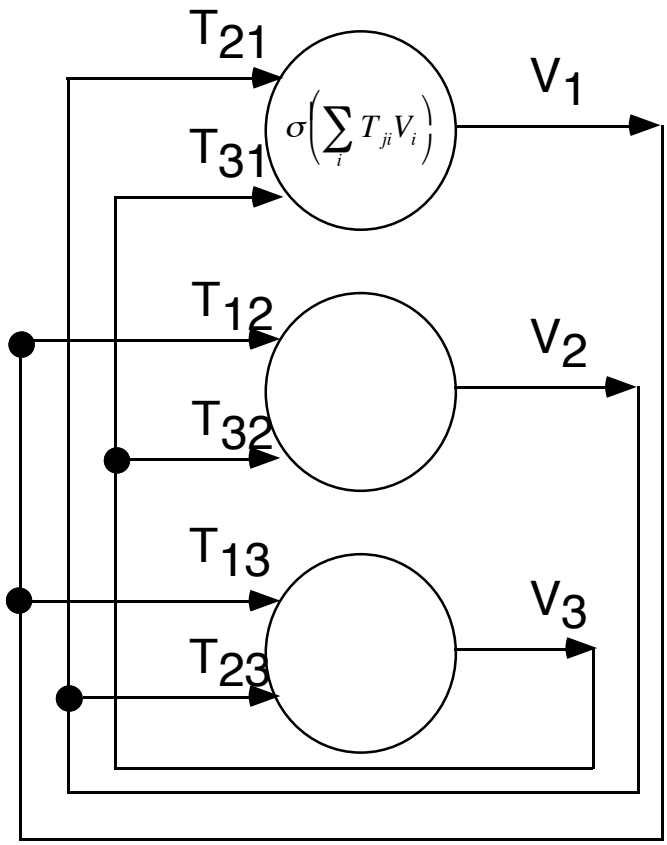
Viewing neural computation in terms of motion over an "energy landscape" provides some useful intuitions. For example, think of memories as consisting of a set of stable points in state space of the neural system--i.e. local minima in which changing the state vector in any direction would only increase energy . Other points on the landscape could represent input patterns or partial data that associated with these memories. Retrieval is a process in which an initial point in state space migrates towards a stable point representing a memory. Mental life may be like moving from one stable state to the next. Hopfield's paper dealt with one aspect: a theory of moving towards a single state and staying there. Hopfield showed conditions under which networks converge to pre-stored memories.

We've already mentioned the relationship of these notions to physics. There is also a large body of mathematics called dynamical systems for which Hopfield nets are special cases. We've already seen an example of a simple linear dynamical system in the limulus equations. In the language of dynamical systems, the energy function is called a Lyapunov function. A useful goal in dynamical system theory is to find a Lyapunov function for a given update rule (i.e. a set of differential equations). The stable states are called "attractors".

Hopfield nets can be used to solve various classes of problems. We've already mentioned memory and recall. They can be used for error correction, as content addressable memories (as in linear autoassociative recall in the reconstruction of missing information), and constraint satisfaction. Energy can be thought of as a measure of constraint satisfaction--when all the constraints are satisfied, the energy is lowest. In this case, one usually wants to find the absolute least global energy. In contrast, for memory applications, we want to find local minima in the energy. Below, we give an example of a Hopfield net which tries to simultaneously satisfy several constraints to solve a random dot stereogram.

■ Basic structure

The network structure consists of TLUs, essentially McCulloch-Pitts model neurons, connected to each other. To follow Hopfield's notation, let T_{ij} be the synaptic weight from neuron j to neuron i. Let V_i be the activity level (zero or one) of unit i.



In Hopfield's analysis, neurons do not connect to themselves--connection matrix has zero diagonal. Further, the connection matrix is symmetric.

$$T_{ii} = 0$$

$$T_{ji} = T_{ij}$$

The weights are fixed ahead of time according to some learning rule, or they could be "hand" and hard-wired to represent some constraints to be satisfied. Hopfield gave a rule for setting the weights according to where the stable states should be.

The neurons can also have bias values (Ui's, not shown in the figure) which are equivalent to individual thresholds for each unit. We use these in the stereo example, below.

■ Dynamics

The network starts off in an initial state (e.g. partial information to be completed by the net, or illegal expressions to be corrected, or stimuli that will produce activity elsewhere in the net representing an association, or ...).

Each neuron computes the weighted sum of inputs, thresholds the result and outputs a 1 or 0 depending on whether the weighted sum exceeds threshold or not. In short, it is a TLU. If the threshold is zero, the update rule is:

$$V_i = \begin{cases} 1 \text{ if } \sum_i T_{ij} V_j > 0 \\ 0 \text{ if } \sum_i T_{ij} V_j < 0 \end{cases}$$

Let's express this rule in the familiar form of a threshold function σ() that is a step function ranging from zero to one, with a transition at zero. Further, if there are bias inputs to each neuron, then the update rule is:

$$V_i = \sigma\left(\sum_j T_{ij} V_j + U_i \right)$$

Only the V's get updated, not the U's.

The updating is *asynchronous*--neurons are updated in random order at random times.

This can be contrasted with synchronous updating in which one computes the output of the net for all of the values at time t+1 from the values at time t, and then updates them all at once. Synchronous updating requires a master clock.

The neat trick is to produce an expression for the energy function. Let's assume the bias units are zero for the moment. What function of the V's and T's will never increase as the network computes?

■ Hopfield's proof that the network descends the energy landscape

Here is the energy function:

$$E = -\frac{1}{2} \sum T_{ij} V_i V_j$$

Let's prove that it has the right properties. We are going to use *Mathematica* to save some ink, and get an idea of what happens to E when one unit changes state (i.e. goes from 0 to 1, or from 1 to 0).

Let's write a symbolic expression for the Hopfield energy function for a 5-neuron network using *Mathematica*. We construct a symmetric connection weight matrix T in which the diagonal elements are zero.

```
T = Table[Which[i==j,0,i<j,t[i,j],i>j,t[j,i]],{i,1,5},
{j,1,5}];
V1 =  Array[v1,{5}];
```

```
energy = (-1/2) (T.V1).V1;
```

The next few lines provide us with an idea of what the formula for a change in energy should look like when one unit (unit 2) changes. Let V2 be the new vector of activities (the same as V1 except for unit 2).

```
V2 = V1;
V2[[2]] = v2[2];
delataenergy = - (1/2) (T.V2).V2 - (-(1/2) (T.V1).V1);
```

```
Simplify[delataenergy]
```

```
(t[1, 2] v1[1] + t[2, 3] v1[3] + t[2, 4] v1[4] + t[2, 5] v1[5])
  (v1[2] - v2[2])
```

Now you can generalize this expression to N-neurons, and an arbitrary ith unit (rather than number 2).

$$\Delta E = -\Delta V_i \sum_{i \neq j} T_{ij} V_j$$

Hopfield showed that any change in the ith unit (e.g. if V[[2]] goes from 0 to 1, or from 1 to 0) if it follows the TLU rule, will not increase the energy--i.e. deltaenergy will not be positive.

The proof is simple.

Case 1: If the weighted sum is negative, then the change in V must either be zero or negative. This is because for the weighted sum to be negative, V must have gone to a zero, and this can only happen if it was zero or one. The product of deltaV and summation term is zero or positive (product of two negative terms), so deltaE is zero or negative.

Case 2: If the weighted sum is positive, the change in V must be either zero or positive. Again, because the weighted sum is positive, V must have gone to a +1--either it was a one and remained so, or changed in the positive direction from a zero. The product of deltaV and summation term is zero or positive (product of two positive terms), so again deltaE is zero or negative.

■ Including adjustable thresholds via additional current inputs.

We can add non-zero thresholds, U, and additional bias terms, I, and easily generalize the basic result. The update rule then is:

$$V_i = \begin{cases} 1 \text{ if } \sum_{j \neq i} T_{ij} V_j + I_i > U_i \\ 0 \text{ if } \sum_{j \neq i} T_{ij} V_j + I_i < U \end{cases}$$

The energy function becomes:

$$E = -\frac{1}{2}\sum_{i \neq j} T_{ij} V_i V_j - \sum_i I_i V_i + \sum_i U_i V_i$$

And you can verify for yourself that a change in state of one unit always leaves deltaenergy zero or negative:

$$\Delta E = -\left[\sum_{j \neq i} T_{ij} V_j + I_i - U_i\right] \Delta V_i$$

Applications of the Hopfield discrete net to Memory

It is not hard to set up the Hopfield net to recall the TIP letters from partial data. What we'd need is some rule for "sculpting" the energy landscape, so that local minima correspond to the letters "T", "I", and "P". We will see how to do this in the next lecture, when we study Hopfield's generalization of the discrete model to one in which neurons have a graded response. At that point we will show how the Hopfield net overcomes limitations of the linear associator. Remember the linear network fails to make discrete decisions when given superimposed inputs.

In the rest of this section, we show how the Hopfield and related algorithms can be used to solve a constraint satisfaction problem.

Constraint satisfaction

■ Hand-wiring the constraints in a neural net.

How does one "sculpt the energy landscape"? One can use a form of Hebbian learning to dig holes (i.e. stable points or attractors) in the energy landscape indicating things to be remembered. Alternatively, one can study the nature of the problem to be solved and hand-wire the network to represent the constraints (i.e. guess at the weights). We will follow the second approach to solve the correspondence problem which crops up in a number of domains in pattern theory and recognition. In particular establishing correspondences between two similar but not quite identical patterns has been a challenging problem in both stereo vision and motion processing.

In the next few sections, we will show how the weights in a network can be set up to represent constraints. Then we will look at three ways of letting the network evolve: asynchronous, synchronous, and partially asychronous updating. The first case exactly satisfies the assumptions required for Hopfield's energy function.

Establishing correspondences: An example of constraint satisfaction

Introduction to stereo and the correspondence problem

If you cross your eyes so that you can perceptually fuse the two random patterns below, you may be able to see a small square floating in front of the random background. This is an example of a random dot stereogram originally developed by Bela Julesz in the 1960's.

It is made by taking a small square sub-patch in one image, shifting it by a pixel or two, to make the second image. Since the subpatch shifted, it leaves a sub-column of pixels unspecified. These get assigned new random values. That's how to make a random stereogram. How does one "solve it"? Human perception solves it, so let us see if we can devise an algorithm to solve it.

The big problem is to figure out which of the pixels in the left eye belong to which ones in the right. A small minority don't have matching pairs (i.e. the ones that got filled in the vertical slot left after shifting the sub-patch). We'll get to this in a moment, but first let's make a stereogram using *Mathematica*.

Making a random dot stereogram using the Map operation

■ Getting and putting pieces of a matrix

There are lots of ways to place a patch at two different locations in two images. We will develop one. *Mathematica* lets you pull submatrices out of a matrix, but doesn't have a function to put submatrices in. For example, the following command pulls out a section for rows 1 and 2, and columns 2 and 3:

```
MatrixForm[ mm = {{1,2,3},{3,4,5},{6,7,8}}   ]
MatrixForm[ mm[[ Range[1,2], Range[2,3] ]]   ]
```

$$\begin{pmatrix} 1 & 2 & 3 \\ 3 & 4 & 5 \\ 6 & 7 & 8 \end{pmatrix}$$

$$\begin{pmatrix} 2 & 3 \\ 4 & 5 \end{pmatrix}$$

Note that Range[] is a function for making lists spanning a range:

```
Range[2,4]
```

```
{2, 3, 4}
```

So how can we put a small matrix into a larger matrix? We do it using

MapAt[function, expression, positions]. **MapAt** will apply a function to the **twoDleftey**e matrix that replaces values at the specified positions with the values from **patch**. Although you could easily write a simple routine that looped through the large matrix inserting the patch, it is particularly easy to modify the function below to insert bits at arbitrary locations, just by setting up the list of coordinates appropriately.

■ Initialize

Let's make a simple random dot stereogram in which a flat patch is displaced by disparity pixels to the left in the left eye's image. il and jl are the lower left positions of the lower left corner of patch in the twoDlefteye. Similarily, ir and jr are the lower left insert positions in the twoDrighteye matrix.

In order to help reduce the correspondence problem later, we can increase the number of gray-levels, or keep things difficult with just a few--below we use four gray-levels.

```
size = 32; patchsize = size/2;
```

```
twoDlefteye = Table[Random[Integer,3],{size},{size}];
twoDrighteye = twoDlefteye;
patch = Table[Random[Integer,3],
{patchsize},{patchsize}];
```

■ Left eye

The left eye's view will have the **patch** matrix displaced one pixel to the left.

```
disparity = 1;
il = size/2-patchsize/2 + 1; jl = il  - disparity;
```

```
i=1;
label[x_] := Flatten[patch][[i++]];
twoDlefteye = MapAt[label, twoDlefteye,
    Flatten[Table[{i,j},
        {i,il,il + Dimensions[patch][[1]]-1},
    {j,jl,jl + Dimensions[patch][[2]]-1}],1] ];
```

■ Right eye

The right eye's view will have the **patch** matrix centered.

```
ir = size/2-patchsize/2 + 1; jr = ir;
```

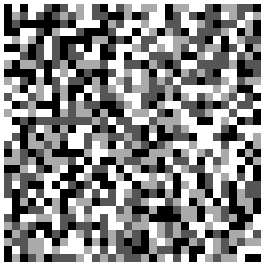
```
i=1;
label[x_] := Flatten[patch][[i++]];
twoDrighteye = MapAt[label, twoDrighteye,
    Flatten[Table[{i,j},
    {i,ir,ir + Dimensions[patch][[1]]-1},
    {j,jr,jr + Dimensions[patch][[2]]-1}],1] ];
```

■ Display a pair of images

It is not easy to fuse the left and right image without a stereo device (it requires placing the images side by side and crossing your eyes. We can check out our images another way.

The visual system also solves a correspondence problem over time. Let's place the two images in cells above each other and then group them. By double-clicking on the cell marker that groups the two images we can collapse them to one cell. Holding down the apple key and the Y key at the same time plays these two pictures in a movie. By slowing it down, you can find a speed in which the central patch almost magically appears to oscillate and float above the the background. If you stop the animation (click on the ‖ of the VCR control panel at the bottom of the window), the central square patch disappears again into the camouflage.

```
ListDensityPlot[twoDlefteye,Mesh->False,
    Frame->False,Axes->False];
ListDensityPlot[twoDrighteye,Mesh->False,
    Frame->False,Axes->False];
```



Two-state neural network implementation using Marr and Poggio (1976) constraints

■ Reduce the problem to one dimension

We will apply the constraints proposed by Marr and Poggio (1976) to try to solve the correspondence problem for just the middle rows of **twoDlefteye** and **twoDrighteye**:

```
lefteye = twoDlefteye[[size/2]];
righteye = twoDrighteye[[size/2]];
```

Because the patch was shifted horizontally, we haven't lost the essense of the problem by reducing it to one dimension.

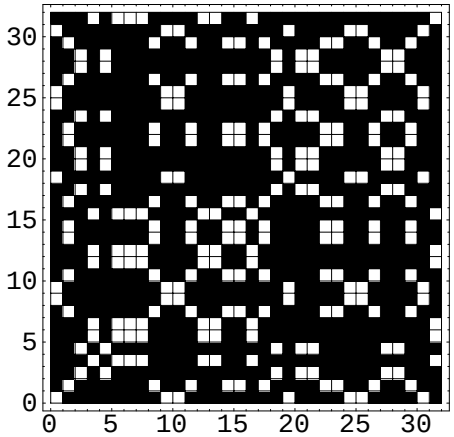
Following Marr and Poggio, we will try to solve the correspondence (i.e. which pairs of pixels in the two images belong together) using three constraints: *compatibility, uniqueness,* and *smoothness*. We will see what these constraints mean as we move along. The challenge is to design a network that enforces these constraints.

■ The compatibility constraint

The compatibility constraint says that like pixels in each image should match. Let's construct a compatibility matrix which is has a one at each location where there is a possible match, and zeros elsewhere.

```
compatibility = Table[If[
        lefteye[[i]]==righteye[[j]],1,0],
        {i,size},{j,size}];
```

```
ListDensityPlot[compatibility];
```



■ The uniqueness and smoothness constraints

We have to find a line through the compatibility matrix to indicate unique matches. So we have to discourage more than one unit from being on in any given row or column (i.e. enforce a uniqueness constraint), but encourage elements that have nearest neighbor support along the diagonals to be on (to encourage regions that have constant disparity). This latter constraint is a smoothness constraint, which refers to the underlying assumption that changes in depth usually change gradually--sudden changes in disparity are rare. (Disparity refers to the relative shift of corresponding points in the two images.)

We will follow Marr and Poggio and set up a threshold logic unit at each location of the compatibility matrix. We are going to have to worry about the boundaries. There are several ways of doing this. One is to have a "free" boundary in which the connection weights at the boundaries are actually different (to compensate for a lack of neighbors in the other directions). A second way is to use a toroidal geometry, restricting indices by the following modulus function: **`myMod[x_] := Mod[x-1,size]+1`**. This option makes it possible to comply with the restriction of symmetric connections everywhere.

The vertical and horizontal connection weights are all equal and negative with a weight **inhib**. The diagonal support is positive with weight **excit**. The network will have biases for each unit (which are equivalent to appropriate thresholds for each unit) proportional to the original compatibility matrix. These biases correspond to the Ui's in the Hopfield net and serve to prevent the network from loosing this strong constraint from the data as the iterations progress.

Note that we won't set up the network calculations using a matrix and matrix multiplications as you will do in the Hopfield memory examples. This is because most of our connections are zero and we don't want to waste time multiplying zero elements and adding them up.

Hopfield Net: Asynchronous updating--all sites visited randomly, at random times

To do random asynchronous updating, you simply pick a site at random and update it, and then pick another, and so on.

```
V = compatibility; V1 = V;
```

```
excit = 2; inhib = -1; k = 6; theta = 13;
threshold[xx_] := N[If[xx > theta, 1, 0]];
```

The neighborhood size along the diagonal, horizontal, and vertical directions are each 8. We use constant weighting.

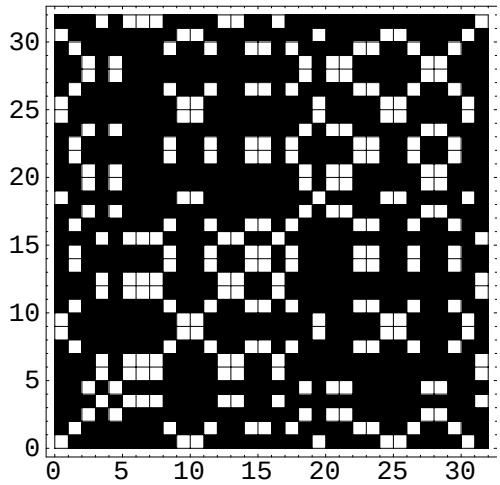
```
numiterations = 10000;
For[iter = 1, iter <= numiterations, iter++,
 If[Mod[iter, 1000] == 1, ListDensityPlot[V1]];
 i = Random[Integer, size - 1] + 1; j = Random[Integer, size - 1] + 1;
V1[[i, j]] = threshold[
  inhib (
     V1[[myMod[i + 1], myMod[j]]] + V1[[myMod[i - 1], myMod[j]]] +
       V1[[myMod[i], myMod[j - 1]]] + V1[[myMod[i], myMod[j + 1]]]

     V1[[myMod[i + 2], myMod[j]]] + V1[[myMod[i - 2], myMod[j]]] +
       V1[[myMod[i], myMod[j - 2]]] + V1[[myMod[i], myMod[j + 2]]]

     V1[[myMod[i + 3], myMod[j]]] + V1[[myMod[i - 3], myMod[j]]] +
       V1[[myMod[i], myMod[j - 3]]] + V1[[myMod[i], myMod[j + 3]]]

     V1[[myMod[i + 4], myMod[j]]] + V1[[myMod[i - 4], myMod[j]]] +
       V1[[myMod[i], myMod[j - 4]]] + V1[[myMod[i], myMod[j + 4]]]

     ) +
  excit (
      V1[[myMod[i + 1], myMod[j + 1]]]  + V1[[myMod[i - 1], myMod[j - 1]]] +
      V1[[myMod[i + 2], myMod[j + 2]]]  + V1[[myMod[i - 2], myMod[j - 2]]] +
      V1[[myMod[i + 3], myMod[j + 3]]]  + V1[[myMod[i - 3], myMod[j - 3]]] +
      V1[[myMod[i + 4], myMod[j + 4]]]  + V1[[myMod[i - 4], myMod[j - 4]]]
      )
   + k compatibility[[i, j]] ] ;
];
```



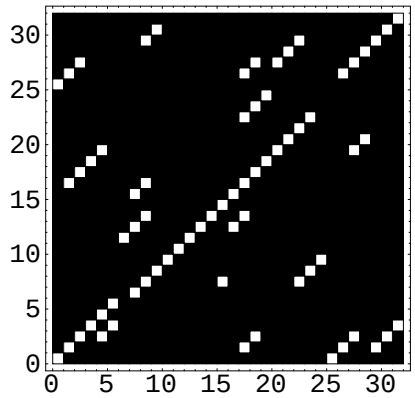
Hopfield Net: Small neighborhood

```
excit = 2; inhib = -1; k = .5; theta = 2.25;
threshold[xx_] := N[If[xx > theta, 1, 0]];
V = compatibility; V1 = V;
```

```
myMod[x_] := Mod[x - 1, size] + 1;
```

```
numiterations = 1000;
For[iter = 1, iter <= numiterations, iter++,
 i = Random[Integer, size - 1] + 1; j = Random[Integer, size - 1] + 1;
V1[[i, j]] = threshold[
  inhib (
     V1[[myMod[i + 1], myMod[j] ]] + V1[[myMod[i - 1], myMod[j] ]] +
       V1[[myMod[i], myMod[j - 1] ]] + V1[[myMod[i], myMod[j + 1] ]]
         ) +
  excit (V1[[myMod[i + 1], myMod[j + 1]]]  + V1[[myMod[i - 1], myMod[j - 1]]])
   + k compatibility[[i, j]] ] ;
];
```

```
ListDensityPlot[V1];
```



Hopfield Net: Almost asynchronous updating--all sites visited randomly, but once on each iteration

Here each unit is updated invidually, in random order, but without duplication. You can either read in the **Permutations.m** file in the **DiscreteMath** directory, or define your own function to produce a random nonrepeating assortment of indices from **2** to **size**-1 (in order to avoid indexing beyond the boundaries).

Let's define a function that samples a list of integers without replacement. This simple routine illustrates how to specify local variables using the **Module[]** function (the local variable is **t**). It also shows another example of how to use the **Map[]** function together with the notion of *pure functions*.

■ Pure functions and Map[]

Here are three ways of doing the same thing:

```
square[x_] := x^2;
Map[square, a+b+c]
```

$a^2 + b^2 + c^2$

Because we may only be defining the squaring operation once, just to be used in the Map operation, we can do it using a pure function **Function[]**.

```
Map[Function[x,x^2], a+b+c]
```

$a^2 + b^2 + c^2$

This kind of operation is done often enough, that <body> & is used to represent the function, and # is used to represent the first variable for the pure function.

```
Map[#^2 &, a+b+c]
```

$a^2 + b^2 + c^2$

You can pick out the columns of a matrix using Map:

```
MatrixForm[mat = {{1,2},{3,4}}]
```

```
1   2
3   4
```

```
Map[#[[2]] &, %]
```

```
{2, 4}
```

■ RandomPermutation2 routine

Here is a function that returns a randomly permuted list of number from 2 to N:

```
RandomPermutation2[n_Integer?Positive] :=
    Module[{t},
        t = Array[{Random[], #+1} &, n];
        t = Sort[t];
        Map[ #[[2]] &, t ]
    ]
```

```
RandomPermutation2[10]
```

```
{8, 9, 10, 6, 11, 2, 5, 3, 7, 4}
```

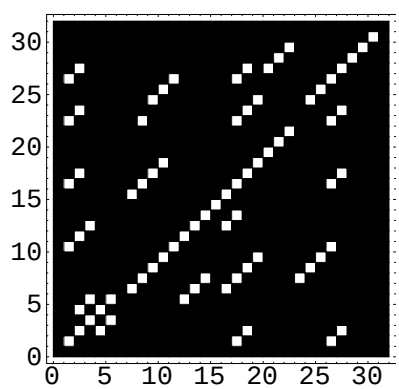
■ Routine to search for correspondences using partial asychronous updating

```
For[i=1,i<=size,i++,
    compatibility[[1,i]] = 0;
    compatibility[[size,i]] = 0;
    compatibility[[i,size]] = 0;
    compatibility[[i,1]] = 0;
    ];
```

```
V = compatibility;
```

```
For[iter=1,iter<=numiterations,iter++,
iindex = RandomPermutation2[size-2];
jindex = RandomPermutation2[size-2];
For[i=1,i<size-1,i++,
For[j=1,j<size-1,j++,
    V[[iindex[[i]],jindex[[j]]]] = threshold[
    inhib (V[[ iindex[[i]]+1, jindex[[j]]  ]] +
    V[[  iindex[[i]]-1, jindex[[j]]  ]] +
    V[[  iindex[[i]], jindex[[j]]-1   ]] +
    V[[  iindex[[i]],  jindex[[j]]+1   ]]) +
    excit (V[[  iindex[[i]]+1,  jindex[[j]]+1   ]] +
    V[[  iindex[[i]]-1,  jindex[[j]]-1    ]]) +
    k compatibility[[  iindex[[i]],  jindex[[j]]    ]]  ]   ]]];
```

```
ListDensityPlot[V];
```



Marr & Poggio's Net: Synchronous updating

Here we will run through all the units and compute their values for the **iter+1**th iteration based on their values on the **iter**th iteration. So they all get updated at once, right at the end of the two internal loops when we set **V = V1**. The starting values are just the compatibility matrix itself. Here, we will clamp the boundary to zero, and not update these elements.

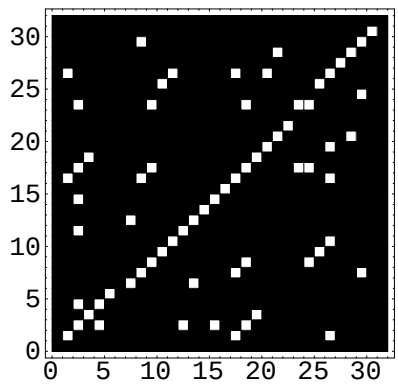
```
For[i=1,i<=size,i++,
    compatibility[[1,i]] = 0;
    compatibility[[size,i]] = 0;
    compatibility[[i,size]] = 0;
    compatibility[[i,1]] = 0;
    ];
```

```
V = compatibility; V1 = V;
```

```
excit = 2; inhib = -1; k = .5; theta = 2.0;
threshold[xx_] := N[If[xx>theta,1,0]];
```

```
numiterations = 1;
For[iter=1,iter<=numiterations,iter++,
    For[i=2,i<size,i++,
    For[j=2,j<size,j++,
        V1[[i,j]] = threshold[
        inhib (V[[i+1,j]] + V[[i-1,j]] +
        V[[i,j-1]] + V[[i,j+1]]) +
        excit (V[[i+1,j+1]] + V[[i-1,j-1]])
        + k compatibility[[i,j]]    ]   ]];
    V = V1; ];
```

```
ListDensityPlot[V1];
```



Our constraints move the solution in the right direction, but it doesn't solve the problem completely. In fact this algorithm requires quite a bit of tweaking for each random dot stereogram.

I will leave this one hanging--with no apology--for you to improve. You can develop some appreciation for the design difficulties posed by even simple constraint satisfaction problems for the neural network modeler.

One addition you could try is to increase the range of the vertical and horizontal inhibition to help force the uniqueness constraint.

Energy and constraint satisfaction

What use is energy?

We haven't computed the Hopfield energy for each iteration. But we could, and in fact this calculation can be useful.

In our toy example of a random dot stereogram, we know the answer (we made the stereogram), so computation of the energy of the final stable state can be compared with the energy of the right answer. If the right answer has a lower energy than the stable state reached, we know we got stuck in a local minimum. If the right answer has a higher energy, then the constraints must have been set up incorrectly. So energy can be a useful number to calculate during network model development.

References

Hopfield, J. J. (1982). Neural networks and physical systems with emergent collective computational abilities. Proc. Natl. Acad. Sci. USA, 79, 2554-2558.

Hopfield, J. J. (1984). Neurons with graded response have collective computational properties like those of two-state neurons. Proc. Natl. Acad. Sci. USA, 81, 3088-3092.

Hopfield, J. J. (1994). Neurons, Dynamics and Computation. Physics Today, February.

Marr, D., & Poggio, T. (1976). Cooperative computation of stereo disparity. Science, 194, 283-287.

Marr, D., Palm, G., & Poggio, T. (1978). Analysis of a cooperative stereo algorithm. Biol. Cybernetics, 28, 223-239.

© 1998 Daniel Kersten, Computational Vision Lab, Department of Psychology, University of Minnesota.