Introduction to Neural Networks

"Energy" and attractor networks
Hopfield networks

---

# Introduction

## Last time

### Supervised learning. Introduced the idea of a "cost" function over weight space

Regression and learning in linear neural networks. The cost was the sum of squared differences between the networks predictions of the correct answers and the correct answers.

The motivation was to derive a "learning rule" that adjusts (synaptic) weights to minimize the discrepancy between predictions and answers.

Last time we showed 4 different ways to find the generating parameters {a,b} = {2, 3} for data with the following generative process:

```
rsurface[a_, b_] := N[Table[{x1 = 1 RandomReal[],
      x2 = 1 RandomReal[], a x1 + b x2 + 0.5 RandomReal[] - 0.25}, {120}], 2];
data = rsurface[2, 3];
Outdata = data[[All, 3]];
Indata = data[[All, 1 ;; 2]];
```

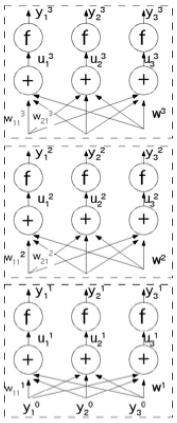
The last method--Widrow-Hoff--was biologically plausible in that it could be interpreted as synaptic weight adjustment.

Linear regression is so common that *Mathematica* has added the following function to find the least squares parameters directly:

```
LeastSquares[Indata, Outdata]
```

```
{1.92373, 3.03521}
```

### Error backpropagation or "backprop"



The problem is: how to assign the weights through supervised learning? We assume the following notation:

$$\{x^p, t^p\},\ \ y^0 = x^p,\ \ \text{training pairs p} = 1,...,M$$

For a given input $\{y^0 = x^p\}$, we feed forward the information to the last layer (layer L) to produce an output $\{\mathbf{y}=\boldsymbol{y}^L\}$. We compare the output to the target value supplied by the "teacher" $\{t = t^p\}$, and compute the error as the sum of squared differences:

$$E(\{w_{ij}^{\lambda}\}) = \sum_{k=1}^{N} (y_k(y^0; w_{ij}^{\lambda}) - t_k)^2$$

where the sum is over all N output units. (For simplicity, we left out the superscript p in $t_k^p$. The subscript k in $t_k^p$ means the element of the corresponding vector of activity $t^p$.) $\lambda$ indexes the weight layers going from $\lambda$=1 to L. Note that the y's at any point after the input depend on the u's (the weighted sum before the non-linearity), each of which in turn depends on all the $w_{ij}^{\lambda}$s before it.

The trick is to find out how to assign credit (and blame) for the error to each of the weights. Gradient descent provides the answer. Adjust the weights such that:

$$\text{new } \mathbf{w}_{ij}^{\lambda} = \text{previous } \mathbf{w}_{ij}^{\lambda} + \Delta \mathbf{w}_{ij}^{\lambda}$$

where

$$\Delta w_{ij}^{\lambda} = -\eta \frac{\partial E}{\partial w_{ij}^{\lambda}}$$

Again, this formula means that if we calculate the gradient $\nabla E = (\frac{\partial E}{\partial w_{11}^{\lambda}} ... \frac{\partial E}{\partial w_{ij}^{\lambda}} ...)$, its negative direction points in the direction of steepest descent. Thus if we update the weight vector to point in this direction, then at the next step we should in general have a lower value of E. I.e.

$$E(\text{new } w_{ij}^{\lambda}) < E(\text{previous } w_{ij}^{\lambda}).$$

The appendix in the previous lecture shows how to calculate a weight adjust with just one layer of weights. This case is related to the area of Generalized Linear Regression (not to be confused with the General Linear Model!).

The real work is figuring out the rule to update the weights at each layer of the network. We won't derive

that here. Instead we'll summarize how the backprop algorithm works. The derivation is primarily difficult because of the need to keep track of multiple indexes while doing the differentations. There are a number of derivations and illustrations on the web that you might find useful. See for example, http://-galaxy.agh.edu.pl/~vlsi/AI/backp_t_en/backprop.html. Also, Andrew Ng has several excellent coursera videos on error backpropagation.

## Summary of backprop algorithm

1. Initialize the weights to small random values
2. Pick a pattern from the input/output collection, say the $p^{th}$ pattern: $\{x^p, t^p\}$. Run the input vector, $y_0 = x^p$ , feedforward through the network. This will generate a set of values $u_i^L$ and $y_i^L$ in all the nodes of the network. Keep in mind that $u_i^L$ is the linear weighted sum of its inputs arriving from layer L-1:

$u_i^L = \sum_{j=1}^{3} w_{ij}^L y_j^{L-1}$.

And $y_i^L = f(u_i^L)$, where f() is the logistic form of the sigmoidal non-linearity

Calculate a delta term (analogous to the Widrow-Hoff rule) for the output layer L:

$$\partial_i^L = \left(t_i^p - y_i^L\left(x^p\right)\right) f'\left(u_i^L\right)$$

where f '() is the derivative of the logistic function. Note why it is important to have an expression for the *derivative* of the function f(). The derivative has a particularly nice form when $f(u) = 1/(1+e^{-u})$. (see Appendix in the previous lecture to see how the logistic function provides a smooth, differentiable non-linearity)

3. Propagate the errors back through the layers:

$$\partial_i^\lambda = f'(u_i^\lambda) \sum_{k=1}^{N} \partial_k^{\lambda+1} w_{ki}^{\lambda+1} \qquad \lambda = \mathrm{L}-1,\ldots,1$$

...the error back propagation or "back prop" part.

4. Calculate weight adjustments (analogous to the outerproduct part of the Widrow-Hoff) and update using:

$$\Delta w_{ij}^\lambda = \eta \partial_i^\lambda y_j^{\lambda-1}$$

5. Repeat steps 2 to 4 until convergence.

## Backprop simulation example: XOR

With appropriate weights, 2 weight layers with 3 hidden units can solve the XOR problem. But this is still a tough problem to learn, mainly because it requires that two very different inputs map to the same output. See the supplementary material for a Mathematica demo that learns the weights for solving the XOR problem.

# Today

Today we will see conditions under which our generic , non-linear neural network can recall from stored memories, make decisions, and self-correct. We will again use the idea of descending the landscape of a function, but instead the function to be minimized will be over state space rather than weight space. These functions are called “objective functions”,“energy functions”, Lyapunov functions. They are used to understand the dynamics of the changes in neural activity.

The motivation goes back to the superposition problem in linear network recall of letters T, I, P:

Can we find an alternative mechanism for doing recall that is more selective? One that avoids the problem of interference?

# Discrete two-state Hopfield network for recall

## Background

We’ve seen how learning can be characterized by gradient descent on an error function e(**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*, **V**(t), 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 follow 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.

**Neural dynamics can be like traveling down a mountain landscape**
Recall that a network’s “state vector”, **V**(t), is simply the vector whose elements are the activities of all the neurons in the network at a specific time t. One can define a scalar 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. But suppose we could specify E(**V**) in such a way that small values of E are "good", and large values are "bad". In other words low values of E(**V**) tell us when the network is getting close to a right answer--as in the search game "you're getting warmer". But because getting closer is getting “cooler” energy-wise,  the sign in the energy metaphor reversed.

We’ve already learned about gradient descent in weight space. This suggests that if one had an energy function, one could compute the time derivative of E(**V**) and set it equal to the gradient of E with respect to **V**. Just like we did with costs over weights, but now “cost” (or energy) over neural activity. 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, and thus we'd go from "bad" to "better" to "good" solutions.

But would such a gradient derived rule correspond to any reasonable neural network model?

In two influential papers, John Hopfield approached the problem in the opposite direction. He started off with a model of neural network connectivity using 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 the theory showed that under certain conditions these networks converge to a stable state, and thus have well-defined properties for computation.

**Why is an "energy" interpretation of neural dynamics useful?**

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 sometimes called "*attractors*".

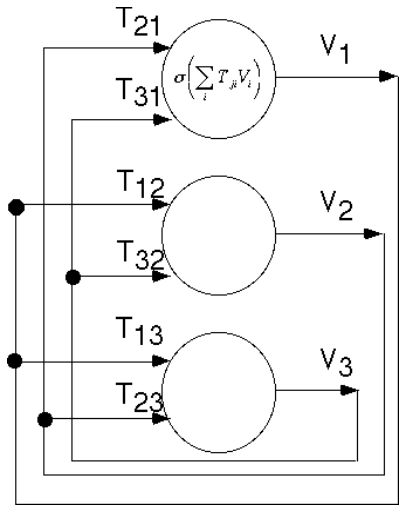
Hopfield nets can be used to solve various classes of problems. We've already mentioned memory and recall. Hopfield nets can be used for error correction, as content addressable memories (as in linear autoassociative recall in the reconstruction of missing information), and *constraint satisfaction*.

Consider the latter case, of constraint satisfaction (also called optimization). 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.

The supplementary material gives 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 (or perceptron units), 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 network, or
--illegal or noisy 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...

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

**Discrete (binary) model neuron revisited.** 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:

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

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 like the weights 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--i.e. whether there is an intuitive advantage to associating parameters in the problem with biases rather than weights.)

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:

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

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.

In[269]:=
```
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}];
```

In[273]:=
```
V1m // MatrixForm
```

Out[273]//MatrixForm=

$$\begin{pmatrix} V1[1] \\ V1[2] \\ V1[3] \\ V1[4] \\ V1[5] \end{pmatrix}$$

In[274]:=
```
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 V2m be the new vector of activities. V2m is the same as V1m except for unit 2, which gets changed from V1[2] to V2[2]:

In[276]:=
```
V2m = V1m;
V2m[[2]] = V2[2];
delataenergy = - (1/2) (Tm.V2m).V2m - (-(1/2) (Tm.V1m).V1m);
```

In[279]:=
```
Simplify[delataenergy]
```

Out[279]=
```
(T[1, 2] V1[1] + T[2, 3] V1[3] + T[2, 4] V1[4] + T[2, 5] V1[5]) (V1[2] - V2[2])
```

On rearranging, we have: - (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 see by inspection how generalize the above expression to N-neurons, and an arbitrary ith unit that gets switched (rather than number 2).

$$\Delta E = -\Delta V_i \sum_{i \neq j} T_{ij} V_j \tag{3}$$

**The main result**

Using the above formula in equation 3, we can now show 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. we want to prove that 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 we need to prove that ΔE is either zero or negative:

$$\Delta E \leq 0$$

**The proof**

Consider the weighted sum describing the input to unit *i* :

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

**Case 1:** If the weighted sum is negative, then the change in $V_i$ must either be zero or negative. This is because if the weighted sum is 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 ΔE=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 $\Delta V_i$ and summation term is positive (product of two negative terms), so ΔE is negative (i.e., the sign of the change in energy in Equation 3 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. So either it was a one and remained so ($\Delta V_i$ = 0, then ΔE=0), or changed in the positive direction from a zero to one ($\Delta V_i$ = +1). In the latter case, the product of $\Delta V_i$ and the summation term (see Equation 3) 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 discrete-state Hopfield network

## Applications 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 that later, 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.

## Applications to 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).

Below 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

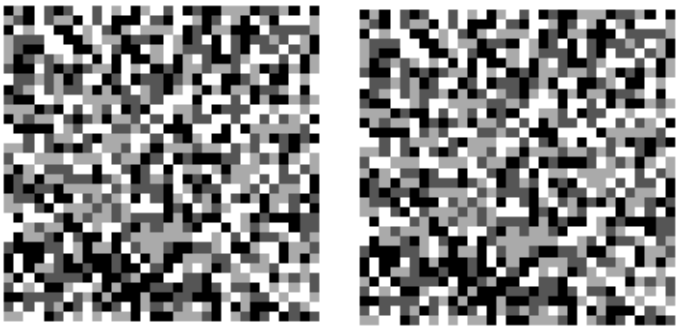
before you do that, you might want to decide which features go with which--i.e. pupils to pupils, nostrils to nostrils, etc.. Establishing correspondences between two similar but not quite identical patterns has also been a central, and challenging problem in both stereo vision and motion processing.

In the supplementary material, (**Correspondence_HopfieldDis.nb**), we show how the weights in a network can be set up to represent constraints. We study 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.

Let’s motivate the problem here using stereo vision. Later, we will also see how the Boltzmann machine can be adapted to the constraint satisfaction problem for stereo vision.

## Introduction to 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 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.)
Below is an example of a random dot stereogram originally developed by Bela Julesz in the 1960's.



The patterns are 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 is a highly simplified example of 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. In other words, 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 because so may of the features (i.e. the pixel intensities) are the same--there is considerable potential for false matches.  Also a minority of points don't have matching pairs (i.e. the ones that got filled in the vertical slot left after shifting the subpatch). We'll get to this in a moment, but first let's make a stereogram using *Mathematica*.

Human perception solves the stereo correspondence, so then we ask whether we can devise a neural network algorithm to solve it.

## 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.

In[280]:=
```
size = 32; patchsize = size/2;
```

In[281]:=
```
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.

In[282]:=
```
disparity = 1;
il = size/2-patchsize/2 + 1; jl = il  - disparity;
```

In[284]:=
```
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.

In[287]:=
```
ir = size/2-patchsize/2 + 1; jr = ir;
```

In[288]:=
```
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 too. We can illustrate this using ListAnimate[]. 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 central square patch disappears again into the camouflage.

In[291]:=
```
gl = ArrayPlot[twoDlefteye, Mesh → False,
    Frame → False, Axes → False, ImageSize → Tiny];
gr = ArrayPlot[twoDrighteye, Mesh → False, Frame → False,
  Axes → False, ImageSize → Tiny];
ListAnimate[{gl, gr, gl, gr}, 4]
```

Out[292]=

In[293]:=
```
GraphicsRow[{gl, gr}]
```

Out[293]=

There are many algorithmic approaches to solving the correspondence problem. Models for way the brain solves it in space for stereo are very different than solutions for the problem over time for motion. Local measurements of stereo disparity and motion direction and speed can be formulated in terms of spatial and space-time filters, adopting quasi-linear filtering methods that we learned earlier.

See **Correspondence_HopfieldDis.nb** in the class web page for a demonstration of using the Hopfield net to solve correspondence.

# Graded response Hopfield network

## The model of the basic neural element

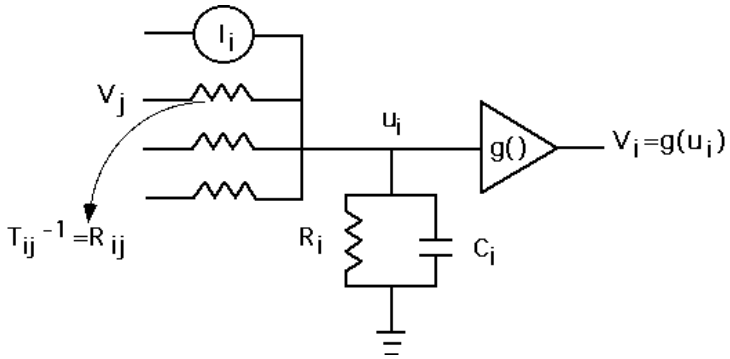
Hopfield's 1982 paper was strongly criticized for having an unrealistic model of the neuron. In 1984, he published another influential paper with an improved neural model which generalized the previous one. The model was intended to capture the fact that neural firing rate can be considered a continuously 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. Earlier, in 1983, Cohen and Grossberg had published a paper showing conditions for network convergence.

Previously, 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().

Electrical engineers define an operational amplifier (or "op-amp") to have a very high input impedance (resistance). The “op-amp” is a basic and versatile building block for constructing analog circuits, such as amplifiers.

For neural modeling, its simplifying key theoretical property is that it essentially draws no current. We’ll see how that simplifies things below.

**Analog model neuron revisited.** The figure 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). (Conductance is the reciprocal of electrical resistance.) $I_i$ 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 two conductances between any two neurons is assumed to be the same. In other words, the weight matrix has a zero diagonal ($T_{ii}$=0), and is symmetric ($T_{ij}$=$T_{ji}$). We follow Hopfield’s convention, and for simplicity let the output range continuously between -1 and 1 for the graded response network (rather than taking on discrete values of 0 or 1 as for the network in the earlier discrete Hopfield network.)

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 (which we used earlier in the derivation of the leaky-integrate-and fire neuron):

-- Kirchoff's rule (sum of currents at a junction has to be zero, i.e. sum in has to equal sum out)
--Ohm's law (I= V/R = V x conductance = V x T), and
--the current across a capacitor is proportional to the rate of change of voltage (I= C du/dt), where C is the constant of proportionality, called the capacitance.

Resistance is the reciprocal of conductance (T=1/R). We write an expression representing the requirement that the total current into the op amp be zero, or equivalently that the sum of the incoming currents at the point of the input to the op amp is equal to the sum of currents leaving:

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

With a little rearrangement, our model of the neuron is:

$$C_i \frac{du_i}{dt} = \sum_j T_{ij} V_j - \frac{u_i}{R_i} + I_i \tag{4}$$

$$V_i = g(u_i) \tag{5}$$

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

Here is where Hopfield's key insight lies. All parameters ($C_i$,$T_{ij}$,$R_i$,$I_i$,g) are assumed fixed, and all we want to know is how the state vector $V_i$ changes with time. We could imagine that the state vector could have almost any trajectory depending on initial conditions, 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:

$V_i \longrightarrow V_i^s$.

where the superscript s indicates a stable state. In other words, the state vector migrates to state-space points where $dV_i/dt = 0$. This is a steady-state solution to the above equations. (Recall the derivation of the steady-state solution for the limulus equations).

*This is an important result because it says that the network can be used to store and recall memories.*

To prove this, Hopfield defined an energy function as:

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

The form of the sigmoidal non-linearity, g(), was taken to be an inverse tangent function (see below). $g^{-1}()$ is the inverse of g(), i.e. it tells us what the input to the non-linearity would be given an output V.

The expression for E looks complicated, but we want to examine how E changes with time, and with a little manipulation, we can obtain an expression for dE/dt which is easy to interpret.

Let's get started. If we take the derivative of E with respect to time, then for symmetric T, we have (using the product rule, and the chain rule for differentiation of composite functions):

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

where we've used Equation 5 to replace $g^{-1}(V)$ by u (after taking the derivative of $\int_0^{V_i} g_i^{-1}(V)\, dV$ with respect to time).

Substituting the left side of the expression from Equation 4 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):

$$= -\sum_i C_i \frac{dg_i^{-1}(V_i)}{dV_i}\left(\frac{dV_i}{dt}\right)^2 \qquad (6)$$

We now want to show that all the factors in the sum are positive (or more precisely non-negative). It is easy to see that $(dV_i/dt)^2$ can't be negative. And capacitance is too, by definition.

But we need to show that the derivative of the inverse g function is always positive too. We'll show it graphically.

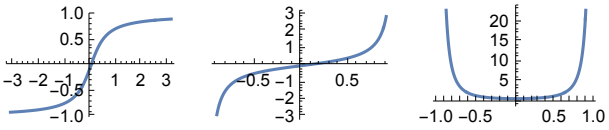
In[339]:=
```
Clear[g]
a1 := (2/Pi); b1 := Pi 1.4 / 2;
g[x_] := N[a1 ArcTan[b1 x]];
inverseg[x_] := N[(1/b1) Tan[x/a1]];
```

The exercise below uses Mathematica to calculate the inverse of g.

The derivative of the inverse of **g[]** , in Mathematica, is **`inverseg'[x]`**. Here are plots of g, the inverse of g, and the derivative of the inverse of g:

In[343]:=
```
GraphicsRow[{Plot[g[x], {x, -π, π}, ImageSize → Tiny],
  Plot[inverseg[x], {x, -0.9, 0.9}], Plot[inverseg'[x], {x, -1, 1}]}]
```

Out[343]=




The third panel on the right shows that the derivative of the inverse is never negative.

**We now have everything we need to prove convergence.**

So because $\frac{dg_i^{-1}(V_i)}{dV_i}$, capacitance and $(dV_i/dt)^2$ are always positive, and given the minus sign, the right hand side of the equation can never be positive--the time derivative of energy is never positive, i.e. energy never increases as the network's state vector evolves in time.

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

Further, from the above equation, we can see that stable points, i.e. where dE/dt is zero, correspond to attractors in state space. Mathematically, we have:

$$dE/dt \le 0$$

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

In the language of dynamical systems, E is said to be a Lyapunov function for the system of differential

equations that describe the neural system whose neurons have graded responses.

► 1. Use the product rule and the chain rule from calculus to fill in the missing steps

The derivative of the inverse of **g[]** is **inverseg′[x]**. Here are plots of g, the inverse of g, and the derivative of the inverse of g:

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

In[337]:=
```
Solve[a ArcTan[b y]==x,y]
```

Out[337]=
$$\left\{\left\{y \to \text{ConditionalExpression}\left[\frac{\text{Tan}\left[\frac{x}{a}\right]}{b}, \left(\text{Re}\left[\frac{x}{a}\right] == -\frac{\pi}{2} \&\& \text{Im}\left[\frac{x}{a}\right] < 0\right) || -\frac{\pi}{2} < \text{Re}\left[\frac{x}{a}\right] < \frac{\pi}{2} || \left(\text{Re}\left[\frac{x}{a}\right] == \frac{\pi}{2} \&\& \text{Im}\left[\frac{x}{a}\right] > 0\right)\right]\right\}\right\}$$

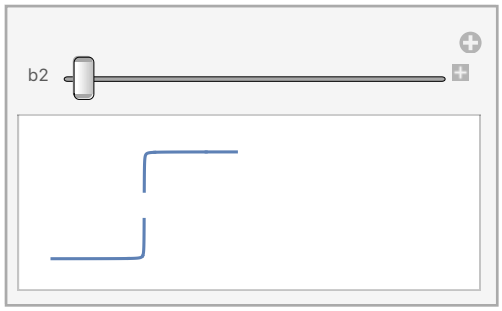
► 3. The continuous valued Hopfield network generalizes the discrete model

As we saw in earlier lectures, dividing the argument of a squashing function such as g[] by a small number makes the sigmoid more like a step or threshold function.

*Note: This non-linearity provides a single-parameter bridge between the discrete (two-state) Hopfield (b2<<1), the continuous Hopfield networks (b2 ~ 1), and linear networks (b2>>1):*

In[338]:=
```
Manipulate[
 Plot[g[1 / b2 x], {x, -π, π}, ImageSize → Tiny, Axes → False], {{b2, 1}, .01, 15}]
```

Out[338]=




# Simulation of a 2 neuron Hopfield network

## Definitions

For simplicity, 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[344]:=
```
Clear[g]

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

## 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[349]:=
```
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 continuous (rather than asynchronous and discrete). In order to do a digital computer simulation, as we did with the limulus case, we will approximate the dynamics with discrete updates of the neurons' activities. The following function computes the output just up to the non-linearity. In other words, the **Hopfield[]** function represents the current input at the next time step to the op-amp model of the ith neuron. We represent these values of u by **uu**, and the weights by matrix **Tm**.

In[354]:=
```
Hopfield[uu_,i_] := uu[[i]] +
    dt ((Tm.g[uu])[[i]] - uu[[i]]);
```

This is a discrete-time approximation that 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 iterations. Then we will plot the pairs of activities of the two neurons over the iterations. The next line randomly samples which of the two neurons to update, i.e. for asynchronous updating.

The next line simulates the evolution through state space of this Hopfield network.

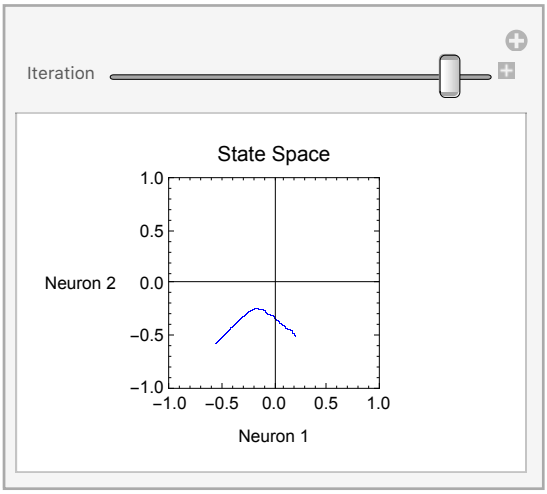
In[355]:=
```
result = Table[{k = RandomInteger[{1, 2}], u[[k]] = Hopfield[u, k], u}, {2800}];
```

In[356]:=
```
result = Transpose[result][[3]];
```

We use ListPlot[] to visualize the trajectory of the state vector.

In[357]:=
```
Manipulate[
 ListPlot[g[result[[1 ;; i]]], Joined → 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]}, ImageSize → Small],
 {{i, 1, "Iteration"}, 1, Dimensions[result][[1]], 1}]
```

Out[357]=




## Energy landscape

Now let's make a contour plot of the energy landscape. We will need a *Mathematica* function expression for the integral of the inverse of the g function (`inverseg'[x]`), 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[358]:=
```
Integrate[(1/b) Tan[x1/a],x1]/.x1->x
```

Out[358]=
$$-\frac{a \operatorname{Log}\left[\operatorname{Cos}\left[\frac{x}{a}\right]\right]}{b}$$

Change the above output to input form:

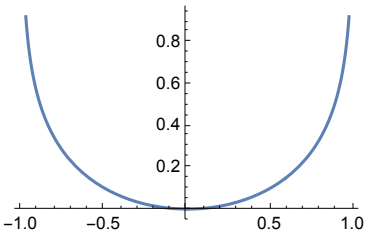
In[359]:=
```
-(a Log[Cos[x/a]]/b)
```

Out[359]=
$$-\frac{a \operatorname{Log}\left[\operatorname{Cos}\left[\frac{x}{a}\right]\right]}{b}$$

And copy and paste the input form into our function definition for the integral of the inverse of g:

In[360]:=
```
inig[x_] := -N[ a1 Log[Cos[x/a1]] / b1 ]; Plot[inig[x], {x, -1, 1}]
```

Out[360]=




We write a function for the above expression for energy:

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

In[361]:=
```
Clear[energy];
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[363]:=
```
gcontour = ContourPlot[energy[{x,y}],{x,-1,1},
    {y,-1,1},AxesOrigin->{0,0},
    PlotRange->{-.1,.8}, Contours->32,ColorFunction → "Pastel",
    PlotPoints->30,FrameLabel->{"Neuron 1","Neuron 2"},
    Frame->True, AspectRatio->1,RotateLabel->False,PlotLabel->"Energy over State Space"];
```

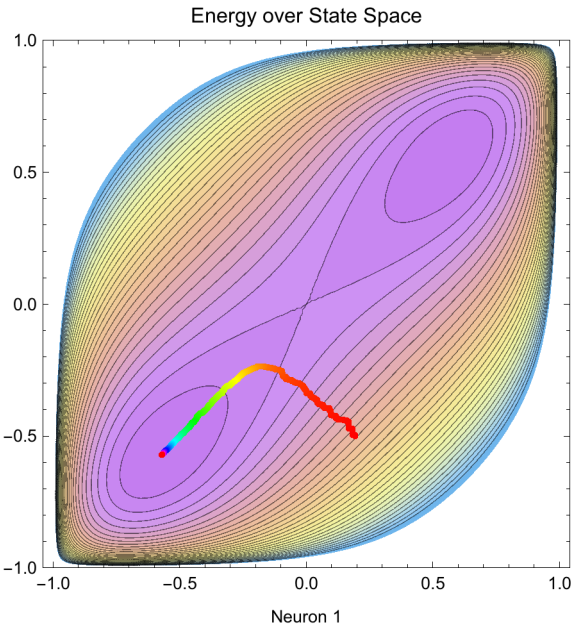
We can visualize the time evolution by color coding state vector points according to Hue[time], where H starts off red, and becomes "cooler" with time, ending up as violet (the familiar rainbow sequence: ROYGBIV).

In[364]:=
```
gcolortraj = Graphics[
  ({Hue[#1/Length[g[result]]], Point[{g[result][[#1]][[1]], g[result][[#1]][[2]]}]} &) /@
   Range[1, Length[g[result]]], AspectRatio → Automatic];
```

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

In[365]:=
```
Show[gcontour, gcolortraj, ImageSize → {340, 340}]
```

Out[365]=




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

# Applications of continuous value Hopfield network: Autoassociative memory

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

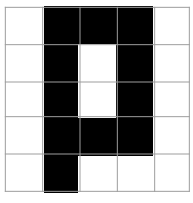
In[366]:=
```
SetOptions[ArrayPlot, ImageSize → Tiny, Mesh → True, DataReversed → True];
```

### The stimuli

We will store the letters P, I, and T in a 25x25 element weight matrix.

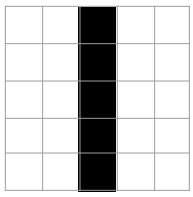
In[367]:=
```
ArrayPlot[Pmatrix =
  {{0, 1, 0, 0, 0}, {0, 1, 1, 1, 0}, {0, 1, 0, 1, 0}, {0, 1, 0, 1, 0}, {0, 1, 1, 1, 0}}]
```

Out[367]=



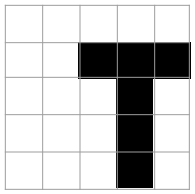
In[368]:=
```
ArrayPlot[Imatrix =
  {{0, 0, 1, 0, 0}, {0, 0, 1, 0, 0}, {0, 0, 1, 0, 0}, {0, 0, 1, 0, 0}, {0, 0, 1, 0, 0}}]
```

Out[368]=



In[369]:=
```
ArrayPlot[Tmatrix =
  {{0, 0, 0, 1, 0}, {0, 0, 0, 1, 0}, {0, 0, 1, 1, 1}, {0, 0, 0, 0, 0}}]
```

Out[369]=



For maximum separation, we will put them near the corners of a hypercube.

In[370]:=
```
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[375]:=
```
{Tv.Iv,Tv.Pv,Pv.Iv, Tv.Tv, Iv.Iv, Pv.Pv}
```

Out[375]=
```
{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[376]:=
```
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[377]:=
```
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 effectively makes a decision between learned alternatives.

## Sensible hallucinations to a noisy input

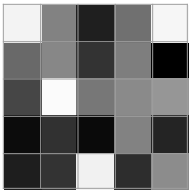
We will set up a version of the graded response Hopfield net with synchronous updating.

In[378]:=
```
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 Rorschach input pattern). We will give as input uncorrelated uniformly distributed noise and find out what the network “perceives”:

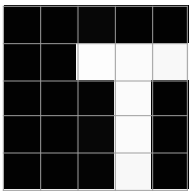
In[380]:=
```
forgettingT = Table[2 RandomReal[] - 1, {i, 1, Length[Tv]}];
ArrayPlot[Partition[forgettingT, width]]
```

Out[380]=



In[381]:=
```
rememberingT = Nest[Hopfield, forgettingT, 30];
ArrayPlot[Partition[g[rememberingT], width], PlotRange → {-1, 1}]
```

Out[381]=



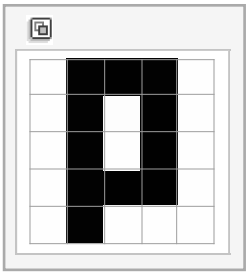
In this case, the random input produces sensible output--the network recalled a specific letter despite the noise.

► 5. Will you always get the same letter as the network's hallucination?

Use Dynamic[] to find out with the button below.

In[382]:=
```
DocumentNotebook[
 Dynamic[ArrayPlot[Partition[g[rememberingT], width], PlotRange → {-1, 1}]]]
```

Out[382]=



In[383]:=
```
Button["Re-initialize", forgettingT = Table[2 RandomReal[] - 1, {i, 1, Length[Tv]}];
 rememberingT = Nest[Hopfield, forgettingT, 30]]
```

Out[383]=

Re-initialize

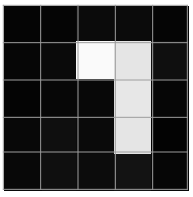
Click the button for random initial conditions

## 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[384]:=
```
ArrayPlot[Partition[g[weights.forgettingT], width], PlotRange → {-1, 1}]
```

Out[384]=



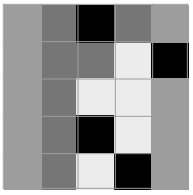
The noisy input doesn't always give a meaningful output. You can try other noise samples. Because of superposition it will typically not give the meaningful discrete memories that the Hopfield net does. However, sometimes the linear matrix memory will produce meaningful outputs and sometimes the Hopfield 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[385]:=
```
forgettingT = 0.1 (0.2 Tv - 0.15 Iv - 0.3 Pv);
ArrayPlot[Partition[forgettingT, width]]
```

Out[385]=

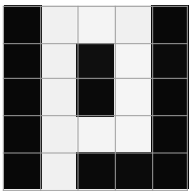


Now let's see what the Hopfield algorithm produces. We will start with a smaller step size. It sometimes takes a little care to make sure the descent begins with small enough steps. If the steps are too big, the network can converge to the same answers one sees with the linear associator followed by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

In[386]:=
```
dt = 0.01;
Hopfield[uu_] := uu + dt (weights.g[uu] - uu);
rememberingT = Nest[Hopfield,forgettingT,30];
```

In[389]:=
```
ArrayPlot[Partition[g[rememberingT], width], PlotRange → {-1, 1}]
```

Out[389]=

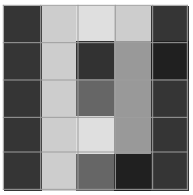


## Comparison with linear associator

The linear associator followed by the squashing function gives:

In[390]:=
```
ArrayPlot[Partition[g[weights.forgettingT], width], PlotRange → {-1, 1}]
```

Out[390]=



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.

► 6. Illustrate how the Hopfield net can be used for error correction in which the input is a noisy version of one of the letters

## 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[409]:=
```
inig[x_] := -N[(a1*Log[Cos[x/a1]])/b1];
energy[Vv_] := -0.5 ((weights.Vv).Vv) +
               Sum[inig[Vv][[i]], {i,Length[Vv]}];
```

In[411]:=
```
energy[.99*Iv]
energy[g[forgettingT]]
energy[g[rememberingT]]
```

Out[411]= −263.969

Out[412]= 4.46595

Out[413]= −244.693

One of the virtues of knowing the energy or Lyapunov 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 what we found was not the minimum.

In[414]:=
```
dt = 0.01;
Hopfield[uu_] := uu + dt (weights.g[uu] - uu);
rememberingT = Nest[Hopfield,forgettingT,600];
```

In[417]:=
```
energy[g[rememberingT]]
```

Out[417]=
```
-244.693
```

# Conclusions

Although Hopfield networks have had a considerable impact on the field, they have had limited practical applications in engineering. 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. There is current theoretical work on Hopfield networks, involving topics such as memory capacity and efficient learning, and relaxations of the strict theoretical presumptions of the original model. For a recent application in neuroscience, see: Kopec et al. (2015). Cortical and Subcortical Contributions to Short-Term Memory for Orienting Movements. Neuron, 88(2), 367–377.

One way that the field has built on the Hopfield model, is towards more general probabilistic 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.

The idea of a dynamical system evolving to reduce energy is important, and remains with us. We will see how minimizing energy is formally equivalent to maximizing the probability that a solution is correct, given its inputs.

# 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.
Kopec, C. D., Erlich, J. C., Brunton, B. W., Deisseroth, K., & Brody, C. D. (2015). Cortical and Subcortical Contributions to Short-Term Memory for Orienting Movements. Neuron, 88(2), 367–377. http://-doi.org/10.1016/j.neuron.2015.08.033
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.