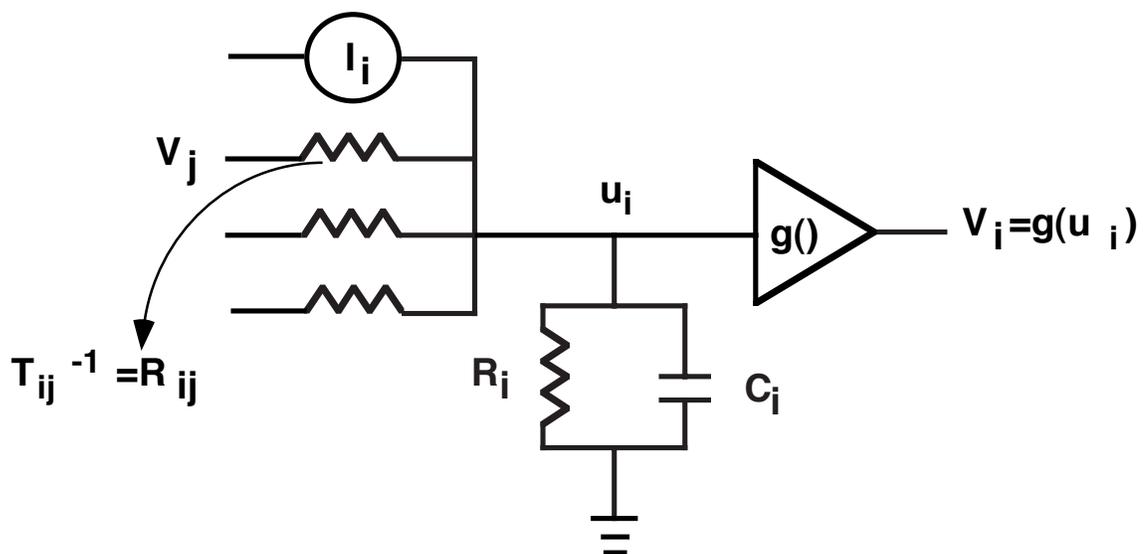


## Graded response Hopfield network

### ■ The model of the basic neural element

Hopfield's 1982 paper was strongly criticized for having an unrealistic model of the neuron. In 1984, he published another influential paper with an improved neural model. The model was intended to capture the fact that neural firing rate can be considered a continuously valued response (recall that frequency of firing can vary from 0 to 500 Hz or so). Earlier we derived an expression for the rate of firing for the "forgetful integrate-and-fire" model of the neuron. Hopfield adopted the basic elements of this model, together with the assumption of a non-linear sigmoidal output, represented below by an operational amplifier with non-linear transfer function  $g()$ . An operational amplifier (or "op amp") has a very high input impedance so it essentially draws no current.

Below is the electrical circuit corresponding to the model of a single neuron. The unit's input is the sum of currents (input voltages weighted by conductances  $T_{ij}$ , corresponding to synaptic weights).  $I_i$  is a bias input current (which is often set to zero depending on the problem). There is a membrane resistance  $R_i$ , and capacitance  $C_i$ —that characterizes the leakiness of the neural membrane.



### ■ The basic neural circuit

Now imagine that we connect up  $N$  of these model neurons to each other to form a completely connected network, except, like the earlier discrete model, neurons are not connected to themselves, and the conductances are symmetric. In other words, the weight matrix has a zero diagonal, and is symmetric. (We follow Hopfield, and let the output range between  $-1$  and  $1$  for the graded response net, rather than  $0$  and  $1$  as for the discrete net.)

The update rule is given by a set of differential equations over the network. The equations are determined by the three basic laws of electricity: Kirchoff's rule (sum of currents at a junction has to be zero), Ohm's law ( $I=V/R$ ), and that the current across a capacitor is proportional to the rate of change of voltage. As we did earlier with the integrate-and-fire model, we write an expression representing the requirement that the total current into the op amp be zero. With a little rearrangement, we have:

$$C_i \frac{du_i}{dt} = \sum_j T_{ij} V_j - \frac{u_i}{R_i} + I_i$$

$$V_i = g(u_i)$$

The first equation is really just a slightly elaborated version of the "leaky integrate and fire" equation we studied in Lecture 3. We now note that the "current in" ( $s$  in lecture 3) is the sum of the currents from all the inputs.

### ■ Proof of convergence

Now here is where Hopfield's main contribution lies. He defined an energy function as:

$$E = -\frac{1}{2} \sum_{i \neq j} T_{ij} V_i V_j + \sum_i (1/R_i) \int_0^{V_i} g_i^{-1}(V) dV + \sum_i I_i V_i$$

The form of the sigmoidal non-linearity,  $g()$ , was taken to be an inverse tangent function (see below). If we take the derivative of  $E$  with respect to time, then for symmetric  $T$ , we have (applying the chain rule for differentiation of composite functions):

$$dE / dt = - \sum_i \frac{dV_i}{dt} \left( \sum_j T_{ij} V_j - u_i / R_i + I_i \right)$$

Substituting the expression from the first equation for the expression between the brackets, we obtain:

$$dE / dt = - \sum_i C_i (dV_i / dt) (du_i / dt)$$

And now replace  $du_i/dt$  by taking the derivative of the inverse  $g$  function (again using the chain rule):

$$= - \sum_i C_i \frac{dg_i^{-1}(V_i)}{dV_i} \left( \frac{dV_i}{dt} \right)^2$$

Below, there is an exercise in which you can show that the derivative of the inverse  $g$  function is always positive. So that means that the right hand side of the equation can never be positive--energy never increases. Further, we can see that stable points, i.e. where  $dE/dt$  is zero, correspond to attractors in state space. Mathematically, we have:

$$dE / dt \leq 0$$

$$dE / dt = 0 \rightarrow dV_i / dt = 0 \text{ for all } i.$$

So  $E$  is a Liapunov function for the system of differential equations describing the neural system whose neurons have graded responses.

## Simulation of a 2 neuron Hopfield network

### ■ Definitions

We will let the resistances and capacitances all be one, and the current input  $I_i$  be zero. Define the sigmoid function,  $g[]$  and its inverse,  $\text{inverseg}[]$ :

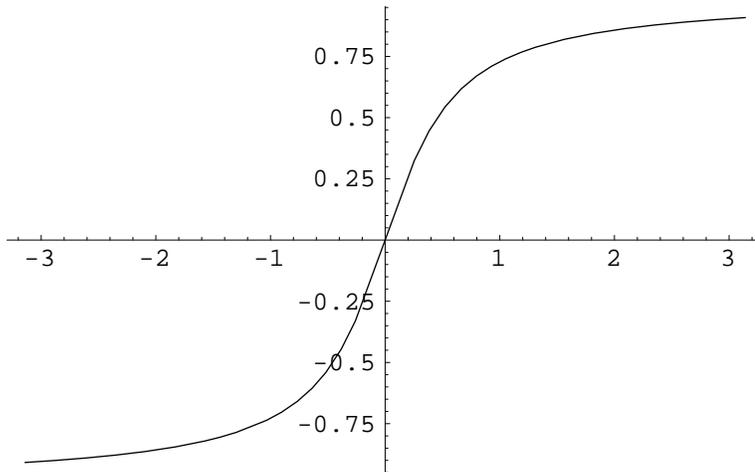
```
In[1]:= a1 := (2/Pi); b1 := Pi 1.4 / 2;
g[x_] := N[a1 ArcTan[b1 x]];
inverseg[x_] := N[(1/b1) Tan[x/a1]];
```

Although it is straightforward to compute the inverse of  $g$  by hand, do it using the `Solve[]` function in *Mathematica*::

```
Solve[a ArcTan[b y]==x,y]
```

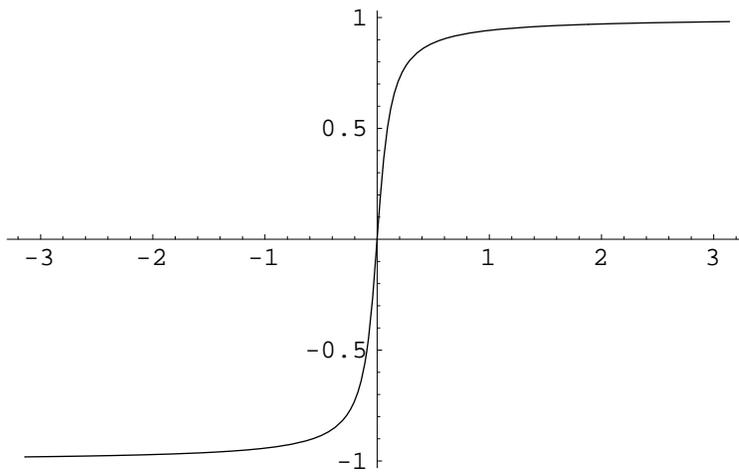
```
{{y -> \frac{\tan(\frac{x}{a})}{b}}}
```

```
Plot[g[x], {x, -Pi, Pi}];
```



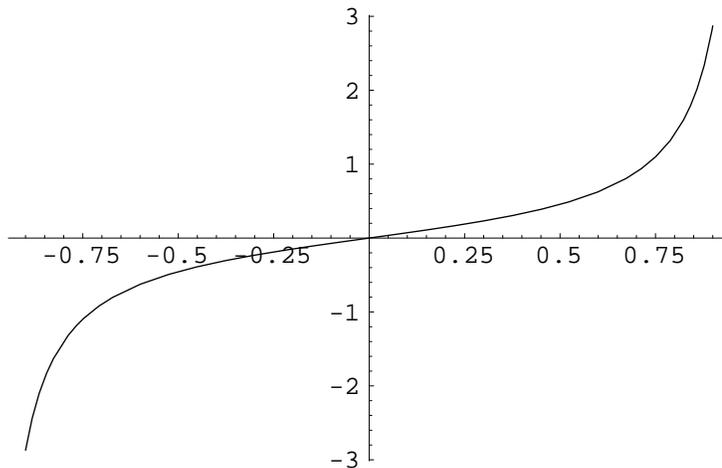
Note how increasing  $b_1$  makes the sigmoid more like a step or threshold function:

```
Plot[N[a1 ArcTan[5 b1 x]], {x, -Pi, Pi}];
```



This is useful, because as  $b_1$  gets large, the continuous net approaches the two-state net.

```
Plot[inverseg[x], {x, -.9, .9}];
```



### Optional Exercise

Calculate the derivative of the inverse of  $g[]$ . Plot it out to demonstrate that it is never zero. Because it is never zero, the rate of change of  $E$  must always be less than or equal to zero.

#### ■ Initialization of starting values

The [initialization](#) section sets the starting output values of the two neurons.

$V = \{0.2, -0.5\}$ , and the internal values  $u = \text{inverseg}[V]$ , the step size,  $dt=0.3$ , and the 2x2 weight matrix,  $Tm$  such that the synaptic weights between neurons are both 1. The synaptic weight between each neuron and itself is zero.

```
In[4]:= dt = 0.3;
Tm = {{0,1},{1,0}};
V = {0.2,-0.5};
u = inverseg[V];
result = {};
```

#### ■ Main Program illustrating descent with discrete updates

Note that because the dynamics of the graded response Hopfield net is expressed in terms of differential equations, the updating is actually continuous (rather than asynchronous and discrete). In order to do a simulation, we will approximate the dynamics with discrete updates of the neurons' activities.

The following function computes the output (just up to the non-linearity) of the  $i$ th neuron for the network with a list of neural values  $uu$ , and a weight matrix  $Tm$ .

```
In[9]:= Hopfield[uu_,i_] := uu[[i]] +
dt ((Tm.g[uu])[[i]] - uu[[i]]);
```

This follows from the above update equations with the capacitances ( $C_i$ ) and resistances ( $R_i$ ) set to 1.

$$C_i \frac{du_i}{dt} = \sum_j T_{ij} V_j - \frac{u_i}{R_i} + I_i$$

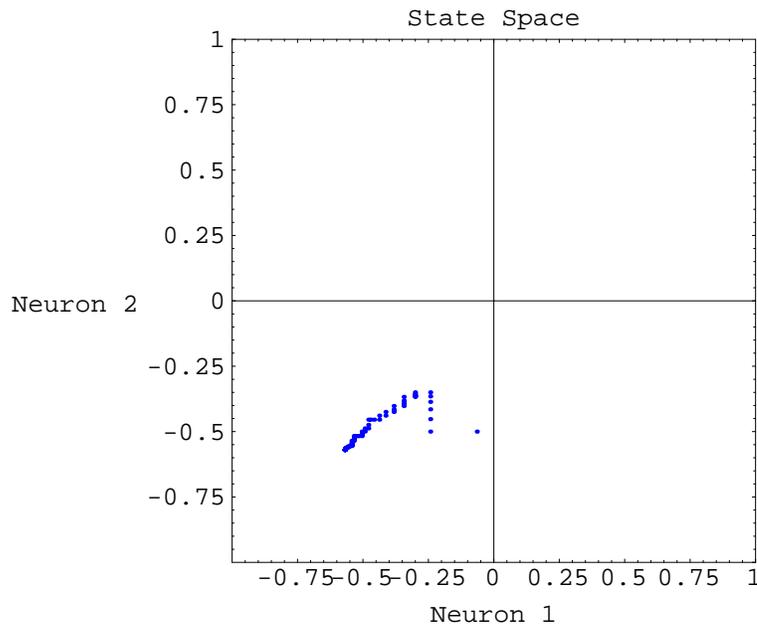
$$V_i = g(u_i)$$

Let's accumulate some results for a series of 80 iterations. Then we will plot the pairs of activities of the two neurons over the 80 iterations. The next line randomly samples the two neurons for asynchronous updating.

```
In[10]:= result =
Table[{k=Random[Integer,{1,2}],u[[k]] =
Hopfield[u,k],u}, {80}];
```

```
In[12]:= result = Transpose[result][[3]];
```

```
In[16]:= gresults = ListPlot[g[result],
PlotJoined->False,AxesOrigin->{0,0},
PlotRange->{{-1,1},{-1,1}},FrameLabel->{"Neuron 1","Neuron 2"},
Frame->True, AspectRatio->1,RotateLabel->False,PlotLabel->"State
Space",
Ticks->None, PlotStyle->{RGBColor[0,0,1]}];
```



## ■ Energy landscape

Now make a contour plot of the **energy landscape**. We will need the integral of the inverse of the  $g$  function, call it **inig[]**. We use the **Integrate[]** function to find it. Then define **energy[x\_,y\_]** and use **ContourPlot[]** to map out the energy function.

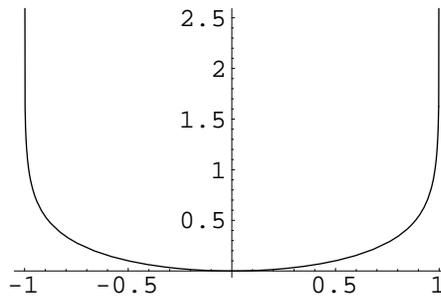
```
Integrate[(1/b) Tan[x1/a], x1] /. x1 -> x
```

$$-\left(\frac{a \operatorname{Log}\left[\cos\left[\frac{x}{a}\right]\right]}{b}\right)$$

Change the above output to input form:

$$-\left(\frac{a \operatorname{Log}\left[\cos\left[\frac{x}{a}\right]\right]}{b}\right)$$

```
inig[x_] := -N[(a1*Log[Cos[x/a1]])/b1];
Plot[inig[x], {x, -1, 1}];
```



We write a function for the above expression for energy:

$$E = -\frac{1}{2} \sum_{i \neq j} T_{ij} V_i V_j + \sum_i (1/R_i) \int_0^{V_i} g_i^{-1}(V) dV + \sum_i I_i V_i$$

```
energy[Vv_] := -0.5 (Tm.Vv).Vv +
  Sum[inig[Vv][[i]], {i, Length[Vv]}];
```

And then define a contour plot of the energy over state space:

```

gcontour = ContourPlot[energy[{x,y}],{x,-1,1},
  {y,-1,1},AxesOrigin->{0,0}, ContourShading->False,
  PlotRange->{-.1,.8}, Contours->32,
  PlotPoints->30,FrameLabel->{"Neuron 2","Neuron 2"},
  Frame->True, AspectRatio->1,RotateLabel->False,PlotLabel->"Energy over
  State Space",
  DisplayFunction->Identity];

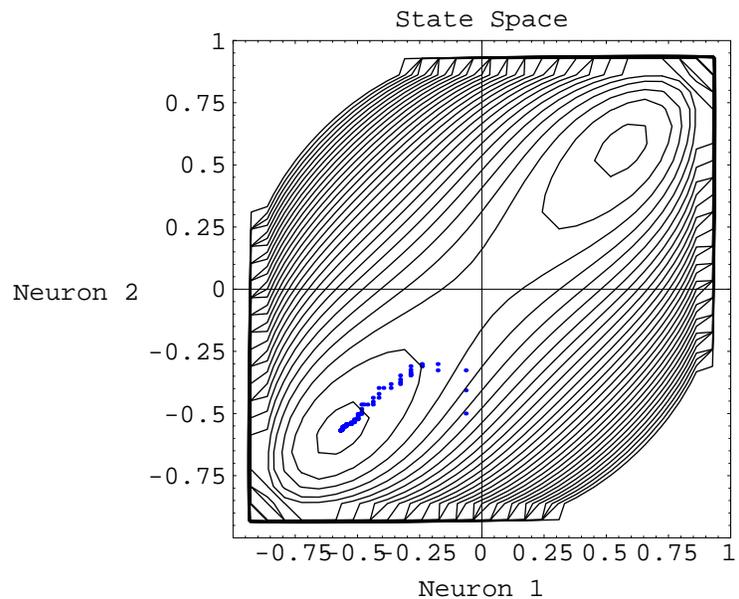
```

Now let's superimpose the trajectory of the state vector on the energy contour plot:

```

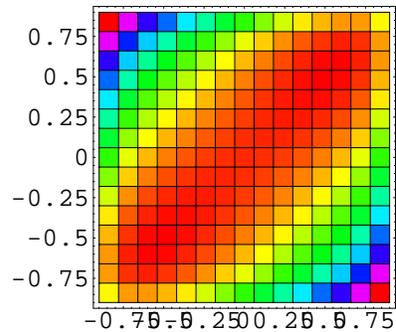
Show[results, gcontour,
  DisplayFunction-> $DisplayFunction];

```



### ■ Cute color plot of energy landscape

```
DensityPlot[energy[{x, y}], {x, -0.9, 0.9}, {y, -0.9, 0.9},  
ColorFunction->Hue];
```



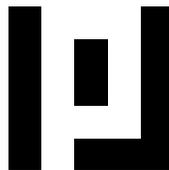
---

## ■ Autoassociative memory example

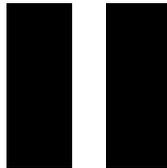
**Sculpting the energy for memory recall using a Hebbian learning rule. TIP example.**

### ■ The stimuli

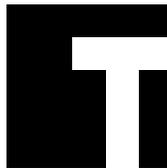
We will store the letters P, I, and T in a 25x25 element weight matrix. For maximum separation, we will put them near the corners of a hypercube.



```
Pmatrix = {
  {0, 1, 0, 0, 0},
  {0, 1, 1, 1, 0},
  {0, 1, 0, 1, 0},
  {0, 1, 0, 1, 0},
  {0, 1, 1, 1, 0}};
```



```
Imatrix = {
  {0, 0, 1, 0, 0},
  {0, 0, 1, 0, 0},
  {0, 0, 1, 0, 0},
  {0, 0, 1, 0, 0},
  {0, 0, 1, 0, 0}};
```



```
Tmatrix = {
  {0, 0, 0, 1, 0},
  {0, 0, 0, 1, 0},
  {0, 0, 0, 1, 0},
  {0, 0, 1, 1, 1},
  {0, 0, 0, 0, 0}};
```

```
width = Length[Pmatrix];
one = Table[1, {i,width}, {j,width}];
Pv = Flatten[2 Pmatrix - one];
Iv = Flatten[2 Imatrix - one];
Tv = Flatten[2 Tmatrix - one];
```

Note that the patterns are not normal, or orthogonal:

```
{Tv.Iv, Tv.Pv, Pv.Iv, Tv.Tv, Iv.Iv, Pv.Pv}
```

```
{7, 3, 1, 25, 25, 25}
```

## ■ Learning

Make sure that you've defined the sigmoidal non-linearity and its inverse above. The items will be stored in the connection weight matrix using the outer product form of the Hebbian learning rule:

```
Weights =
Outer[Times, Tv, Tv] + Outer[Times, Iv, Iv] +
Outer[Times, Pv, Pv];
```

Note that in order to satisfy the requirements for the convergence theorem, we should enforce the diagonal elements to be zero. (Is this necessary for the network to converge?)

```
For[i = 1, i <= Length[Weights], i++, Weights[[i, i]] = 0];
```

## Hopfield graded response neurons applied to reconstructing the letters T,I,P. The non-linear network makes decisions

In this section, we will compare the Hopfield network's response to the linear associator. First, we will show that the Hopfield net has sensible hallucinations--a random input can produce interpretable stable states. Further, remember a major problem with the linear associator is that it doesn't make proper discrete decisions when the input is a superposition of learned patterns. The Hopfield net makes a decision.

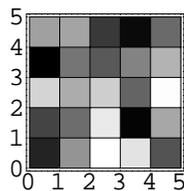
### ■ Sensible hallucinations to a noisy input

We will set up a version of the graded response Hopfield net with synchronous updating. (In a homework exercise, you will compare asynchronous and synchronous updating.)

```
dt = 0.03;
Hopfield[uu_] := uu + dt (Weights.g[uu] - uu);
```

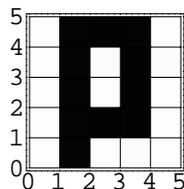
Let's first perform a kind of "Rorschach blob test" on our network (but without the usual symmetry to the input pattern). We will give as input uncorrelated uniformly distributed noise and find out what the network sees:

```
forgettingT = Table[2 Random[] - 1, {i, 1, Length[Tv]}];
ListDensityPlot[Partition[forgettingT, width]];
```



```
rememberingT = Nest[Hopfield, forgettingT, 30];
```

```
ListDensityPlot[Partition[g[rememberingT], width],
  PlotRange->{-1, 1}];
```

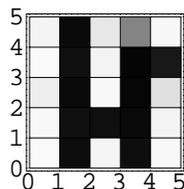


In this case, the random input produces sensible output--the network saw the letter **P**. Do you think you will always get **P** as the network's hallucination?

### ■ Comparison with linear associator

What is the linear associator output for this input? We will follow the linear output with the squashing function, to push the results towards the hypercube corners:

```
ListDensityPlot[Partition[g[Weights.forgettingT],
  width], PlotRange->{-1, 1}];
```

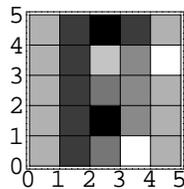


The noisy input didn't give a meaningful output this time. You can try other noise samples. Because of superposition it will not, on average, give the meaningful discrete memories that the Hopfield net does. However, sometimes the linear matrix memory will produce meaningful outputs and sometimes the Hopfield net will produce nonsense depending on how the energy landscape has been sculpted. If the "pits" are arranged badly, one can introduce valleys in the energy landscape that will produce spurious results.

### ■ Response to superimposed inputs

Let us look at the problem of superposition by providing an input which is a linear combination of the learned patterns. Let's take a weighted sum, and then start the state vector fairly close to the origin of the hypercube:

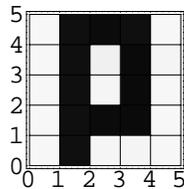
```
forgettingT = .1 (0.2 Tv - 0.15 Iv - 0.3 Pv);
ListDensityPlot[Partition[forgettingT,width]];
```



Now let's see what the Hopfield algorithm produces. We will start with a smaller step size. It sometimes takes a little care to make sure the descent begins with small enough steps. If the steps are too big, the network can converge to the same answers one sees with the linear associator followed by the non-linear sigmoid.

```
dt = 0.01;
Hopfield[uu_] := uu + dt (Weights.g[uu] - uu);
rememberingT = Nest[Hopfield,forgettingT,30];
```

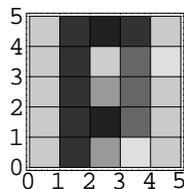
```
ListDensityPlot[Partition[g[rememberingT],width],
PlotRange->{-1,1}];
```



### ■ Comparison with linear associator

The linear associator followed by the squashing function gives:

```
ListDensityPlot[Partition[g[Weights.forgettingT],
width], PlotRange->{-1,1}];
```



For the two-state Hopfield network with Hebbian storage rule, the stored vectors are stable states of the system. For the graded response network, the stable states are near the stored vectors. If one tries to store the items near the corners of the

hypercube they are well separated, and the recall rule tends to drive the state vector into these corners, so the recalled item is close to what was stored.

### ■ Using the formula for energy

The energy function can be expressed in terms of the integral of the inverse of  $g$ , and the product of the weight matrix times the state vector, times the state vector again:

```
inig[x_] := -N[(a1*Log[Cos[x/a1]])/b1];
energy[Vv_] := -0.5 ((Weights.Vv).Vv) +
  Sum[inig[Vv][[i]], {i,Length[Vv]}];
```

```
energy[.99 Pv]
energy[g[forgettingT]]
energy[g[rememberingT]]
```

```
-244.367
```

```
-0.528994
```

```
-219.84
```

One of the virtues of knowing the energy or Liapunov function for a dynamical system, is being able to check how well you are doing. We might expect that if we ran the algorithm a little bit longer, we could move the state vector to an even lower energy state if -219 was not the minimum. The fact that the energy of .99 Pv is -244 suggests that we are not to the minimum yet.

```
dt = 0.01;
Hopfield[uu_] := uu + dt (Weights.g[uu] - uu);
rememberingT = Nest[Hopfield,forgettingT,60];
```

```
energy[g[rememberingT]]
```

```
-239.471
```

---

## References

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.

Hentschel, H. G. E., & Barlow, H. B. (1991). Minimum entropy coding with Hopfield networks. *Network*, 2, 135-148.

Mead, C. (1989). *Analog VLSI and Neural Systems*. Reading, Massachusetts: Addison-Wesley.

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