Introduction to Neural Networks
U. Minn. Psy 5038

"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($\mathbf{W}$), in weight-space. We are going to study analogous dynamics for recall behavior, rather than learning.

Hopfield (1982) showed that for certain networks composed of threshold logic units, the *state vector* evolves through time in such as way as to decrease the value of a function called an "energy function". Here 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($\mathbf{V}$), where $\mathbf{V}$ is vector whose elements are the neural activites at time t. Suppose we could specify E($\mathbf{V}$) in such a way that small values of E are "good", and large values are "bad". In other words low values of E($\mathbf{V}$) tell us when the network is getting close to a right answer--as in the search game "you're getting warmer", but with the sign in the energy metaphor reversed. Then starting with this function, we could compute the time derivative of E($\mathbf{V}$) and set it equal to the gradient of E with respect to $\mathbf{V}$. Then as we did for an error function over weight space, we could define a rule for updating $\mathbf{V}$ (but now in state space) over time such that we descend E($\mathbf{V}$) in the direction of the steepest gradient at each time step, and thus we'd go from "bad" to "better" to "good" solutions. But this gradient derived 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 model of neural network connectivity using the threshold logic units (TLU) for neurons, and posited an energy function for network activity. 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 showed 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. With this metaphor, mental life may be like moving from one (quasi) 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.
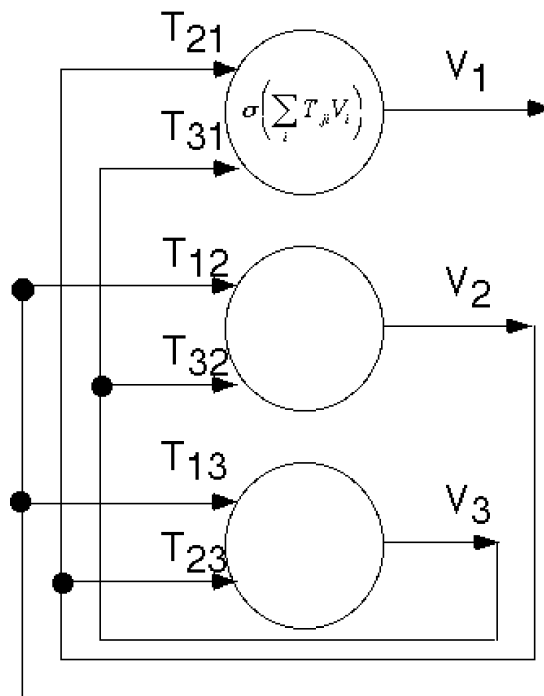
We've already mentioned the relationship of these notions to physics. There is also a large body of mathematics called dynamical systems for which Hopfield nets are special cases. We've already seen an example of a simple linear dynamical system in the limulus equations with recurrent inhibition. In 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. Hopfield nets can be used for error correction, as content addressable memories (as in linear autoassociative recall in the reconstruction of missing information), and constraint satisfaction.

Consider the latter case. Energy can be thought of as a measure of constraint satisfaction--when all the constraints are satisfied, the energy is lowest. Energy represents the cumulative tension between competing and cooperating constraints. 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. In this case, if E(V) ends up in a local minimum, that is usually not a good solution. Later we will learn ways to get out of local minimum for constraint satisfaction problems.

■ Basic structure

The (discrete) Hopfield 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--the connection matrix T has a 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 (e.g. outer-product form of Hebb), 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) that 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 which, depending on the network's particular application, may represent:

>   --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

>   --initial hypotheses to constrain further computation, or...

...or we can just set the initial state randomly, and then investigate its convergence properties.

As a TLU, 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. If the threshold is zero, the update rule is:

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

Let's express this rule in the familiar form of a threshold function $\sigma()$ that is a step function ranging from zero to one, with a transition at zero. Further, if there are bias inputs (U_i's) 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 U_i's, can also be thought of as adjustable thresholds, that could be learned or set by the application. Recall that we could fold the U_i's into the weights, and fix their inputs to 1. The decision whether or not to do this depends on the problem.)

The updating for the Hopfield net 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 buffer and a master clock. Asynchronous updating seems more plausible for many neural network implementations, because an individual TLU updates its output state whenever the information comes in.

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 trick--the energy function:

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

Let's prove that it has the right properties. Because of the asynchronous updating, we'll assume it is OK to consider just one unit at a time. To be concrete, consider say, unit 2 of a 5 unit network.

We are going to use *Mathematica* to 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.

```
Tm = Table[Which[i==j,0,i<j,T[i,j],i>j,T[j,i]],{i,1,5},
{j,1,5}];
V1m =  Array[V1,{5}];
```

```
energy = (-1/2) (Tm.V1m).V1m;
```

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).

```
V2m = V1m;
V2m[[2]] = V2[2];
delataenergy = - (1/2) (Tm.V2m).V2m - (-(1/2) (Tm.V1m).V1m);
```

```
Simplify[delataenergy]
```

$$(V1(2) - V2(2)) (V1(1)\, T(1, 2) + V1(3)\, T(2, 3) + V1(4)\, T(2, 4) + V1(5)\, T(2, 5))$$

Or rearranging: - (V2(2) - V1(2))*(T(1, 2) V1(1) + T(2, 3) V1(3) + T(2, 4) V1(4) + T(2, 5) V1(5))

Let $\Delta V_i = (V2(i) - V1(i))$. Now you can generalize the above expression to N-neurons, and an arbitrary ith unit (rather than number 2).

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

Using the above formula, Hopfield then showed that any change in the ith unit (if it follows the TLU rule), will not increase the energy--i.e. deltaenergy ΔE will not be positive. E.g. if $V_{i=2} = V[[2]]$ goes from 0 to 1, or from 1 to 0, the energy change is negative or zero. In general, ΔE is either zero or negative:

∎

$\Delta E \leq 0$

The proof is straightforward. Consider the weighted sum:

$$\sum_{i \neq j} T_{ij} V_j$$

Case 1: If the weighted sum is negative, then the change in V_i must either be zero or negative.

This is because for the weighted sum to be negative, the TLU rule (equation 1) says that V_i must be set to zero. How did it get there? Either it was a zero and remained so (then $\Delta E=0$), or changed in the negative direction from a one to a zero (i.e. $\Delta V_i = -1$). In the latter case, the product of ΔV_i and summation term is positive (product of two negative terms), so ΔE is negative (i.e., the sign of the change in energy is the product of MINUS*MINUS*MINUS = MINUS).

Case 2: If the weighted sum is positive, the change in V_i must be either zero or positive.

Because the weighted sum is positive, by the TLU rule V_i must have been set to a +1--either it was a one and remained so ($\Delta V_i = 0$, then $\Delta E=0$), or changed in the positive direction from a zero ($\Delta V_i = +1$). In the latter case, the product of ΔV_i and the summation term is positive (product of two positive terms), so ΔE is negative (i.e., the sign of the change in energy is the product of MINUS*PLUS*PLUS = MINUS).

■ Including adjustable thresholds via additional current inputs.

Hopfield generalized the update rule to add non-zero thresholds, U, and additional bias terms, I, and then generalized the basic result with added terms to the energy function. The update rule then has the familiar form for TLU nets with bias:

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

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 associative 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 applied to constraint satisfaction problems. We'll also see examples of the local minima 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. reason out, or make an educated guess as to what the weights should be).

We will follow the second approach to solve the correspondence problem. This problem crops up in a number of domains in pattern theory and recognition, and occurs whenever the features in two patterns need to be matched up, but one pattern is an unknown distortion of the other. Imagine, for example, the image of a face in memory, and you want to test whether an incoming stimulus is from the same person. Both patterns have more or less the same features, but they don't superimpose because they are from different views or expressions. So you want to try to morph one on to the other, but before you do that, you might want to decide which features go with which--i.e. pupils to pupils, nostrils to nostrils, etc.. Establishing correspondences between two similar but not quite identical patterns has also been a central, and challenging problem in both stereo vision and motion processing.
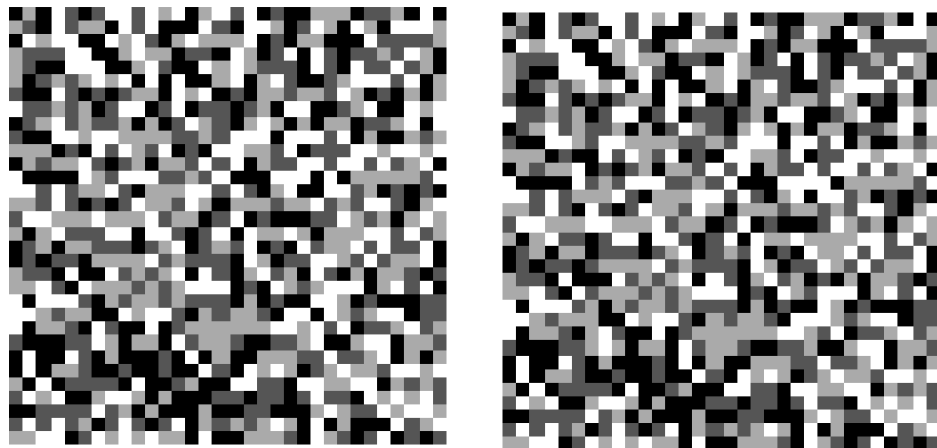
In the next few sections, we will show how the weights in a network can be set up to represent constraints. Then we will look at three ways of letting the network evolve: asynchronous, synchronous, and partially asynchronous updating. The first case exactly satisfies the assumptions required for Hopfield's 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. Crossing your eyes means that the left image goes to the right eye, and the right to the left. (Some people are better at looking at the left image with the left eye, and right with the right eye. For this type of human, the images below should be exchanged.)

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 is shifted, it leaves a sub-column of pixels unspecified. These get assigned new random values.

The stereogram mimics what happens in the real world when you look at an object that stands out in depth from a background--the left eye's view is slightly different than the right eye's view. The pixels for an object in front are shifted with respect to the background pixels in one eye's view compared to the other. There is a *disparity* between the two eyes. The distances between two points in the left eye and the distance of the images of the same two points in the right eye are, in general different, and depend on the relative depth of the two points in the world.

To see depth in a random dot stereogram, the human visual system effectively solves a *correspondence problem*. The fundamental problem is to figure out which of the pixels in the left eye belong to which ones in the right. This is a non-trivial computational problem when so may of the features (i.e. the pixel intensities) look the same--there is considerable potential for false matches. 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*.

Human perception solves the stereo correspondence, so let us see if we can devise a neural network style algorithm to solve it.

Making a random dot stereogram using the Map operation

■ **Review: Getting and putting pieces of a matrix**

There are lots of ways to place a patch at two different locations in two images. We'll define a function to put submatrices in. *Mathematica* lets you pull submatrices out of a matrix. For example, the following shows three commands for pulling out a section consisting of rows 1 and 2, and columns 2 and 3:

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

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

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

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

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

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

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

{2, 3, 4}

There are several ways to insert a submatrix into a larger matrix (e.g. using **ReplacePart[]**). We"ll insert a patch of pixels into the left and right eye using **MapAt[function, expression, positions]**. **MapAt** will apply a function to the **twoDlefteye** 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 = RandomInteger[3, {size, size}]; twoDrighteye = twoDlefteye;
patch = RandomInteger[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] ];
```
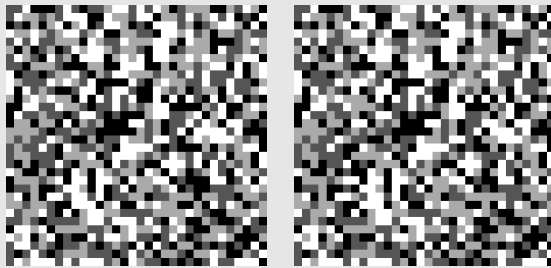
■ **Display a pair of images**

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 (or use the menu choice, Graphics:Rendering:Animate Selected Graphics). 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.

```
gl = ArrayPlot[twoDlefteye, Mesh → False, Frame → False, Axes → False,
   ImageSize → Tiny]
gr = ArrayPlot[twoDrighteye, Mesh → False, Frame → False, Axes → False,
   ImageSize → Tiny]
```

```
GraphicsRow[{gl, gr}]
```


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

■ **Simplify 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 ]];

ArrayPlot[{lefteye, righteye}, AspectRatio → 1/8, Mesh → True]
```


We've displayed the left eye's view above the right eye's, so you can try to spot which pixels have been shifted. Because the patch was shifted horizontally, we haven't lost the essence of the correspondence problem by reducing it to one dimension. You can see the ambiguity in the correspondence problem.

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 similar features in each eye should match, in other words like pixels (same graylevel) in each image should match. We can represent compatibility by constructing a "compatibility matrix" which has a 1 at each location where there is a possible match (i.e. in graylevel color), and zeros elsewhere.
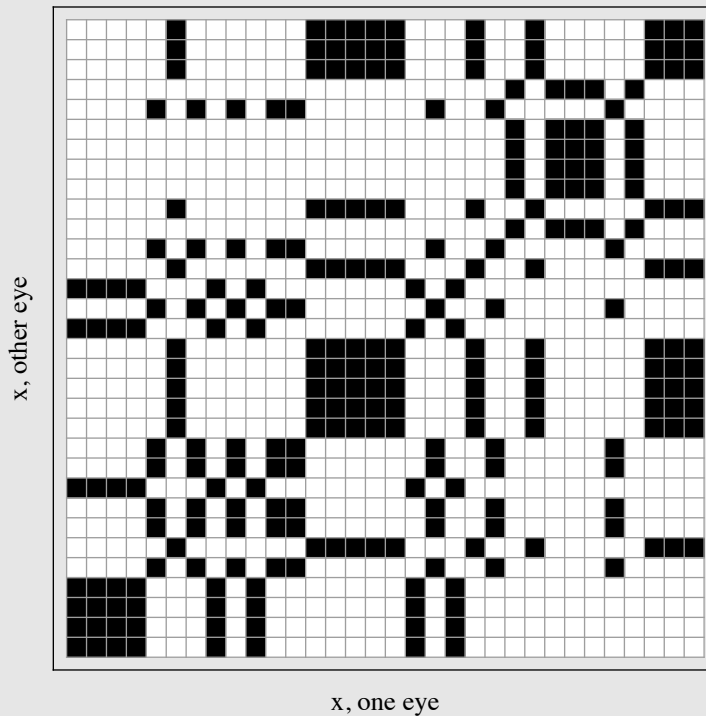
```
compatibility = Table[If[
        lefteye[[i]]==righteye[[j]],1,0],
        {i,size},{j,size}];
```

Here's the plot of the compatibility of the middle row (size/2) for the left and right eye's stereo pictures, where black represents 1 (compatible), and white 0 (not compatible).

```
ArrayPlot[compatibility, DataReversed → True, Mesh → True,
 Frame → True, FrameLabel → {"x, other eye", "x, one eye"}]
```


■ The uniqueness and smoothness constraints

But many of these possible matches seem unlikely. A given point in one eye shouldn't have lots of matches in the other eye--a point on a surface in the world typically maps to at most one point in each eye. We would like 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*).

Also when we think about how features along surfaces typically change as a function of position, we realize that surfaces to a first approximation are smooth--thus, nearby points on a surface project to nearby points in the two eyes. For convenience, let's assume an even stronger constraint in which nearby points have constant disparities. (Recall that disparity refers to the relative shift of corresponding points in the two images.)

This means to we should encourage elements that have nearest neighbor support along the diagonals to be on (to encourage regions that have constant disparity). This latter constraint is called a "surface smoothness constraint", which refers to the underlying assumption that changes in depth usually change gradually--sudden changes in disparity are rare.

We will follow Marr and Poggio and set up a threshold logic unit at each location of the compatibility matrix, and wire each up to reflect the above constraints. 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). Although not biologically realistic, 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 is computationally convenient and makes it possible to comply with the restriction of symmetric connections everywhere.

To encourage uniqueness, let's make the vertical and horizontal connection weights all equal and negative with a weight **inhib**. To encourage smoothness, we'll make the diagonal support positive with mutually excitatory weights **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. So the code below will look messier.

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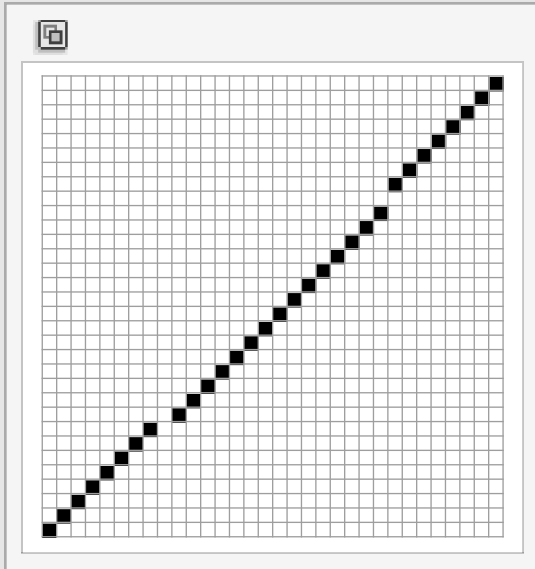
```
V = compatibility; V1 = V;
```

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

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

The neighborhood size along the diagonal, horizontal, and vertical directions are each 8. We use constant weighting. Although the code looks messy, it is doing something pretty simple. Most of the lines of code are just specifying which neighboring elements of the compatibility matrix should excite or inhibit the unit being updated.

```
DocumentNotebook[
 Dynamic[ArrayPlot[V1, DataReversed → True, Mesh → True,
   ImageSize → Small]]]
```



```
Timing[numiterations = 10 000; For[iter = 1, iter ≤ numiterations, iter++,
  If[Mod[iter, 10] == 1, ArrayPlot[V1, Mesh → True]];
  i = RandomInteger[size - 1] + 1; j = RandomInteger[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〛];];]
```

{16.4135, Null}

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. Energy can be a useful number to calculate during network model development.

▌ References

Hertz, J., Krogh, A., & Palmer, R. G. (1991). Introduction to the theory of neural computation (Santa Fe Institute Studies in the Sciences of Complexity ed.). Reading, MA: Addison-Wesley Publishing Company.

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.

Mayhew, J. E. W., & Frisby, J. P. (1981). Psychophysical and Computational Studies Towards a Theory of Human Stereopsis. Artificial Intelligence, 17, 349-385.

Yuille, A. L., Geiger, D., & Bülthoff, H. H. (1991). Stereo integration, mean field theory and psychophysics. Network, 2, 423-442.