Introduction to Neural Networks

Probability and neural networks

- Initialize standard library files:

```math
Off[General::spell1];
SetOptions[ContourPlot, ImageSize -> Small];
SetOptions[Plot, ImageSize -> Small];
SetOptions[ListPlot, ImageSize -> 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 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 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"):

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

Then we need a rule to adjust the weights so as to bring $P' \rightarrow P$ 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.

### Stereo correspondence: An example of constraint satisfaction

The stereo correspondence problem was introduced briefly in a previous lecture. Let’s study the problem in more detail to understand how one might set up a neural architecture to solve the problem.

The stereogram mimics 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. 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 when so many of the features (i.e. the pixel intensities) look the same—there is considerable potential for false matches. A small minority don’t have matching pairs (i.e. the ones that got filled in the vertical slot left after shifting the sub-patch). We’ll get to this in a moment, but first let’s make a stereogram using Mathematica.
Human perception solves the stereo correspondence, so let us see if we can devise a neural network style algorithm to solve it. We should note ahead of time that there are much better solutions to the stereo correspondence problem. However, our example is a simple illustration of how an analysis of constraints can be used to set up a neural network, how stochastic updating can avoid local minima, while at the same time learning something about the computational problem of stereo vision.

Making a random dot stereogram

- Initialize

As in the previous lecture, we 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. \( i_l \) and \( j_l \) are the lower left positions of the lower left corner of patch in the twoDlefteye. Similarly, \( i_r \) and \( j_r \) 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.

\[
\text{In}[39]:= \quad \text{size} = 32; \text{patchsize} = \text{size}/2;
\]

\[
\text{In}[40]:= \quad \text{twoDlefteye} = \text{RandomInteger}[3, \{\text{size}, \text{size}\}]; \text{twoDrighteye} = \text{twoDlefteye}; \\
\text{patch} = \text{RandomInteger}[3, \{\text{patchsize}, \text{patchsize}\}];
\]

- Left eye

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

\[
\text{In}[41]:= \quad \text{disparity} = 1; \\
\text{il} = \text{size}/2-\text{patchsize}/2 + 1; \text{jl} = \text{il} - \text{disparity};
\]

\[
\text{In}[43]:= \quad \text{i}=1; \\
\text{label}[x_] := \text{Flatten}[\text{patch}][[\text{i}++]]; \\
\text{twoDlefteye} = \text{MapAt}[\text{label}, \text{twoDlefteye}, \\
\text{Flatten[Table}[\{\text{i},\text{j}\}, \\
\{\text{i},\text{il},\text{il} + \text{Dimensions[patch]}[[1]]-1\}, \\
\{\text{j},\text{jl},\text{jl} + \text{Dimensions[patch]}[[2]]-1\},1\}];
\]

- Right eye

The right eye's view will have the patch matrix centered.

\[
\text{In}[46]:= \quad \text{ir} = \text{size}/2-\text{patchsize}/2 + 1; \text{jr} = \text{ir};
\]
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`:

```math
\text{lefteye} = \text{twoDlefteye}\left[\frac{\text{size}}{2}\right]; \text{righteye} = \text{twoDrighteye}\left[\frac{\text{size}}{2}\right];
\text{ArrayPlot}\left[\{\text{lefteye, righteye}\}, \text{AspectRatio} \rightarrow \frac{1}{8}, \text{Mesh} \rightarrow \text{True}\right]
```

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:

![Diagram showing grid points marked with WHITE](image)

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[56]:

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

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[57]:

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

Out[57]=

- **The uniqueness constraint**

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**).

- **The smoothness 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 \text{U}i's in the Hopfield net and serve to prevent the network from losing this strong constraint from the data as the iterations progress.

**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.

\[
\text{V} = \text{compatibility}; \text{V1} = \text{V};
\]

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

\[
\text{myMod}[x_] := \text{Mod}[x - 1, \text{size}] + 1;
\]

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.
Lect_14_Probability.nb

In[62]:=

Dynamic[ArrayPlot[V1, DataReversed -> True, Mesh -> True,
            ImageSize -> Small, ColorFunction -> ColorData["GrayTones"]]]

Out[62]=

In[63]:=

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 + 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 + 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[63]= {7.90138, 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**
  
  Form of probability density function does change because we require the probability "mass" to be unchanged: \(p(x)dx = p(y)dy\).

  Suppose, \(y = f(x)\)

  \[p_Y(y) \delta y = p_X(x) \delta x\]

  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:

  \[
  \text{In}[65]:= \text{RandomVariate[NormalDistribution[0, 1]]}
  \]

  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 nusamples = 1, 2, .., 10,..
Manipulate[zi = Table[RandomReal[] - nusamples/2, {1000}];
 Histogram[zi, ImageSize -> Small], {nusamples, 1, 30, 1}]


We’ll use the density mapping theorem to turn uniformly distributed random numbers \texttt{RandomReal[]} into gaussian distributed random numbers with mean $=0$ and standard deviation $=1$.

\[
P_Y (y) \delta y = P_X (x) \delta x
\]

\[
P_Y (y) \frac{\delta y}{\delta x} = P_X (x)
\]

Suppose $P_Y (y) = 1$ (over the unit interval, but zero elsewhere). Then

\[
y (x) = \int_{-\infty}^{x} P_X (x') \, dx' = P (x)
\]

(1)

Thus if we sample from the uniform distribution to get $y$, $x$ should be distributed according to $P_X (x)$. 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 \textit{Mathematica}'s built-in function to get the inverse cumulative.

Method 2a: Applied to Gaussian

$\text{InverseErf}[ ]$ is the inverse of:

\[
erf (z) = \frac{2}{\sqrt{\pi}} \int_0^{z} e^{-t^2} \, dt
\]

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

\[
\text{Integrate}\left[\exp\left(-\frac{(x-x_0)^2}{2 \sigma^2}\right), \{x, -\infty, \infty\}\right]
\]

Let \(x_0=0\) and \(\sigma=1\):
Note that \( \text{Plot}[	ext{PDF}[\text{NormalDistribution}[0,1], x1], (x1, -4, 4)] \); gives the same thing using the built-in normal distribution function.

### Cumulative gaussian

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

1.

We can plot up cumulgauss:

```
Plot[cumulgauss[x], (x, -4, 4)]
```
\( \text{lcumulgauss} = \text{Table}[\{x, \text{cumulgauss}[x]\}, \{x, -4.\&, 4.\& , 0.25\&\}] \);
ListPlot[lcumulgauss, Filling -> Axis]

- Make inverse cumulative gaussian table

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

\( \text{invcumulgauss} = \text{RotateLeft}[\text{lcumulgauss}, \{0, 1\}] \);
To see what this does, evaluate:

\[
\{(x1, y1), (x2, y2), (x3, y3)\}
\text{RotateLeft}[(\{(x1, y1), (x2, y2), (x3, y3)\}, \{0, 1\}]
\[
\{(x1, y1), (x2, y2), (x3, y3)\}
\]
\[
\{(y1, x1), (y2, x2), (y3, x3)\}
\]
ListPlot[invcumulgauss, Filling -> Axis]

- 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 interpola-
Interpolation works by fitting polynomial curves to the data. Try the test below with various interpolation orders (the default is 3)

```math
test = Interpolation[{{1, 2.}, {2, 4}, {3, 9}, {4, 16.}},
                   InterpolationOrder -> 1];
Plot[test[x], {x, 1, 4}]
```

```math
interinvlcumulgauss = Interpolation[invlcumulgauss];
Plot[interinvlcumulgauss[x], {x, 0.01, 0.99}]
```

- Draw samples with a standard deviation of Sqrt[10]

```math
Round[10 interinvlcumulgauss[RandomReal[]]]
```

12

- Draw a bunch of samples, and plot up histogram

```math
z = Table[Round[10 interinvlcumulgauss[RandomReal[]]], {10000}];
domain = Range[-20, 20];
Freq = (Count[z, #1] &)/@domain;
```
digression...a quick & dirty way to smooth is to do a moving average

Plot up cumulative histogram

```mathematica
CumFreq = FoldList[Plus, 0, Freq];
ListPlot[CumFreq, Filling -> Axis]
```

Same thing, with Accumulate[] (new in Mathematica 6):
CumFreq = Accumulate[Freq];
ListPlot[CumFreq, Filling -> Axis]

---

### 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**

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

  (2)

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

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

- **Sampling from textures using local updates**

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

- Set up image arrays and useful functions

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

We can try several types of potentials.
Ising potential

\[ f[x_\_ , s_\_ , n_\_] := \text{If}[\text{Abs}[x] < 0.5, 0, 1]; \]
\[ s0 = 1.; n0 = 5; \]
\[ \text{Plot}[f[x, s0, n0], \{x, -2, 2\}, \text{AxesLabel} \rightarrow \{"!\text\{
SubscriptBox[\text{V}, \text{i}]\}, \text{i}\} - !\text\{
SubscriptBox[\text{V}, \text{j}]\}, \text{j}\} \text\{"}, f]\]

Geman & Geman potential

\[ f[x_\_ , s_\_ , n_\_] := N[\sqrt{\left(\text{Abs}\left[\frac{x}{s}\right]^n / \left(1 + \text{Abs}\left[\frac{x}{s}\right]^n\right)\right)}]; \]
\[ s0 = 0.25.; n0 = 2; \]
\[ \text{Plot}[f[x, s0, n0], \{x, -2, 2\}, \text{PlotRange} \rightarrow \{0, 1\}, \text{AxesLabel} \rightarrow \{"!\text\{
SubscriptBox[\text{V}, \text{i}]\}, \text{i}\} - !\text\{
SubscriptBox[\text{V}, \text{j}]\}, \text{j}\} \text\{"}, f]\]

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

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

```

gibbsdraw[avg_, T_] :=
Module[{},
   temp = Table[gibbspotential[x + 1, avg, T], {x, 0, ngray - 1}];
   temp2 = FoldList[Plus, temp[[1]], temp];
   temp10 = Table[Einset[temp2[[j]], 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}];]
```

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

```
For[iter = 1, iter ≤ 10, iter ++, T = 0.25;
   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]]]; 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

\[
\text{anneal}[\text{iter}_-, T0_, a_] := T0 / \left( a \left( \frac{1}{a} + \log(\text{iter}) \right) \right);
\]

\[
\text{Plot}\left[ \text{anneal}[\text{iter}, T0, 1], \{\text{iter}, 1, 20\}, \text{PlotRange} \to \{0, 2\}, \text{AxesOrigin} \to \{0, 0\} \right]
\]

- 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.

\[
brown = N[\text{Table}[\text{RandomInteger}[[1, ngray]], \{i, 1, \text{size}\}, \{i, 1, \text{size}\}]];
\]

\[
gg2 = \text{Dynamic}[\text{ArrayPlot}[\text{brown}, \text{Mesh} \to \text{False},
\quad \text{ColorFunction} \to \text{ColorData}[^\text{SouthwestColors}^],
\quad \text{PlotRange} \to \{1, ngray\}]]
\]

\[
\text{ArrayPlot}\left[ \text{brown}, \text{Mesh} \to \text{False},
\quad \text{ColorFunction} \to \text{ColorDataFunction}[[0, 1], \text{SouthwestColors}],
\quad \text{PlotRange} \to \{1, ngray\} \right]
\]

\[
\text{For}[\text{iter} = 1, \text{iter} \leq 10, \text{iter}++, T = \text{anneal}[\text{iter}, T0, 1];
\quad \text{For}[j1 = 1, j1 \leq \text{size} \text{ size}, j1++, \{i, j\} = \text{RandomInteger}[[1, \text{size}], 2];
\quad \text{avg} = \{(\text{brown}[\text{next}[i], j]), (\text{brown}[i, \text{next}[j]], \text{brown}[i, \text{previous}[j]],
\quad \text{brown}[\text{previous}[i], j]); \text{brown}[i, j] = \text{gibbsdraw}[\text{avg}, T][1]]]];
\quad \text{ArrayPlot}\left[ \text{brown}, \text{Mesh} \to \text{False},
\quad \text{ColorFunction} \to \text{ColorData}[^\text{SouthwestColors}^],
\quad \text{PlotRange} \to \{1, ngray\} \right];
\]

\[
\]
References


