Introduction to Neural Networks

Probability and neural networks

- Initialize standard library files:

```math
In[5]:=
Off[General::spell1];
SetOptions[ContourPlot, ImageSize \rightarrow \text{Small}];
SetOptions[Plot, ImageSize \rightarrow \text{Small}];
SetOptions[ListPlot, ImageSize \rightarrow \text{Small}];
```

---

**Introduction**

**Last time**

From energy to probability. From Hopfield net to Boltzmann machine. We showed how the Hopfield network, which minimized an energy function, could be viewed as finding states that increase probability.

**Neural network computations from the point of view of statistical inference**

By treating neural network learning and dynamics in terms of probability computations, we can begin to see how a common set of tools and concepts can be applied to:

1. **Inference**: as a process which makes the best guesses given data. E.g. find \(H\) such that \(p(H \mid \text{data})\) is biggest.

2. **Learning**: a process that discovers the parameters of a probability distribution. E.g. find weights such that \(p(\text{weights} \mid \text{data})\) is biggest

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

E.g. draw samples from \(p(\text{data} \mid H)\)
Today

- Boltzmann machine: learning the weights
- Go over stereo example of constraint satisfaction
- Examples of generative modeling: drawing samples

**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:

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

$$\kappa = \frac{1}{\sum_{\text{all states } k} e^{-E_k/T}}$$

Recall that the second equation is the normalization constant that 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 using stochastic sampling, 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.

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 $P'$. 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) measure or relative entropy (also known as the "Gibbs G measure").

\[
G(T_1, T_2, \ldots, T_l, \ldots) = \sum_{\text{all states}} P(V) \log \left( \frac{P(V)}{P'(V)} \right)
\]

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:

\[
\Delta T_{ij} = \varepsilon (p_{ij} - p'_{ij})
\]

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.

### 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 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.

\[
\text{In}[570]:=\text{Clear}[\text{label}]
\]

\[
\text{In}[571]:=\text{size} = 32; \text{patchsize} = \text{size}/2; \text{offset} = \text{size}/4; \text{glevels} = 2;
\]
\textbf{Left eye}

The left eye's view will have the \texttt{patch} matrix displaced \texttt{n=disparity} pixels to the left.

\begin{verbatim}
In[576]:= disparity = 2;
\end{verbatim}

\textbf{Right eye}

The right eye's view will have the \texttt{patch} matrix centered.

\begin{verbatim}
In[577]:= twoDlefteye[[offset ;; offset + patchsize - 1, offset ;; offset + patchsize - 1]] =
    patch[[1 ;; patchsize, 1 ;; patchsize]];

twoDrighteye[[offset ;; offset + patchsize - 1, offset + disparity ;; offset + disparity + patchsize - 1]] =
    patch[[1 ;; patchsize, 1 ;; patchsize]];
\end{verbatim}
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 `twoDrigheye`:

```math
In[582]:= lefteye = twoDlefteye[size/2]; righteye = twoDrigheye[size/2];
lefteye = twoDlefteye[23]; righteye = twoDrigheye[23];
ArrayPlot[{lefteye, righteye}, AspectRatio -> 1/8, Mesh -> True]
```

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

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 graylevel color), and zeros elsewhere.

In[584]:=

\[
\text{compatibility} = \text{Table}[\text{If[}
\text{lefteye[[i]]==righteye[[j]],1,0],}
\text{\{i,size\},\{j,size\}};\]

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[585]:=

\[
\text{ArrayPlot[compatibility, DataReversed → True, Mesh → True,}
\text{Frame → True, FrameLabel → \{"x, other eye", "x, one eye"\},}
\text{ColorFunction → ColorData["GrayTones"]}}\]

Out[585]=

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

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

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

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

\[
\text{In}[586]:= \quad \text{V} = \text{compatibility}; \text{V1} = \text{V};
\]

\[
\text{In}[587]:= \quad \text{excit} = 2; \text{inhib} = -1; \text{k} = 6; \text{theta} = 13; \\
\text{threshold}[\text{xx}_] := \text{N}[[\text{If}[\text{xx} > \text{theta}, 1, 0]]];
\]

\[
\text{In}[589]:= \quad \text{myMod}[\text{x}_] := \text{Mod}[\text{x} - 1, \text{size}]+1;
\]

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.
Stereo correspondence: Running the algorithm

```mathematica
In[590]:= DocumentNotebook[
    Dynamic[
        ArrayPlot[V1, DataReversed -> True, Mesh -> True,
            ImageSize -> Small, ColorFunction -> ColorData["GrayTones"]]
    ]]
```

```mathematica
Out[590]=
```
With a deterministic TLU update rule

```
In[591]:= Timing[numiterations = 10000; For[iter = 1, iter \[less_equal] 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[
            inhibit (V1[[myMod[i + 1], myMod[j]]] + V1[[myMod[i - 1], myMod[j]]] + 
                V1[[myMod[i], myMod[j - 1]]] + 
                V1[[myMod[i], myMod[j + 1]]] V1[[myMod[i + 2], myMod[j]]] + 
                V1[[myMod[i - 2], myMod[j]]] + V1[[myMod[i], myMod[j - 2]]] + 
                V1[[myMod[i], myMod[j + 2]]] V1[[myMod[i + 3], myMod[j]]] + 
                V1[[myMod[i - 3], myMod[j]]] + V1[[myMod[i], myMod[j - 3]]] + 
                V1[[myMod[i], myMod[j + 3]]] V1[[myMod[i + 4], myMod[j]]] + 
                V1[[myMod[i - 4], myMod[j]]] + V1[[myMod[i], myMod[j - 4]]] + 
                V1[[myMod[i], myMod[j + 4]]]) + 
            excit (V1[[myMod[i + 1], myMod[j]]] + V1[[myMod[i - 1], myMod[j - 1]]] + 
                V1[[myMod[i + 2], myMod[j + 1]]] V1[[myMod[i + 2], myMod[j - 2]]] + 
                V1[[myMod[i + 3], myMod[j + 2]]] V1[[myMod[i + 3], myMod[j - 3]]] + 
                V1[[myMod[i + 4], myMod[j + 3]]] V1[[myMod[i + 4], myMod[j - 4]]] + 
                k compatibility[i, j]];]]
```

Out[591]= {6.38582, Null}
With simulated annealing

```mathematica
In[554]:= temp0 = .01;
iter = 1;

temp = temp0 / Log[1 + iter];

boltz[x_, T_] := 1 / (1 + Exp[-(x - theta) / T]);

Timing[numiterations = 50000; 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;
   xxx = inhib * (V1[[myMod[i + 1, myMod[j]]] + V1[[myMod[i - 1, myMod[j]]] +
     V1[[myMod[i], myMod[j + 1]]] V1[[myMod[i + 2, myMod[j]]] +
     V1[[myMod[i - 2, myMod[j]]] V1[[myMod[i], myMod[j - 2]]] +
     V1[[myMod[i], myMod[j + 2]]] V1[[myMod[i + 3], myMod[j]]] +
     V1[[myMod[i - 3], myMod[j]]] V1[[myMod[i], myMod[j - 3]]] +
     V1[[myMod[i], myMod[j + 3]]] V1[[myMod[i], myMod[j + 4]]] +
     V1[[myMod[i], myMod[j]]]] +
   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 + 1, myMod[j + 1]]] +
     V1[[myMod[i + 3, myMod[j + 3]]] + V1[[myMod[i + 1, myMod[j + 3]]] +
     V1[[myMod[i + 4, myMod[j + 4]]] + V1[[myMod[i + 4, myMod[j + 4]]] +
     k compatibility[i, j]];
pdeltaE = N[boltz[xxx, temp]];]

V1[[i, j]] = If[pdeltaE >= RandomReal[], 1, 0]]];

Out[557]= {27.4437, Null}
```

Generative modeling

We first go over a few basics.

Density mapping theorem

Suppose we have a change of variables that maps a discrete set of x's uniquely to y's: \(X \rightarrow Y\).
- **Discrete random variables**

No change to probability function. The mapping just corresponds to a change of labels, so the probabilities $p(X) = p(Y)$.

- **Continuous random variables**

In this case, the form of probability density function changes because we require the probability "mass" to be unchanged:

$$p(x) \, dx = p(y) \, dy$$

Suppose, $y = f(x)$

$$p_Y(y) \, dy = p_X(x) \, dx$$

Transformation of variables is used in making random number generators for probability densities other than the uniform distribution, such as a Gaussian.

Below we’ll need to use the cumulative distribution function: $CDF(x) = \text{prob} \{ X < x \} = \int_{-\infty}^{x} p(X) \, dX$

- **Univariate sampling**

- **Making a univariate (scalar) gaussian random number generator:**

  We assume we have a random number generator that provides uniformly distributed numbers between 0 and 1. How can we get numbers that are Gaussian distributed?

  Well, the easiest way is to use a built-in function:

  ```math
  In[593]:= RandomVariate[NormalDistribution[0, 1]]
  Out[593]= 0.726157
  ```

  but we’d like to better understand some principles behind generating random numbers for a specified distribution.

  - **Method 1: Just for Gaussian. Use Central Limit Theorem**

    If all we want to do is make a Gaussian random number generator from a uniformly distributed generator, we can use the Central Limit Theorem. The Central Limit Theorem says that the sum of a sufficiently large number of independent random variables drawn from the same underlying distribution (with finite mean and variance), will be approximately normally distributed. The approximation gets better as the number of samples increases.

    Try the cell below with $\text{nusamples} = 1, 2, .., 10, ..$. 

We'll use the density mapping theorem to turn uniformly distributed random numbers RandomReal[] into gaussian distributed random numbers with mean =0 and standard deviation =1.

\[
\begin{align*}
  & \mathbb{P}_Y (y) \, \delta y = \mathbb{P}_X (x) \, \delta x \\
  & \mathbb{P}_Y (y) \, \frac{\delta y}{\delta x} = \mathbb{P}_X (x)
\end{align*}
\]

Suppose \( \mathbb{P}_Y (y) = 1 \) (over the unit interval, but zero elsewhere). Then

\[
y (x) = \int_{-\infty}^{x} \mathbb{P}_X (x') \, \mathrm{d}x' = F (x)
\]

Thus if we sample from the uniform distribution to get \( y \), \( x \) should be distributed according to \( \mathbb{P}_X (x) \).

To do this, we need a mapping from \( y \rightarrow x \). This is given by the inverse cumulative distribution, i.e. \( P^{-1}(y) \).

Let's implement this. The quick way is to use Mathematica's built-in function to get the inverse cumulative.

Method 2a: Applied to Gaussian

InverseErf[ ] is the inverse of:
\[
\text{erf}(z) = \frac{2}{\sqrt{\pi}} \int_0^z e^{-t^2} \, dt
\]

We can use this to define a function for the inverse cumulative of a gaussian:

\begin{verbatim}
In[595]:= Clear[z];
    z[p_] := Sqrt[2] InverseErf[1 - 2 p];
    Plot[z[y], {y, 0, 1}]
    Out[597]=

Out[597]=
\end{verbatim}

\begin{verbatim}
In[598]:= binsize = 0.1;
    zl = Table[z[RandomReal[]], {5000}];
    freq = BinCounts[zl, {-3, 3, binsize}];
    ListPlot[freq, Filling -> Axis]
    Out[601]=
\end{verbatim}

- **Method 2b: From scratch: Works for almost any distribution.**

Suppose we have a discrete representation of any cumulative distribution. How can we generate samples? For illustration purposes, we'll illustrate the method with a discretization of the Gaussian.

Our first goal is to produce a discrete approximation to the cumulative gaussian. To review where things come from, we'll start with the definition of a Gaussian, and make sure it is normalized.
In[602]:= \text{Integrate}[
\text{Exp}[-(x - x0)^2 / (2 \cdot \sigma^2)], \{x, -\text{Infinity}, \text{Infinity}\}]

Out[602]= \text{ConditionalExpression}\left[\frac{\sqrt{2 \pi}}{\sqrt{\frac{1}{\sigma^2}}}, \text{Re}[\sigma^2] > 0\right]

Let \(x0=0\) and \(\sigma=1\):

In[603]:= \text{Plot}\left[\frac{e^{-\frac{x1^2}{2}}}{\sqrt{2 \pi}}, \{x1, -4, 4\}\right]

Out[603]=

Note that \text{Plot}[\text{PDF}[\text{NormalDistribution}[0,1],x1],\{x1,-4,4\}] gives the same thing using the built-in normal distribution function.

\begin{itemize}
\item \textbf{Cumulative gaussian}
\end{itemize}

In[605]:= \text{Clear}[\text{cumulgauss}, x, x1];
\text{cumulgauss}[x_] := \text{NIntegrate}[\text{Exp}[-(x1^2) / 2] / (\text{Sqrt}[2 \cdot \text{Pi}])],
\{x1, -\text{Infinity}, x\}]

In[607]:= \text{cumulgauss}[\text{Infinity}]


We can plot up \text{cumulgauss}: 
In[608]:= 
\[\text{Plot[cumulgauss[x], \{x, -4, 4\}]\]}

Out[608]=

```
-4 -2 0 2 4
```

Now make a discrete version of the cumulative distribution:

In[609]:= 
\[\text{lcumulgauss = Table\[\{x, cumulgauss[x]\}, \{x, -4., 4., 0.25\}\]};
\text{ListPlot[lcumulgauss, Filling \rightarrow \text{Axis}}\]

Out[610]=

```
-4 -2 0 2 4
```

- **Make inverse cumulative gaussian table**

This is a useful trick whenever you want an inverse function, given a discrete representation.

In[611]:= 
\[\text{invlcumulgauss = RotateLeft[lcumulgauss, \{0, 1\}]\]}

To see what this does, evaluate:

In[612]:= 
\[\{(x1, y1), (x2, y2), (x3, y3)\}
\text{RotateLeft[\{(x1, y1), (x2, y2), (x3, y3)\}, \{0, 1\}]\]}

Out[612]=
\[\{(x1, y1), (x2, y2), (x3, y3)\}\]

Out[613]=
\[\{(y1, x1), (y2, x2), (y3, x3)\}\]
Make interpolated function of the inverse cumulative

Another useful trick.

Interpolation works by fitting polynomial curves to the data. Try the test below with various interpolation orders (the default is 3)
**Draw samples with a standard deviation of \( \text{Sqrt}[10] \)**

```plaintext
\text{In[624]}:= \text{Round[10 interinvlcumulgauss[RandomReal[]]]}
\text{Out[624]= 7}
```

**Draw a bunch of samples, and plot up histogram**

```plaintext
\text{In[625]}:= \text{z = Table[Round[10 interinvlcumulgauss[RandomReal[]]], \{10000\}] ;}
\text{domain = Range[-20, 20];}
\text{Freq = (Count[z, \#1] \&)/@domain;}
\text{In[628]}:= \text{ListPlot[Freq, Filling -> Axis]}
\text{Out[628]=}
```

digression...a quick & dirty way to smooth is to do a moving average
In[629]:= ListPlot[MovingAverage[Freq, 6], Filling -> Axis]

Out[629]=

Plot up cumulative histogram

In[630]:= CumFreq = FoldList[Plus, 0, Freq];
ListPlot[CumFreq, Filling -> Axis]

Out[631]=

Same thing, with Accumulate[]:

In[632]:= CumFreq = Accumulate[Freq];
ListPlot[CumFreq, Filling -> Axis]

Out[633]=
Texture synthesis: An example of generative modeling of data

Last time we took a quick look at the application of the Boltzmann machine 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**

\[
\text{Local energy (potential) at location } i = \sum_{j \in N(i)} f(V_i - V_j) \tag{2}
\]

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

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

- **Sampling from textures using local updates**

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

- Set up image arrays and useful functions

\[
\text{In[634]:=}
\begin{align*}
\text{size} &= 32; \ T0 = 1.; \ \text{ngray} = 16; \\
\text{brown} &= \text{N[Table[RandomInteger[\{1, \text{ngray}\}, \{i, 1, \text{size}\}, \{i, 1, \text{size}\}]];}
\end{align*}
\]
\[
\text{next}[x_] := \text{Mod}[x, \text{size}]+1; \\
\text{previous}[x_] := \text{Mod}[x-2, \text{size}]+1; \\
\text{Plus @@ Flatten[brown]}; \\
\text{Length(Flatten[brown])}
\]

We can try several types of potentials.

- Ising potential

\[
\text{In[639]:=}
\begin{align*}
\text{Clear[f];} \\
\text{f}[x_, s_, n_] := \text{If[Abs}[x] < 0.5, 0, 1]; \\
\text{s0} = 1.; \ \text{n0} = 5; \\
\text{Plot[f[x, s0, n0], \{x, -2, 2\},} \\
\text{AxesLabel }\to \text{"Vs\textasciicircum{i}, \text{Vs}\textasciicircum{j}"}, \ f]\}
\end{align*}
\]

\[
\text{Out[642]=}
\]

\[
\begin{array}{c}
\text{f} \\
\hline
\text{1.0} \\
\text{0.8} \\
\text{0.6} \\
\text{0.4} \\
\text{0.2} \\
\hline
\text{-2} \quad \text{-1} \quad 1 \quad 2 \quad \text{Vs}-\text{Vs}_{\text{j}}
\end{array}
\]
- **Geman & Geman potential**

```math
\begin{align*}
\text{Clear}[f]; \\
f[x_, s_, n_] := N\left[\sqrt{\left(\frac{\text{Abs}[x]}{s}\right)^n \left(1 + \text{Abs}[x]^n\right)}\right]; \\
s0 = 0.25; \text{n0} = 2; \\
\text{Plot}[f[x, s0, n0], (x, -2, 2), \text{PlotRange} \to \{0, 1\}, \text{AxesLabel} \to \\
\{"\!\!\!\{\text{\textSubscriptBox[\{V\}, \{i\}\]}, \!\!\!\{\text{\textSubscriptBox[\{V\}, \{j\}\]}\}"}, f\}]
\end{align*}
```

![Graph of f(x, s0, n0)](image)

- **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. avg is a list of the levels at the neighbors of the site i.

```math
\begin{align*}
\text{Clear}[\text{gibbspotential}, \text{gibbsdraw}, \text{tr}]; \\
\text{gibbspotential}[x_, \text{avg}_-, \text{T}_-] := \\
N[\text{Exp}\left(-\left(f[x - \text{avg}[[1]], s0, \text{n0}] + f[x - \text{avg}[[2]], s0, \text{n0}] + \right.\right. \\
\left.\left. f[x - \text{avg}[[3]], s0, \text{n0}] + f[x - \text{avg}[[4]], s0, \text{n0}]\right)/\text{T}\right]\right]
\end{align*}
```

- **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 `FoldList` (the cumulative sum).
In[649]:=

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];
      maxtemp = Max[temp2]; mintemp = Min[temp2];
      ri = RandomReal[{mintemp, maxtemp}]; x = Floor[tr[ri]];
      Return[{{x, temp2}}];]

In[650]:=

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

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.
\begin{verbatim}
In[654]:= brown = N[Table[RandomInteger[{1, ngray}], {i, 1, size}, {j, 1, size}]];
gg2 = Dynamic[ArrayPlot[brown, Mesh -> False, 
  ColorFunction -> ColorData["SouthwestColors"], PlotRange -> {1, ngray}]]

Out[655]=

In[658]:= For[iter = 1, iter ≤ 100, iter++, T = anneal[iter, T0, 1]; 
  For[j1 = 1, j1 ≤ size 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];]; 
  ArrayPlot[brown, Mesh -> False, 
  ColorFunction -> ColorData["SouthwestColors"], PlotRange -> {1, ngray}]]
\end{verbatim}

References


