Introduction to Neural Networks
U. Minn. Psy 5038

"Energy" and attractor networks
Graded response Hopfield net

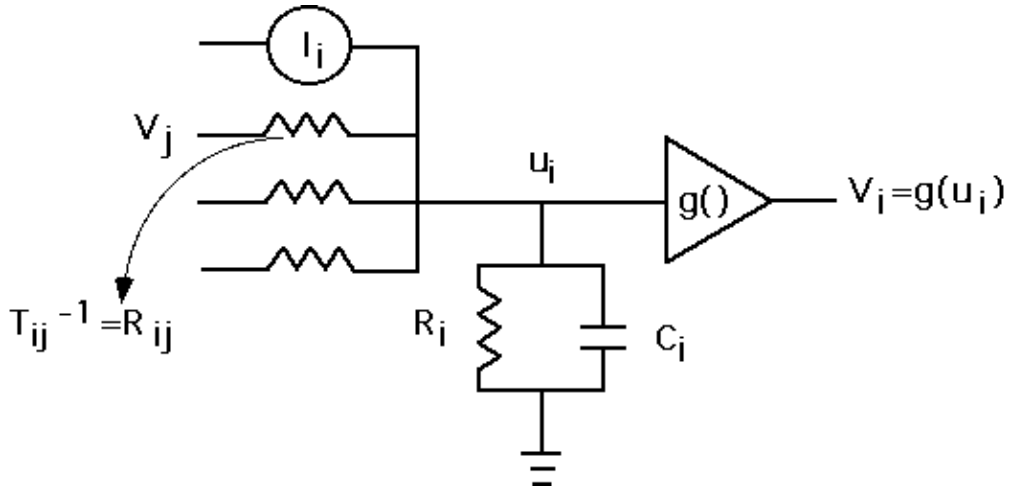
Graded response Hopfield network

■ 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). The question was whether this more realistic model also showed well-behaved convergence properties (Cohen and Grossberg published a paper in 1983 showing conditions for network convergence).

Earlier we derived an expression for the rate of firing for the "leaky 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 capacitance C_i and membrane resistance R_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. 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 ($T_{ii}=0$), and is symmetric ($T_{ij}=T_{ji}$). (We follow Hopfield, and let the output range between -1 and 1 for the graded response net, rather than 0 and 1as for the discrete net in the previous lecture.)

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 (I=Cdu/dt). Resistance is the reciprocal of conductance (T=1/R). 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:

$$C_i \frac{du_i}{dt} = \sum_j T_{ij} V_j - \frac{u_i}{R_i} + I_i \tag{1}$$

$$V_i = g(u_i)$$

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. All parameters (C_i,T_{ij},R_i,I_i,g) are fixed, and we want to know how the state vector V_i changes with time. We could imagine that the state vector could have almost any trajectory, and wander arbitrarily around state space--but it doesn't. In fact, just as for the discrete case, the continuous Hopfield network converges to stable attractors. Suppose that at some time t, we know $V_i(t)$ for all units (i=1 to N). Then Hopfield proved that the state vector will migrate to points in state space whose values are constant: $V_i \to V_i^s$. In other words, to state space points where $dV_i/dt = 0$. This is a steady-state solution to the above equations. This is an important result because it says that the network can be used to store memories.

To prove this, Hopfield defined an energy function as:

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

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 (1) 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.

And because capacitance and $(dV_i/dt)^2$ are also positive, 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, **inverseg[]**:

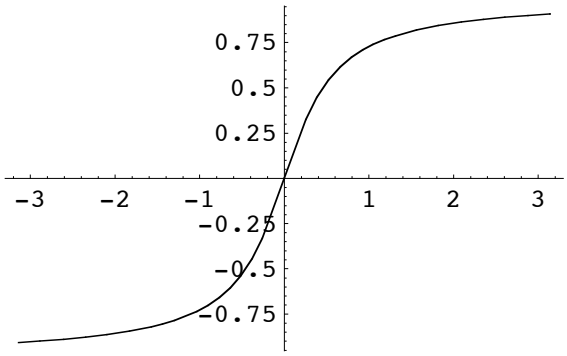
In[36]:=
```
a1 := (2/Pi); b1 := Pi 1.4 / 2;
g[x_] := N[a1 ArcTan[b1 x]];
inverseg[x_] := N[(1/b1) Tan[x/a1]];
```

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

In[39]:=
```
Solve[a ArcTan[b y]==x,y]
```

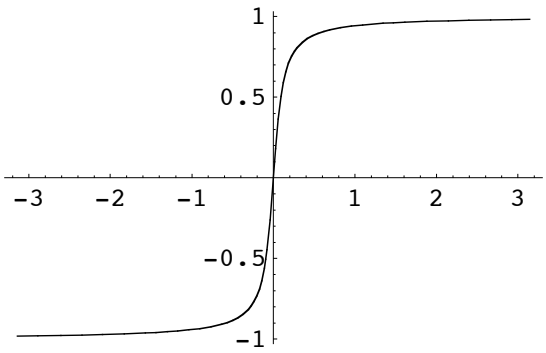
Out[39]=
$$\left\{\left\{y \to \frac{\tan\left(\frac{x}{a}\right)}{b}\right\}\right\}$$

In[40]:=
```
Plot[g[x],{x,-Pi,Pi}];
```



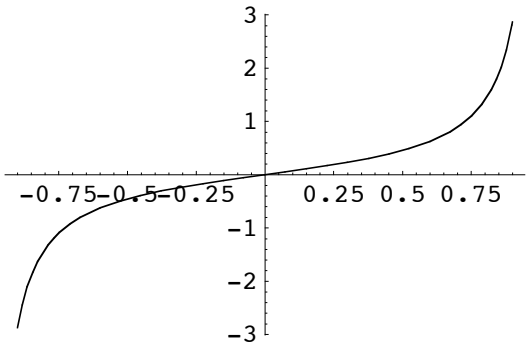
Note how increasing b1 makes the sigmoid more like a step or threshold function:

In[45]:=
```
Plot[N[a1 ArcTan[5 b1 x]], {x, -Pi, Pi}];
```



This is useful, because as **b1** gets large, the we'd expect the continuous net to behave like the discrete (two-state) net.

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



Exercise

Use **D[ ]** to 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.

In[51]:=
```
Plot[inverseg'[x], {x, -1, 1}];
```

■ 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 = 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[228]:=
```
dt = 0.01;
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 digital computer 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 ith neuron for the network with a list of neural values **uu**, and a weight matrix **Tm**.

In[233]:=
```
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.

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

$$V_i = g(u_i)$$

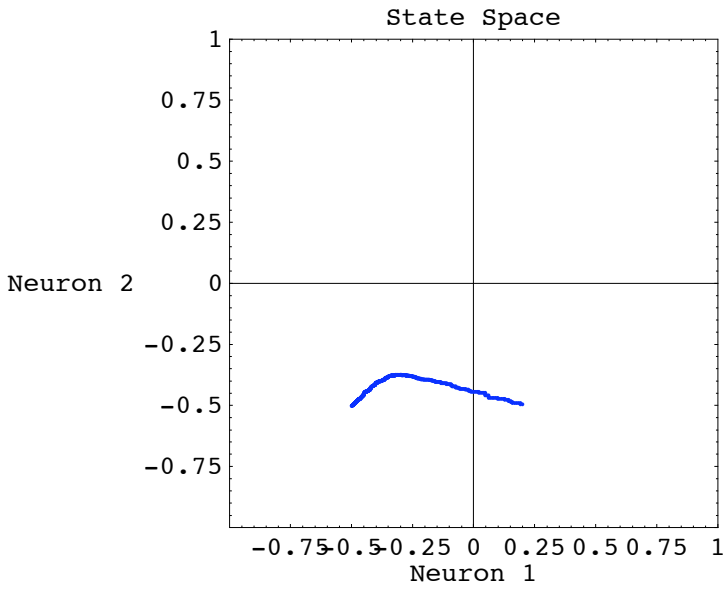
Let's accumulate some results for a series of 800 iterations. Then we will plot the pairs of activities of the two neurons over the 80 iterations. The next line randomly samples the two neurons for asynchronous updating.

In[234]:=
```
result =
Table[{k=Random[Integer,{1,2}],u[[k ]] =
    Hopfield[u,k],u}, {800}];
```

In[235]:=
```
result = Transpose[result][[3]];
```

To visualize the trajectory of the state vector, we can use ListPlot:

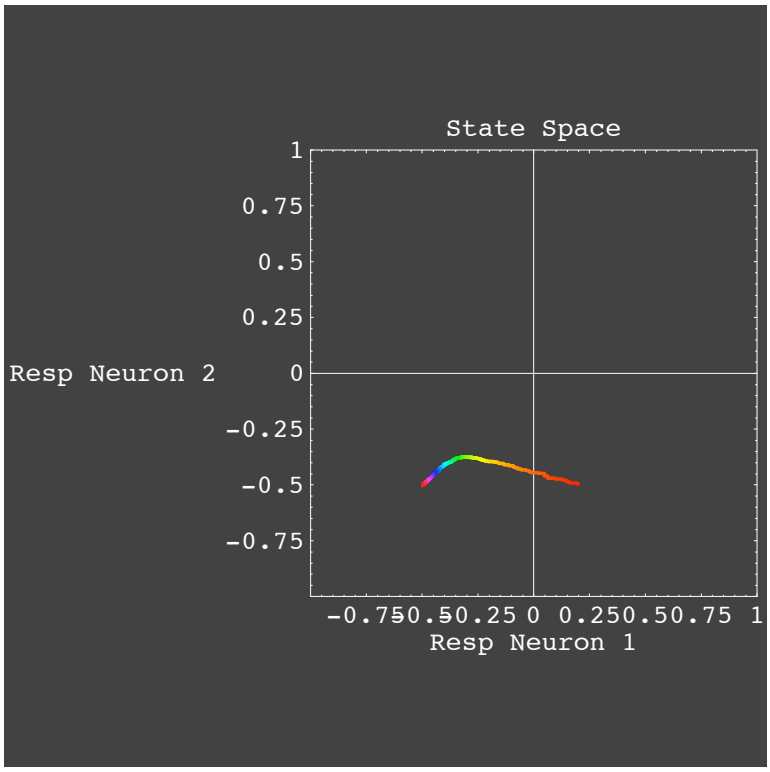
In[236]:=
```
gresults = ListPlot[g[result],
    PlotJoined->False,AxesOrigin->{0,0},
    PlotRange->{{-1,1},{-1,1}},FrameLabel->{"Neuron 1","Neuron 2"},
    Frame->True, AspectRatio->1,RotateLabel->False,PlotLabel->"State
Space",
    Ticks->None, PlotStyle->{RGBColor[0,0,1]}];
```



We can visualize the time evolution by color coding state vector points according to Hue[time], where H starts off red, and becomes "cooler" with time, ending up as violet (the familiar rainbow sequence: ROYGBIV).

In[237]:=
```
gcolortraj =
  Graphics[
   ({Hue[#1/Length[g[result]]],
      Point[{g[result][[#1]][[1]], g[result][[#1]][[2]]}]} &) /@
    Range[1, Length[g[result]]], AspectRatio → Automatic];
```

In[238]:=
```
Show[gcolortraj, AxesOrigin → {0, 0}, Axes → True,
  PlotRange → {{-1, 1}, {-1, 1}},
  FrameLabel → {"Resp Neuron 1", "Resp Neuron 2"}, Frame → True,
  AspectRatio → 1, RotateLabel → False, PlotLabel → "State Space",
  Ticks → None, Background → GrayLevel[0.2]];
```



■ 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.

In[239]:=
```
Integrate[(1/b) Tan[x1/a],x1]/.x1->x
```

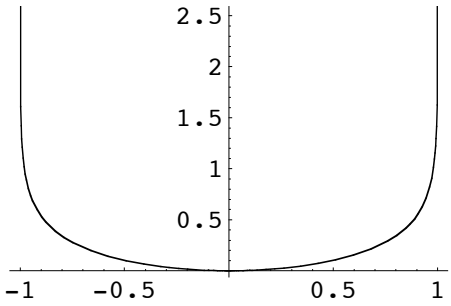
Out[239]=
$$-\frac{a \log\left(\cos\left(\frac{x}{a}\right)\right)}{b}$$

Change the above output to input form:

In[240]:=
```
  a Log[Cos[x/a]]
-(---------------)
      b
```

Out[240]=
$$-\frac{a \log\left(\cos\left(\frac{x}{a}\right)\right)}{b}$$

In[241]:=
```
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:

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

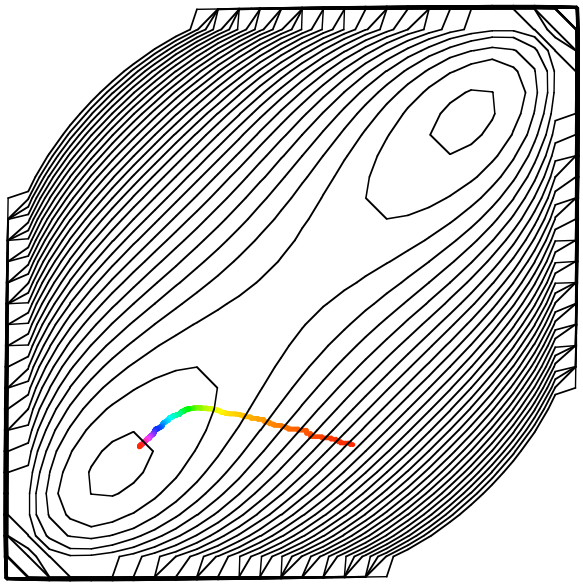
In[243]:=
```
energy[Vv_] := -0.5 (Tm.Vv).Vv + Sum[inig[Vv][[i]], {i,Length[Vv]}];
```

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

In[244]:=
```
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:

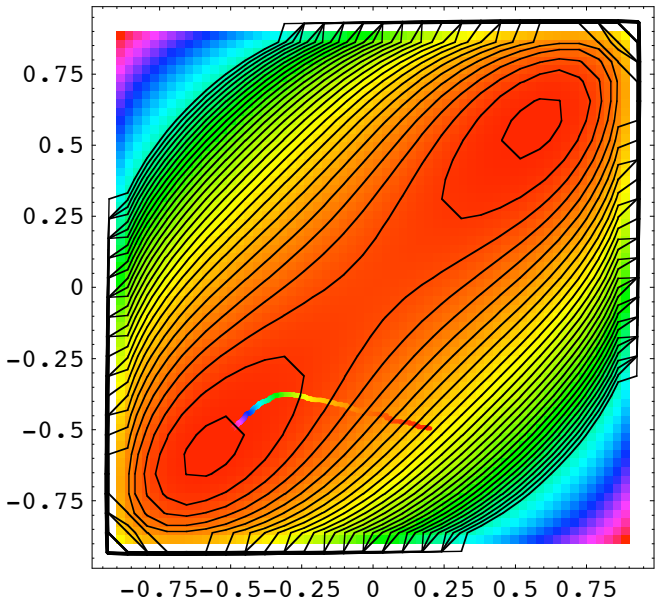
In[245]:=
```
Show[gcolortraj, gcontour,
    DisplayFunction-> $DisplayFunction];
```



■ Color plot of energy landscape

In[246]:=
```
genergy =
DensityPlot[energy[{x,y}],{x,-.9,.9},{y,-.9,.9},PlotRange->All,
Mesh->False,PlotPoints->60,ColorFunction->Hue,DisplayFunction→Identity];
```

In[247]:=
```
Show[genergy,gcolortraj, gcontour,
    DisplayFunction-> $DisplayFunction];
```



Has the network reached a stable state? If not, how many iterations does it need to converge?

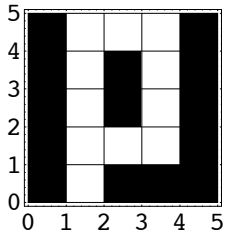
Autoassociative memory example

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

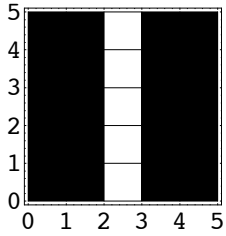
■ The stimuli

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.

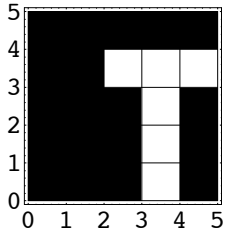
In[123]:=
```
ListDensityPlot[
  Pmatrix = {{0, 1, 0, 0, 0}, {0, 1, 1, 1, 0}, {0, 1, 0, 1, 0},
    {0, 1, 0, 1, 0}, {0, 1, 1, 1, 0}}];
```



In[124]:=
```
ListDensityPlot[
  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}}];
```



In[125]:=
```
ListDensityPlot[
  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}}];
```



In[126]:=
```
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:

In[131]:=
```
{Tv.Iv,Tv.Pv,Pv.Iv, Tv.Tv, Iv.Iv, Pv.Pv}
```

Out[131]=
```
{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:

In[132]:=
```
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?)

In[133]:=
```
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.

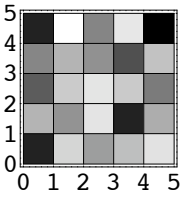
■ Sensible hallucinations to a noisy input

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.)

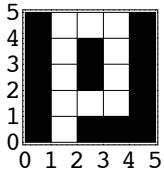
In[134]:=
```
dt = 0.03;
Hopfield[uu_] := uu + dt (Weights.g[uu] - uu);
```

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:

In[192]:=
```
forgettingT = Table[2 Random[] - 1,{i,1,Length[Tv]}];
ListDensityPlot[Partition[forgettingT,width]];
```



In[194]:=
```
rememberingT = Nest[Hopfield,forgettingT,30];
ListDensityPlot[Partition[g[rememberingT],width],
                PlotRange->{-1,1}];
```



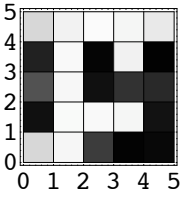
In this case, the random input produces sensible output--the network saw the letter **P**.

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:

In[196]:=
```
ListDensityPlot[Partition[g[Weights.forgettingT],
    width], PlotRange->{-1,1}];
```

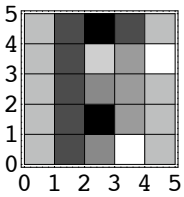


The noisy input didn't give a meaningful output this time. You can try other noise samples. Because of superposition it will typically not give the meaningful discrete memories that the Hopfield net does. However, sometimes the linear matrix memory will produce meaningful outputs and sometimes the Hopfield 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:

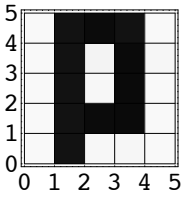
In[197]:=
```
forgettingT = .1 (0.2 Tv - 0.15 Iv - 0.3 Pv);
ListDensityPlot[Partition[forgettingT,width]];
```



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.

In[199]:=
```
dt = 0.01;
Hopfield[uu_] := uu + dt (Weights.g[uu] - uu);
rememberingT = Nest[Hopfield,forgettingT,30];
```

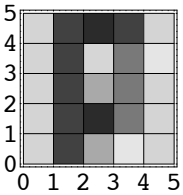
In[202]:=
```
ListDensityPlot[Partition[g[rememberingT],width],
                PlotRange->{-1,1}];
```



■ Comparison with linear associator

The linear associator followed by the squashing function gives:

In[203]:=
```
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:

In[204]:=
```
inig[x_] := -N[(a1*Log[Cos[x/a1]])/b1];
energy[Vv_] := -0.5 ((Weights.Vv).Vv) +
            Sum[inig[Vv][[i]], {i,Length[Vv]}];
```

In[206]:=
```
energy[.99 Pv]
energy[g[forgettingT]]
energy[g[rememberingT]]
```

Out[206]= −244.367

Out[207]= −0.528994

Out[208]= −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.

In[209]:=
```
dt = 0.01;
Hopfield[uu_] := uu + dt (Weights.g[uu] - uu);
rememberingT = Nest[Hopfield,forgettingT,600];
```

In[212]:=
```
energy[g[rememberingT]]
```

Out[212]=
```
-244.693
```

Conclusions

Although Hopfield networks have had a considerable impact on the field, they have had limited practical application. There are more efficient ways to store memories, to do error correction, and to do constraint satisfaction or optimization. As models of neural systems, the theoretical simplifications severely limit their generality. Although there is current theoretical work on Hopfield networks, it has involved topics such as memory capacity and efficient learning, and relaxations of the strict theoretical presumptions of the original model. The field has moved on to more general probablistic theories of networks that include older models as special cases. In the next few lectures we will see some of the history and movement in this direction.

References

Bartlett, M. S., & Sejnowski, T. J. (1998). Learning viewpoint-invariant face representations from visual experience in an attractor network. *Network, 9*(3), 399-417.

Chengxiang, Z., Dasgupta, C., & Singh, M. P. (2000). Retrieval properties of a Hopfield model with random asymmetric interactions. *Neural Comput, 12*(4), 865-880.

Cohen, M. A., & Grossberg, S. (1983). Absolute Stability of Global Pattern Formation and Parallel Memory Storage by Competitive Neural Networks. IEEE Transactions SMC-13, 815-826.

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. Vol. Lecture Notes Volume 1). Reading, MA: Addison-Wesley Publishing Company.

Hopfield, J. J. (1982). Neural networks and physical systems with emergent collective computational abilities. *Proc Natl Acad Sci U S A, 79*(8), 2554-2558.

Computational properties of use of biological organisms or to the construction of computers can emerge as collective properties of systems having a large number of simple equivalent components (or neurons). The physical meaning of content-addressable memory is described by an appropriate phase space flow of the state of a system. A model of such a system is given, based on aspects of neurobiology but readily adapted to integrated circuits. The collective properties of this model produce a content-addressable memory which correctly yields an entire memory from any subpart of sufficient size. The algorithm for the time evolution of the state of the system is based on asynchronous parallel processing. Additional emergent collective properties include some capacity for generalization, familiarity recognition, categorization, error correction, and time sequence retention. The collective properties are only weakly sensitive to details of the modeling or the failure of individual devices.

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.

A model for a large network of "neurons" with a graded response (or sigmoid input-output relation) is studied. This deterministic system has collective properties in very close correspondence with the earlier stochastic model based on McCulloch - Pitts neurons. The content- addressable memory and other emergent collective properties of the original model also are present in the graded response model. The idea that such collective properties are used in biological systems is given added credence by the continued presence of such properties for more nearly biological "neurons." Collective analog electrical circuits of the kind described will certainly function. The collective states of the two models have a simple correspondence. The original model will continue to be useful for simulations, because its connection to graded response systems is established. Equations that include the effect of action potentials in the graded response system are also developed.

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.

Saul, L. K., & Jordan, M. I. (2000). Attractor dynamics in feedforward neural networks. *Neural Comput, 12*(6), 1313-1335.