Introduction to Neural Networks
Probability, Energy & the Boltzmann machine

Initialize

- Spell check off. Plots small.

```
In[83]:= Off[General::spell1];
SetOptions[Plot, ImageSize -> Small];
SetOptions[ArrayPlot, ColorFunction -> Hue, Mesh -> True, ImageSize -> Tiny];
```

Introduction

- The past two lectures
  Learning as searching for weights that minimize prediction errors (statistical regression & Widrow-Hoff)
  Network dynamics as minimizing “energy” (Hopfield)
  Common theme: analysis of objective functions (e.g. sum of squared differences in regression, or energy in network dynamics) can provide useful insights into neural networks.

- Today
  Deepen and broaden our conceptual tools by establishing a relationship between “energy” and probability. This will provide an important transition in the development of neural network models and in the material in this course.
  Focus first on constraint satisfaction and seeking a global energy minimum
  Then on learning a statistical model

- Preview of future
  By treating neural network learning and dynamics in terms of probability computations, we’ll begin to see how a common set of tools and concepts can be applied to:
  1. Inference (as in perception and memory recall): as a process which makes the best guesses given data. What it means to be a “best” is specified in terms of computations on probability distributions (and later in the course, utility functions).
  2. Learning: a process that discovers the parameters of a probability distribution
3. **Generative modeling**: a process that generates data from a probability distribution

---

### Stereo correspondence: An example of constraint satisfaction

**Making a random dot stereogram**

- **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`. Similarly, `ir` and `jr` are the lower left insert positions in the `twoRighteye` 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.

```plaintext
In[4]:= size = 32; patchsize = size/2;
glevels = 4;

In[6]:= twoDlefteye = RandomInteger[glevels - 1, {size, size}];
twoRighteye = twoDlefteye;
patch = RandomInteger[glevels, {patchsize, patchsize}];
```

- **Left eye**

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

```plaintext
In[9]:= disparity = 1;
   il = size/2-patchsize/2 + 1; jl = il - disparity;

In[11]:= i=1;
labeled = Flatten[patch][[i++]];
twoDlefteye = MapAt[labeled, 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.

  ```math
  \text{In[14]} := \text{ir} = \text{size}/2 - \text{patchsize}/2 + 1; \text{jr} = \text{ir};
  ```

  ```math
  \text{In[15]} := 
  \text{i} = 1; 
  \text{label}[x_] := \text{Flatten}[\text{patch}][[i++]]; 
  \text{twoDrighteye} = \text{MapAt}[\text{label}, \text{twoDrighteye}, 
  \text{Flatten}[[\text{i}, \text{j}]], 
  \{\text{i}, \text{ir}, \text{ir} + \text{Dimensions}[\text{patch}][[1]] - 1\}, 
  \{\text{j}, \text{jr}, \text{jr} + \text{Dimensions}[\text{patch}][[2]] - 1\}];
  ```

  ```math
  \text{In[18]} := 
  \text{gl} = \text{ArrayPlot}[\text{twoDlefteye}, \text{Mesh} \rightarrow \text{False}, 
  \text{Frame} \rightarrow \text{False}, \text{Axes} \rightarrow \text{False}, \text{ImageSize} \rightarrow \text{Tiny}]; 
  \text{gr} = \text{ArrayPlot}[\text{twoDrighteye}, \text{Mesh} \rightarrow \text{False}, \text{Frame} \rightarrow \text{False}, 
  \text{Axes} \rightarrow \text{False}, \text{ImageSize} \rightarrow \text{Tiny}];
  ```

  ```math
  \text{In[20]} := 
  \text{GraphicsRow}[[\text{gl}, \text{gr}]]
  ```

  ```math
  \text{Out[20]} :=
  ```

  ![Images](https://via.placeholder.com/150)

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

- **Simplify the problem to one dimension**

  We will apply the constraints proposed by Marr and Poggio (1976) to try to solve the correspondence problem for just the middle rows of `twoDlefteye` and `twoDrighteye`:
In[21]:= lefteye = twoDlefteye[size/2]; righteye = twoDrighteye[size/2];
ArrayPlot[{lefteye, righteye}, AspectRatio -> 1/8, Mesh -> True]

Out[21]=

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

What would it mean to solve the correspondence problem in which there is a unique and correct answer? One way to think of it is in terms of a plot of the left and right eye’s pixel positions along the horizontal and vertical axes of a 2D grid. Then we mark the grid points with WHITE to indicate points that should correspond to each other (if we knew the answer), as shown here:

But how to find this answer from the data? And to do it in a neurally plausible way?

Following Marr and Poggio, we will try to solve the correspondence (i.e. which pairs of pixels in the two images belong together) using three constraints:

- compatibility,
- uniqueness, and
- smoothness.

We will see what these constraints mean as we move along. The challenge is to design a network that enforces these constraints.

- **The compatibility constraint**

The compatibility constraint says that similar features in each eye should match, in other words like pixels (same
graylevel) in each image should match. We can represent compatibility by constructing a "compatibility matrix" which has a 1 at each location where there is a possible match (i.e. in grayscale color), and zeros elsewhere.

```
In[22]:=
compatibility = Table[
    If[
        lefteye[[i]]==righteye[[j]],1,0],
    {i,size},{j,size}];
```

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

```
In[23]:=
ArrayPlot[compatibility, DataReversed -> True, Mesh -> True,
  Frame -> True, FrameLabel -> {"x, other eye", "x, one eye"},
  ColorFunction -> ColorData["GrayTones"]]
```

```
Out[23]=
```

The uniqueness and smoothness constraints

But many of these possible matches seem unlikely. A given point in one eye shouldn't have lots of matches in the other eye--a point on a surface in the world typically maps to at most one point in each eye. We would like to find a line through the compatibility matrix to indicate unique matches. So we have to discourage more than one unit from being on in any given row or column (i.e. enforce a uniqueness constraint).

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

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

We will follow Marr and Poggio and set up a threshold logic unit at each location of the compatibility matrix, and wire each up to reflect the above constraints. We are going to have to worry about the boundaries. There are several ways of doing this. One is to have a "free" boundary in which the connection weights at the boundaries are actually different (to compensate for a lack of neighbors in the other directions). Although not biologically realistic, a second way is to use a toroidal geometry, restricting indices by the following modulus function: \( \text{myMod}[x_] := \text{Mod}[x-1, \text{size}]+1 \). This option is computationally convenient and makes it possible to comply with the restriction of symmetric connections everywhere.

To encourage uniqueness, let's make the vertical and horizontal connection weights all equal and negative with a weight \( \text{inhib} \). To encourage smoothness, we'll make the diagonal support positive with mutually excitatory weights \( \text{excit} \). The
network will have biases for each unit (which are equivalent to appropriate thresholds for each unit) proportional to the original compatibility matrix. These biases correspond to the Ui’s in the Hopfield net and serve to prevent the network from loosing this strong constraint from the data as the iterations progress. Note that we won’t set up the network calculations using a matrix and matrix multiplications as you will do in the Hopfield memory examples. This is because most of our connections are zero and we don’t want to waste time multiplying zero elements and adding them up. So the code below will look messier.

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

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

\[
V = \text{compatibility; } V1 = V; \\
\text{excit} = 2; \text{inhib} = -1; \text{k} = 6; \text{theta} = 13; \\
\text{threshold}[\text{xx}_\text{]} := \text{N}[\text{If}[\text{xx} > \text{theta}, 1, 0]]; \\
\text{myMod}[\text{x}_\text{]} := \text{Mod}[\text{x} - 1, \text{size}] + 1;
\]

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

```
ln[29]: = Timing[numiterations = 10000; For[iter = 1, iter <= numiterations, iter++,
   If[Mod[iter, 10] == 1, ArrayPlot[V1, Mesh -> True]]; i = RandomInteger[size - 1] + 1; j = RandomInteger[size - 1] + 1; V1[i, j] = threshold[inhib (V1[myMod[i + 1], myMod[j]] +
   V1[myMod[i - 1], myMod[j]] + V1[myMod[i], myMod[j - 1]] +
   V1[myMod[i]], myMod[j + 1]] V1[myMod[i] + 2, myMod[j]] +
   V1[myMod[i - 2], myMod[j]] + V1[myMod[i], myMod[j - 2]] +
   V1[myMod[i], myMod[j] + 2] V1[myMod[i + 1], myMod[j]] +
   V1[myMod[i], myMod[j - 3], myMod[j]] + V1[myMod[i], myMod[j - 3]] +
   V1[myMod[i], myMod[j + 3]] V1[myMod[i] + 4, myMod[j]] +
   V1[myMod[i - 4], myMod[j]] + V1[myMod[i], myMod[j - 4]] +
   V1[myMod[i], myMod[j + 4]] +
   excit (V1[myMod[i + 1], myMod[j + 1]] + V1[myMod[i - 1], myMod[j - 1]] +
   V1[myMod[i + 2], myMod[j + 2]] + V1[myMod[i - 2], myMod[j - 2]] +
   V1[myMod[i] + 3, myMod[j + 3]] + V1[myMod[i] + 3, myMod[j + 3]] +
   V1[myMod[i] + 4, myMod[j + 4]] + V1[myMod[i - 4], myMod[j - 4]] +
   k compatibility[i, j]]]);
```

Out[29]=

```
{4.41914, Null}
```
With simulated annealing

```math
\text{temp0} = 1;
\text{iter} = 1;
\text{temp} = \frac{\text{temp0}}{\log(1 + \text{iter})};
```

```math
\text{boltz}[x\_, T\_] := 1 / (1 + \exp[-(x - \text{theta}) / T]);
```

```math
\text{Timing}[\text{numiterations} = 10000; \text{For}[\text{iter} = 1, \text{iter} \leq \text{numiterations}, \text{iter}++,
\text{If}[\text{Mod}[\text{iter}, 10] == 1, \text{ArrayPlot}[\text{V1}, \text{Mesh} \to \text{True}]],
\text{i} = \text{RandomInteger}[\text{size} - 1] + 1; \text{j} = \text{RandomInteger}[\text{size} - 1] + 1;
\text{xxx} = \text{inhib} \times
(V1[[\text{myMod}[i + 1], \text{myMod}[j]]], V1[[\text{myMod}[i - 1], \text{myMod}[j]]], V1[[\text{myMod}[i], \text{myMod}[j + 1]]]
\text{V1[[myMod[i], myMod[j + 1]]] V1[[myMod[i + 2], myMod[j]]] + V1[[myMod[i - 2], myMod[j]]] V1[[myMod[i], myMod[j - 2]]] +
\text{V1[[myMod[i], myMod[j + 2]]] V1[[myMod[i + 3], myMod[j]]] + V1[[myMod[i - 3], myMod[j]]] V1[[myMod[i], myMod[j - 3]]] +
\text{V1[[myMod[i], myMod[j + 3]]] V1[[myMod[i + 4], myMod[j]]] + V1[[myMod[i - 4], myMod[j]]] V1[[myMod[i], myMod[j - 4]]] +
\text{V1[[myMod[i], myMod[j + 4]]] + excit \times (V1[[\text{myMod}[i + 1], \text{myMod}[j + 1]]], V1[[\text{myMod}[i - 1], \text{myMod}[j - 1]]] + V1[[\text{myMod}[i + 2], \text{myMod}[j + 2]]] + V1[[\text{myMod}[i - 2], \text{myMod}[j - 2]]] V1[[\text{myMod}[i + 3], \text{myMod}[j + 3]]] +
\text{V1[[myMod[i - 3], myMod[j - 3]]] V1[[myMod[i + 4], myMod[j + 4]]] + V1[[\text{myMod}[i - 4], \text{myMod}[j - 4]]] + k \text{compatibility}[i, j];
\text{pdeltaE} = \text{N}[\text{boltz}[\text{xxx}, \text{temp}]];)
```

```math
\text{V1[i, j] = If}[\text{pdeltaE} \geq \text{RandomReal}[\, 1, 0\, ]];
```

```
{4.50071, \text{Null}}
```

Statistical physics, computation, and statistical inference

At the beginning of this course, we noted that John von Neumann, one of the principal minds behind the architecture of the modern digital computer, wrote that brain theory and theories of computation would eventually come to more resemble statistical mechanics or thermodynamics than formal logic. We have already seen in the Hopfield net, the development of the analogy between statistical physics systems and neural networks. The relationship between computation and statistical physics was subsequently studied by a number of physicists (cf. Hertz et al., 1991). We are going to look at a neural network model that exploits the relationship between thermodynamics and computation both to find global minima and to modify weights. Further, we will see how relating energy to probability leads naturally to statistical inference theory. Much of the current research in neural network theory is done in the context of statistical inference (Bishop, 1995; Ripley, 1996; MacKay, 2003).
Probability Preliminaries

We'll go over the basics of probability theory in more detail later in a later lecture. Today we'll review enough to see how energy functions can be related to probability, and energy minimization to maximizing probability.

Random variables, discrete probabilities, probability densities, cumulative distributions

- **Discrete distributions**: random variable $X$ can take on a finite set of discrete values

  
  $X = \{x(1),...,x(N)\}$
  
  \[
  \sum_{i=1}^{N} p_i = \sum_{i=1}^{N} p(X = x(i)) = 1
  \]

- **Continuous densities**: $X$ takes on continuous values, $x$, in some range.

  Density: $p(x)$
  
  Analogous to material mass, we can think of the probability over some small domain of the random variable as "probability mass":
  
  \[
  \text{prob}(x < X < dx + x) = \int_{x}^{x+dx} p(x) \, dx
  \]
  
  \[
  \text{prob}(x < X < dx + x) \approx p(x) \, dx
  \]
  
  By analogy with discrete distributions, the area under $p(x)$ must be unity:
  
  \[
  \int_{-\infty}^{\infty} p(x) \, dx = 1
  \]
  
  like an object always weighing 1.

  Cumulative distribution:
  
  \[
  \text{prob}(X < x) = \int_{-\infty}^{x} p(X) \, dX
  \]

- **Densities of discrete random variables**

  The Dirac Delta function, $\delta[\bullet]$, allows us to use the mathematics of continuous distributions for discrete ones, by defining the density as:
  
  \[
  p[x] = \sum_{i=1}^{N} p_i \delta[x - x[i]]
  \]
  
  where $\delta[x - x[i]] = \begin{cases} \infty & \text{for } x = x[i] \\ 0 & \text{for } x \neq x[i] \end{cases}$
  
  Think of the delta function, $\delta[\bullet]$, as $\epsilon$ wide and $1/\epsilon$ tall, and then let $\epsilon \to 0$, so that:
\[ \int_{-\infty}^{\infty} \delta(y) \, dy = 1 \]

The above density, \( p[x] \), is a series of spikes. It is infinitely high only at those points for which \( x = x[i] \), and zero elsewhere. But "infinity" is scaled so that the local mass or area around each point \( x[i] \), is \( p_t \).

**Check out Mathematica's functions: DiracDelta, KroneckerDelta. What is the relationship of KroneckerDelta to IdentityMatrix?**

- **Joint probabilities**

  \( \text{Prob}\,(X \text{ AND } Y) = p(X, Y) \)

  Joint density: \( p(x, y) \)

  Two events, \( X \) and \( Y \), are said to be independent if the probability of their occurring together (i.e. their "joint probability") is equal to the product of their probabilities:

  \[ p(X, Y) = p(X)p(Y) \]

- **Conditional probabilities**

  Suppose two random variables, \( X \) and \( Y \), are independent. What if I told you the value of \( Y \) (say \( Y=y \), where \( y \) is some specific value)? This would not change your knowledge about the possible values of \( X \). The intuition is that knowledge of \( Y \) provides no help with making statistical decisions about \( X \). Conditional probabilities are written: \( p(X \mid Y) \). If \( X \) and \( Y \) are independent, then \( p(X \mid Y) = p(X) \).

  Imagine you are blindfolded, and asked to guess who is sitting next to you, i.e. \( X = ? \). If you are in class (\( Y = "\text{in class}" \), your guesses would be quite different than if you are at home (\( Y = "\text{at home}" \)). The conditional probabilities, \( p(X = \text{your classmate} \mid \text{you are sitting in class}) \) and \( p(X = \text{your classmate} \mid \text{you are sitting at home}) \) are different.

- **Three basic rules of probability**

  Suppose we know everything there is to know about a set of variables \( (A,B,C,D,E) \). What does this mean in terms of probability? It means that we know the joint distribution, \( p(A,B,C,D,E) \). In other words, for any particular combination of values (\( A=a, B=b, C=c, D=d, E=e \)), we can calculate (e.g. using a formula), look up in a table, or determine using an algorithm, the number \( p(A=a,B=b, C=c, D=d, E=e) \).

  Deterministic relationships are special cases.

- **Rule 1: Conditional probabilities from joints: The product rule**

  Probability about an event changes when new information is gained.

  \[ \text{Prob}(X \text{ given } Y) = p(X\mid Y) \]

  \[ p(X \mid Y) = \frac{p(X, Y)}{p(Y)} \]

  \[ p(X, Y) = p(X \mid Y) \, p(Y) \]

  The form of the product rule is the same for densities as for probabilities.
■ Rule 2: Lower dimensional probabilities from joints: The sum rule (marginalization)

\[ p(X) = \sum_{i=1}^{N} p(X, Y(i)) \]

\[ p(x) = \int_{-\infty}^{\infty} p(x, y) \, dy \]

■ Rule 3: Bayes' rule

From the product rule, and since \( p[X,Y] = p[Y,X] \), we have:

\[ p(Y \mid X) = \frac{p(X \mid Y) \, p(Y)}{p(X)}, \text{ and using the sum rule, } p(Y \mid X) = \frac{p(X \mid Y) \, p(Y)}{\sum_{F} p(X, Y)} \]

■ Preview

1. Inference: a process which makes guesses given data. Optimal inference makes the best guess according to some criterion.

   e.g. given data \( X=x \), what value of \( Y \) produces the biggest \( p(Y\mid X=x) \)? I.e. the most probable value of \( Y \), given \( X = x \)?

2. Learning: a process that discovers the parameters of a probability distribution

   Supervised learning: Given training pairs \( \{ X_i, Y_i \} \), what is \( p(X, Y) \)?

   Unsupervised learning: Given data \( \{ X_i \} \), what is \( p(X) \)?

3. Generative modeling: a process that generates data from a probability distribution

   Given \( p(X) \), produce sample data \( \{ X_i \} \).

Below we introduce the Boltzmann machine as a historical example of how one can do accomplish all three processes within one network architecture.

### Probability and energy

■ Probabilities of hypotheses contingent on data, Bayes' rule

Conditional probabilities are central to modeling statistical decisions about hypotheses that depend on data. For example, the posterior probability of \( H \), given data \( d \) is written:

\[ p(H \mid d) \]

It is called "posterior", because it is the probability after one has the data. It is more constrained than the prior probability, \( p(H) \). It is called the "prior" because it represents the knowledge one has before having the data.

If one has a formula for the posterior probability, then it is possible to devise optimal strategies to achieve well-defined goals of inference. For example, imagine the data are given and thus these variables are fixed. A device that picks the
hypothesis, \( H \), that makes the posterior probability biggest is optimal in the sense that it makes the fewest errors on average. In this case, the goal is well-defined--to achieve the least average error rate. This kind of decision maker is called a maximum a posteriori (or MAP) estimator.

Often it is easier to find a formula for \( p(d \mid H) \), then the other way around. In other words, we might have a good idea of the generative model -- \( p(d|H) \) and \( p(H) \). \( p(d|H) \) is called the likelihood and \( p(H) \) is the prior.

If the prior \( p(H) \) and likelihood are both known, one can do MAP estimation in this case too because of Bayes' rule which tells us the posterior given the prior and the likelihood:

\[
p(H \mid d) = \frac{p(d \mid H) p(H)}{p(d)}
\]

There are two assumptions that can simplify MAP estimation. First, \( p(d) \) is fixed at some constant value (we have the data and so \( p(d) \) isn't changing while we try to decide on the best hypothesis to explain the data). Further, we often don't have reason to prefer one hypothesis \( H=H' \), over any other, say \( H=H'' \). So \( p(H) \) is constant. If these two conditions hold, then MAP estimation is equivalent to finding the \( H \) that makes \( p(d|H) \) biggest. This latter strategy is called maximum likelihood estimation.

- **Putting probability and energy together: The Gibbs distribution**

Let \( E(V_1, \ldots, V_n) \) be an energy function for a network, as in a Hopfield model. Then we can write a probability function, called the Gibb's distribution, for the network as:

\[
p(V_1, \ldots, V_n) = \kappa \exp\left(\frac{-E(V_1, \ldots, V_n)}{T}\right)
\]

\( T \) is a parameter that controls the "peakedness" of the probability distribution (e.g. later we'll see that if \( V \)'s are continuous and the energy is a quadratic function of the \( V \)'s, the Gibbs distribution becomes a multivariate Gaussian, and if each unit has the same variance, and they are independent, we have \( T = 2\sigma^2 \)). From the physicist's point of view, \( T \) is temperature--e.g. the hotter the matter, the more variance there is in the particle velocities. For a magnetic material, an increase in thermal fluctuations makes it more likely for little atomic magnets to flip out of their otherwise regular arrangement. \( \kappa \) is a normalization constant determined by the constraint that the total probability over all possible states must equal one. From a neural perspective, the higher \( T \), the more likely the neuron is to take on values that are not strictly determined by its inputs.

- **If the Hopfield net seeks states that minimize energy, what kind of statistical estimator is the Hopfield net?**

Now imagine that some of the values of our units are given and fixed. In other words a subset of the units are declared to be the input, and are "clamped" at specific levels. Call these fixed values \( V_1^s \). The others, \( V_i \), can vary. The conditional probability is written as:
\[
p(V_1, \ldots, V_m | V'_1, \ldots, V'_k) = \kappa \exp \left( \frac{-E(V_1, \ldots, V_m; V'_1, \ldots, V'_k)}{T} \right)
\]

So from a statistician's point of view, a network that is evolving to minimize an energy function, is doing a particular form of Bayesian estimation, MAP. Of course, there is one major caveat--the network may settle into a local energy minimum, which would correspond to a local probability mode. And a local mode may not be the most probable, just as a local minimum may not be the global minimum.

- **Sidenote on terminology**

We've already noted that energy is equivalent to the Lyapunov function of dynamical systems. Other analogous terms you may run into are: Hamiltonian (in statistical mechanics), and cost or objective functions in optimization theory.

---

**Boltzmann machine: Constraint satisfaction**

**Introduction**

We've seen how local minima in an energy function can be useful stable points for storing memories. However, for constraint satisfaction problems such as the stereo example, local minima can be a real annoyance--one would like to find the global minimum, because this corresponds to the state-vector that should best satisfy the constraints inherent in the weights and the data input.

One of the early contributions to improving the odds of finding the global minimum was an algorithm called the Boltzmann machine by Ackley et al. (1985). Like the Hopfield network, the Boltzmann machine is a recurrent network with units connected to each other with symmetric weights. The units are binary threshold logic units.

But there were two key differences. Unlike the 1982 Hopfield net, the Boltzmann machine uses a stochastic update rule that allows occasional increases in energy.

More importantly, the Boltzmann machine had a feature that made it a more potentially powerful learning machine. Although fully connected, some units could be "hidden" and others "visible". Similar to the back-prop nets, the hidden units could learn to capture statistical dependencies of higher order than two.

First we take a look at the update rule, and then the learning rule. The learning rule will lead us to a different view of supervised learning, in which the goal is to model the state of the environment--the joint probability of the various possible combinations of inputs.

**Finding the global minimum: Theory**

- **TLUs and energy again**
The starting point is the discrete Hopfield net (1982), with a view towards solving constraint satisfaction, rather than memory problems. Energy is then a measure of the extent to which a possible combination of hypotheses violates the constraints of the problem. We’ve seen this with the stereo problem. Let V1 and V2 be the outputs of neural elements 1 and 2. These two outputs can be thought to represent local "hypotheses". A positive connection weight (e.g. \( T_{12} > 0 \)) means that local hypotheses, V1 and V2 support one another. A negative weight would mean that the two hypotheses inhibit each other--and should discourage both from being accepted at the same time.

Some of the inputs can be clamped (held to a constant value, as in the stereo compatibility matrix below), and the rest allowed to evolve. In this way the network finds the conditional local minimum (or equivalently, the maximum of the corresponding conditional probability).

\[
V_i = \begin{cases} 
V_i^c & \text{if } i \text{ is a clamped unit} \\
1 & \text{if } \sum T_{ij} V_j \geq U_i \\
0 & \text{if } \sum T_{ij} V_j < U_i 
\end{cases} 
\]  

\[E = -\sum_{i<j} T_{ij} V_i V_j + \sum_i U_i V_i\]  

(Notation: the sum for \( i\neq j \) is the same as the 1/2 the sum for \( i \) not equal to \( j \), because the weight matrix is assumed to be symmetric, and the diagonals are not included. So this may look different than the Hopfield energy, but it isn't.).

As we have done several times before, we can remove the explicit dependence on the threshold \( U_j \), by including weights -\( U_i \), each with a clamped input of 1 that effectively biases the unit. Then, as we saw for the Hopfield net:

\[E = -\sum_{i<j} T_{ij} V_i V_j\]

Consider the \( i \)th unit. If \( V_i \) goes from 1 to 0, the energy gap between two states corresponding to \( V_i \) being off (hypothesis \( i \) rejected) or on (hypothesis \( i \) accepted) is:

\[\Delta E_i = \sum_j T_{ij} V_j\]

---

**Boltzmann update rule**

Recall that under certain conditions, Hopfield showed that energy can't increase.

To allow escapes from local minima, the idea is to allow occasional *increases* in energy, an idea that goes back to 1953 (Metropolis et al., 1953). So rather than using a hard-threshold to decide whether to output 1 or 0, we use the weighted sum as an input to a probabilistic decision rule, based on the familiar sigmoidal logistic function. Let's see how it works.

Set

\[V_i = 1 \text{ with probability } p_i\]
where \( p_i \) approaches 1, if the weighted sum of the inputs to \( i \) (\( \Delta E_i \)) gets big enough:

\[
p_i = p(\Delta E_i) = \frac{1}{1 + e^{-\Delta E_i/T}}
\]

Again, \( T \) is a free parameter that plays the role of temperature in thermodynamics.

(Notation alert: \( T_{ij} \) refers to the weights between neuron \( j \) and \( i \), and \( T \) to the temperature. Completely different meanings for \( T! \))

When we implement the algorithm below, we draw a (uniformly distributed) number between 0 and 1 "out of a hat", and if that number is less than \( p_i = p(\Delta E_i) \) we set \( V_i \) to 1. Otherwise, set it to zero. Specifically, the update rule is:

\[
V_i = \begin{cases} 
1 & \text{if } \text{Random}[\text{Real}] < p(\Delta E_i) = p(\sum_j T_{ij} V_j) \\
0 & \text{otherwise}
\end{cases}
\]

If the temperature is very low (\( T \) about zero), the update rule is the same as that for a deterministic TLU. This is because if the weighted sum of inputs is way bigger than zero, \( p \) is virtually at 1. The probability of setting \( V \) to 1 is then guaranteed. If the weighted sum is less than zero, \( p \) is zero, and the probability of setting \( V \) to 1 is nil.

Here is a plot of \( p_i \), with a high and low temperatures:

\[
\text{Simulated annealing}
\]

Now if we just let the Boltzmann rule update the state vector at a high temperature, the network will never settle to a stable point in state space. Conversely, if we set the temperature low, the network is likely to get stuck in a local minimum. The key idea behind introducing the notion of temperature is to start off with a high temperature, and then gradually ‘cool’ the
network. This simulates the physical process of annealing. If one heats metal, and then cools it rapidly, there is less crystalline structure or alignment of the atoms. This is a high energy state, and is desirable for making strong metal. The steel has been tempered. Slower cooling allows the substance to achieve a lower energy state with more alignment, with correspondingly more potential fractures. Although bad for metal strength, slow annealing is good for constraint satisfaction problems.

The "Gibbs Sampler" is a general form of updating for n-valued nodes making up a Markov Random Field (Geman and Geman, 1984). It has been shown that a suitably slow annealing schedule will guarantee convergence (Geman and Geman, 1984):

\[ T(n) > \frac{c}{\log(1 + n)} \]

This annealing schedule, however, can be VERY slow, in fact too slow to be usually practical, except for small scale problems.

Both of these are examples of Markov chain Monte Carlo methods.

**Local minimum demonstration**

Let's look at a simple example where the standard discrete Hopfield net gets stuck in a local minimum, but the Boltzmann machine with annealing gets out of it. We'll construct in a 2D grid (similar to the stereo example). Each unit gets connected to its four nearest neighbors with weights given by weight (=1). As illustrated in an exercise below, the global minimum for this network is when all the units are turned off. But does the TLU update rule get us there from all initial states?

#### Initialization

We will use a toroidal geometry to keep the programming simple.

```mathematica
In[102]:= Mod2[x_,n_] := Mod[x-1,n] + 1;
threshold[x_] := N[If[x>=0,1,-1]];

In[104]:= weight = 1;
size = 16;
numiterations = 10;
```

Below, we will deliberately construct a weight matrix so that the energy function has a local minimum at the following state vector:

```mathematica
In[107]:= V = Table[-1,{i,size},{j,size}];
V[[2,3]] = 1; V[[2,4]] = 1; V[[2,5]] = 1;
V[[3,3]] = 1; V[[3,4]] = 1; V[[3,5]] = 1;
V[[4,3]] = 1; V[[4,4]] = 1; V[[4,5]] = 1;
```
Here is a picture of the state vector with the local minimum:

```
In[111]:= ArrayPlot[V, PlotRange -> {-5, 5}]
```

```
Out[111]=
```

- Asynchronous updating without annealing: Getting stuck

Each unit is connected to its four nearest neighbors with weights given by weight (=1). The rest of the weights are zero. So the update rule is:

```
In[112]:= update[Vv_, ii_, jj_] :=
    threshold[weight (Vv[[ Mod2[ii + 1, size],
            jj ]]] +
    Vv[[ Mod2[ii - 1, size], jj ]] +
    Vv[[ ii, Mod2[jj - 1, size] ]] +
    Vv[[ ii, Mod2[jj + 1, size] ]]);
```

```
In[113]:= iter=1;
```

```
In[100]:= gg0 = Dynamic[ArrayPlot[V, PlotRange -> {-5, 5}]]
```

```
Out[100]=
```
In[101]:= updateAndPlot := Module[{}, 
    For[i = 1, i ≤ size size, i++, 
        iindex = RandomInteger[size - 1] + 1; 
        jindex = RandomInteger[size - 1] + 1; 
        V[iindex, jindex] = update[V, iindex, jindex]; 
        ArrayPlot[V, PlotRange -> (-5, 5), 
            Epilog -> Inset[iter++, {size - 2, size - 2}]]];

In[114]:= Button["Update and Plot", updateAndPlot]

Out[114]= Update and Plot

nothing is happening...not very interesting!

- Asynchronous updating with annealing: Getting unstuck

In[115]:= 
   temp0 = 1;
   iter = 1;
   temp = \frac{temp0}{\log[1 + iter]};
Options exercise 1: Energy function

Write a function to calculate the energy function for the above network.

What is the energy of the ground state?

What is the energy of the local minimum we constructed above?

Options exercise 2: Thermal equilibrium

Make two versions of the above simulation in which 1) the temperature is fixed; 2) the temperature is gradually lowered
(as above). Start with a random initial setting of the network.

### Texture synthesis: An example of generative modeling of data

The ideas in the Boltzmann machine can also be applied to problems of pattern synthesis. This involves a different perspective on the same network--now instead of viewing it as doing inference, the network is a generative model. Let’s let the units take on continuous values. Consider the quantized case. Suppose rather than just binary values, our units can take on a range of values. We'll introduce the idea with a demonstration of a pattern synthesizer for texture generation.

#### Local energy

\[
\text{Local energy (potential) at location } i = \sum_{j \in N(i)} f(V_i - V_j)
\]

The local energy determines a local conditional probability for the values at the ith site:

\[
p(V_i \mid V_j, j \in N_i) = \kappa e^{-\sum_{j \in N(i)} f(V_i - V_j)}
\]

#### Sampling from textures using local updates

To draw a sample at the ith node, we draw from the local (conditional) probability distribution:

![Gibbs Sampler Diagram](image.png)
The Gibbs Sampler

- Set up image arrays and useful functions

\[
\text{size} = 32; \ T0 = 1.; \ \text{ngray} = 16;
\]
\[
\text{brown} = \text{N@Table[RandomInteger[\{1, \text{ngray}\}], \{i, 1, \text{size}\}, \{i, 1, \text{size}\}]};
\]
\[
\text{next}[\_] := \text{Mod}[\_, \text{size}] + 1;
\]
\[
\text{previous}[\_] := \text{Mod}[\_ - 2, \text{size}] + 1;
\]
\[
\text{Plus @@ Flatten[brown]};
\]
\[
\text{Length}[\text{Flatten[brown]}]
\]

We can try several types of potentials.

- Ising potential

\[
\text{Clear}[f];
\]
\[
f[\_, s\_, n\_] := \text{If[Abs}[\_] < 0.5, 0, 1];
\]
\[
s0 = 1.; \ n0 = 5;
\]
\[
\text{Plot}[f[\_, s0, n0], \{\_, -2, 2\}, \text{AxesLabel} \rightarrow
\]
\[
\{"\!*\*\*\*\SubscriptBox\[(\_\_\_\_\_\_\_), \SubscriptBox\[(\_\_\_), \_\_\_\_\_\_\_\_\_\_\_]\,-1111\!*\*\*\*\SubscriptBox\[(\_\_\_\_\_\_\_), \SubscriptBox\[(\_\_), \_\_\_\_\_\_\_]\]", \_}\}
\]

\[
\text{Out}[147]=
\]
\[
\begin{array}{c}
 f \\
 1.0 \\
 0.8 \\
 0.6 \\
 0.4 \\
 0.2 \\
 -2 -1 1 2 \\
 V_i - V_j
\end{array}
\]
**Geman & Geman potential**

\[
\text{Clear}[f];
\]
\[
f[x_, s_, n_] := N[\sqrt{\left(\frac{\text{Abs}[x]}{s}\right)^n \left(1 + \text{Abs}[x]^n\right)}];
\]
\[
s0 = 0.25; \quad n0 = 2; \\
\text{Plot}[f[x, s0, n0], \{x, -2, 2\}, \text{PlotRange} \rightarrow \{0, 1\}, \\
\text{AxesLabel} \rightarrow \{"V_i-V_j", f\}]
\]

**Define the potential function using nearest-neighbor pair-wise "cliques"**

Suppose we are at site i. \(x\) is the activity level (or graylevel in a texture) of unit (or site) i. \(\text{avg}\) is a list of the levels at the neighbors of the site i.

\[
\text{Clear}[\text{gibbspotential}, \text{gibbsdraw}, \text{tr}];
\]
\[
\text{gibbspotential}[x_, \text{avg}_, T_] := 
N[\text{Exp}[-(f[x - \text{avg}[[1]], s0, n0] + f[x - \text{avg}[[2]], s0, n0] + \\
f[x - \text{avg}[[3]], s0, n0] + f[x - \text{avg}[[4]], s0, n0]) / T]]; \\
\]

**Define a function to draw a single pixel gray-level sample from a conditional distribution determined by pixels in neighborhood**

The idea is to calculate the cumulative distribution corresponding to the local conditional probability, pick a uniformly distributed number, which determines the value of the sample \(x\) (through the distribution). To save time, we avoid having to normalize the cumulative (it should asymptote to 1) by drawing a uniformly distributed random number between the max and min values of the output of \text{FoldList} (the cumulative sum).
In[154]:= gibbsdraw[avg_, T_] := 
Module[{}, temp = Table[gibbspotential[x + 1, avg, T], {x, 0, ngray - 1}];
  temp2 = FoldList[Plus, temp[[1]], temp];
  temp10 = Table[{temp2[[i]], i - 1}, {i, 1, Dimensions[temp2][1]}];
  tr = Interpolation[temp10, InterpolationOrder -> 0];
  mintemp = Min[temp2];
  maxtemp = Max[temp2];
  x = Floor[tr[ri]]; Return[{x, temp2}];]

In[155]:= gg = Dynamic[ArrayPlot[brown, Mesh -> False, ColorFunction -> ColorData["SouthwestColors"], PlotRange -> {1, ngray}]]

Out[155]=

In[156]:= For[iter = 1, iter <= 10, iter++, T = 0.25;
  For[j1 = 1, j1 <= size, j1++, {i, j} = RandomInteger[{{1, size}, 2}];
   avg = {brown[[next[i]], j]], brown[[i, next[j]]], brown[[i, previous[j]]],
   brown[[previous[i], j]]}, brown[[i, j]] = gibbsdraw[avg, T][1];]; gg;

■ "Drawing" a pattern sample

Was it a true sample? Drawing true samples means that we have to allow sufficient iterations so that we end up with images whose frequency corresponds to the model. How long is long enough?
Finding modes

- Define annealing schedule

```
In[157]:= anneal[iter_, T0_, a_] := T0 / (a (1 + Log[iter]));
Plot[anneal[iter, T0, 1], {iter, 1, 20},
    PlotRange -> {0, 2}, AxesOrigin -> {0, 0}]
```

- Producing a texture sample with annealing

In the previous simulation, the temperature (variance) was fixed. We can also use annealing, to try to draw a sample near a mode.

```
In[159]:= brown = N[Table[RandomInteger[{1, ngray}], {i, 1, size}, {i, 1, size}]];
gg2 = Dynamic[ArrayPlot[brown, Mesh -> False,
    ColorFunction -> ColorData["SouthwestColors"], PlotRange -> {1, ngray}]]
```
Boltzmann Machine: Learning

We've seen how a stochastic update rule improves the chances of a network evolving to a global minimum. Now let's see how learning weights can be formulated as a statistical problem.

The Gibbs distribution again

Suppose T is fixed at some value, say T=1. Then we could update the network and let it settle to thermal equilibrium, a state characterized by some statistical stability, but with occasional jiggles. Let $V_\alpha$ represent the vector of neural activities.

The probability of a particular state $\alpha$ is given by:

$$ p(V_\alpha) = Ke^{-E_\alpha / T} $$

$$ K = \sum_{all \ states \ k} e^{-E_k / T} $$

Recall that the second equation is the normalization constant the ensures that the total probability (i.e. over all states) is 1.

We divide up the units into two classes: hidden and visible units. Values of the visible units are determined by the environment. If the visible units are divided up into "stimulus" and "response" units, then the network should learn associations (supervised learning).

If they are just stimulus units, then the network observes and organizes its interpretation of the stimuli that arrive (unsupervised learning).

Our goal is to have the hidden units discover the structure of the environment. Once learned, if the network was left to run freely, the visible units would take on values that reflect the structure of the environment they learned. In other words, the network has a generative model of the visible structure. This is like dreaming.

Consider two probabilities over the visible units, V:

$P(V)$ - probability of visible units taking on certain values determined by the environment.

$P'(V)$ - probability that the visible units take on certain values while the network is running at thermal equilibrium.

If the hidden units have actually "discovered" the structure of the environment, then the probability $P$ should match $P'$. 

In[161]:= 

```mathematica
For[iter = 1, iter <= 10, iter++,
T = anneal[iter, T0, 1]; For[j = 1, j <= size, j++,
{i, j} = RandomInteger[{1, size}, 2]; avg = {brown[[next[i], j],
brown[[i, next[2]]], brown[[i, previous[[j]]]], brown[[previous[i], j]]};
brown[[i, j]] = gibbsdraw[avg, T][1];]; ArrayPlot[brown, Mesh -> False,
ColorFunction -> ColorData["SouthwestColors"], PlotRange -> {1, ngray}]];
```
How can one achieve this goal? Recall that for the Widrow-Hoff and error backpropagation rules, we started from the constraint that the network should minimize the error between the network’s prediction of the output, and the actual target values supplied during training. We need some measure of the discrepancy between the desired and target states for the Boltzmann machine. The idea is to construct a measure of how far away two probability distributions are from each other—how far P is from P'. One such function is the Kullback-Leibler (KL) divergence measure or relative entropy (also known as the "Gibbs G measure").

$$G(T_{12}, T_{13}, \ldots, T_{ij}, \ldots) = \sum_{\text{all states}} P(V_\alpha) \log \left( \frac{P(V_\alpha)}{P'(V_\alpha)} \right)$$

Then we need a rule to adjust the weights so as to bring P' -> P in the sense of reducing the KL measure G. Ackley et al. derived the following rule for updating the weights so as to bring the probabilities closer together. Make weight changes \(\Delta T_{ij}\) such that:

$$\Delta T_{ij} = \varepsilon \left( p_{ij} - p'_{ij} \right)$$

where \(p_{ij}\) is the probability of \(V_i\) and \(V_j\) both being 1 when environment is clamping the states at thermal equilibrium averaged over many samples. \(p'_{ij}\) is the probability of \(V_i\) and \(V_j\) being 1 when the network is running freely without the environment at equilibrium.

### Descendants of Boltzmann machines

As noted above, convergence through simulated annealing can be impractically slow. The mean field approximation is one technique used to improve convergence (cf. Ripley, 1996). Boltzmann machines can be considered a special case of belief networks which we will study later (Ripley, 1996). Learning, as you might imagine, is also very slow because of the need to collect lots of averages before doing a weight update.

The Boltzmann machine learns to approximate the joint probability distribution on a set of binary random variables. Some of the variables are designated inputs, and others outputs. Learning large scale joint distributions is known to be a hard problem in statistics, and the success of the Boltzmann machine has been limited to small scale problems. One successor to the Boltzmann machine is the Helmholtz machine and its derivatives (Dayan et al., 1995; Hinton, 1997).

Texture modeling has received considerable attention in biological vision and computer graphics. A recent advance in learning pattern distributions is the Minimax theory (Zhu and Mumford, 1997). Here is an observed sample:

![Sample Image](Image)

Here is a synthesized sample after Minimax entropy learning:
Below is an a sample of a "text" texture from http://www.cns.nyu.edu/~lcv/texture/

In general, it is a hard problem to draw true samples from high dimensional spaces. One needs a quantitative model of the distribution and a method such as Gibbs sampling to draw samples. And in the above text example, the sample was not drawn from an explicit probability distribution.

For a recent example of an application of texture synthesis to understanding the networks of the brain’s visual system, see Freeman, J & Simoncelli, E. P. (2011)

### Sleep-wake algorithms & “deep belief” networks.

For a computational example of an application to learning and inference, see:

http://www.cs.toronto.edu/~hinton/digits.html


### References


