Introduction to Neural Networks

More sampling

Overview

Today

More sampling, leading to using Markov Chain Monte Carlo (MCMC) applications to inference on graphs

Why sampling?

Boltzmann machine’s update rule review

Recall the Boltzmann machine was a fully connected network consisting of binary units. The update rule was to set

$V_i = 1$ with probability p_i

where: $p_i = p(\Delta E_i) = \frac{1}{1+e^{-\Delta E_i / T}}$, where $\Delta E_i = \sum_j T_{ij} V_j$

T (without the indices) is analogous to temperature in thermodynamics. For T=1, this update rule should by now be quite familiar. If we add a bias term, replace V_j and T_{ij} with vector x and matrix w, it is the same expression that we saw logistic regression, and for the simple probabilistic model of neural spike generation (Lecture 18) where the probability of a neuron firing is:

$$p(y_i = 1 \mid x) = \frac{1}{1+e^{-(w.x+b)}}$$

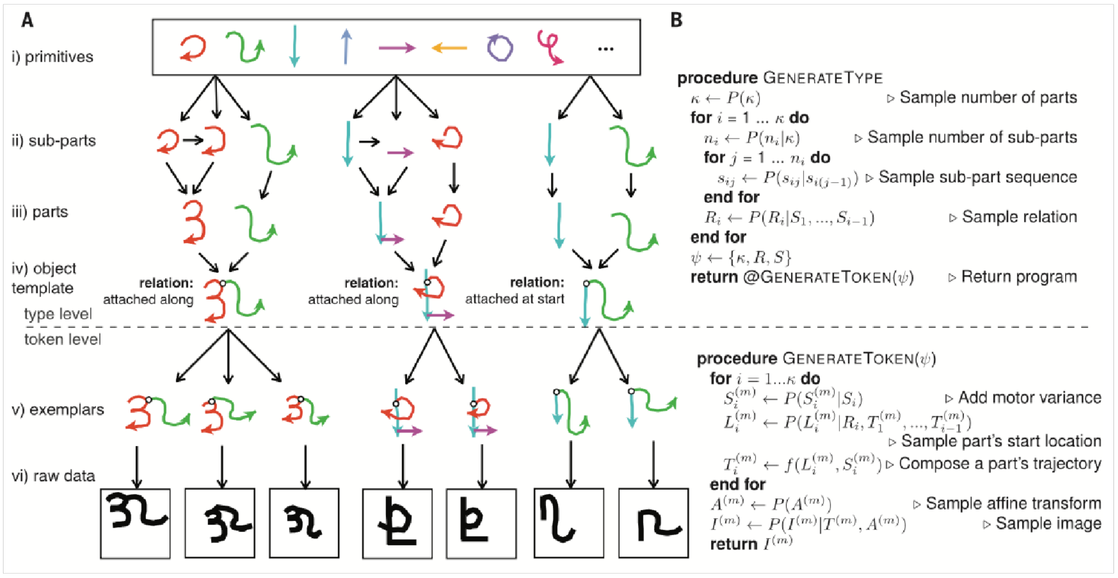
To implement the update rule for the Boltzmann machine, we draw a uniformly distributed number between 0 and 1, and if that number is less than $p_i = p(\Delta E_i)$ we set the output of the neuron, V_i, to 1; otherwise, set it to zero:

This is an example of the use of Monte Carlo sampling to do inference. But sampling can be applied in many more ways, for example to learn the weights given training data, and more ways described below.

Earlier we also saw how to sample in one dimension. But sampling gets much harder as the dimensionality increases. To further motivate sampling, consider several applications:

Pattern synthesis

Draw samples from a generative model which might be complex. E.g. involving lots of lateral interactions as in an undirected graph. Or involving a top-down directed graph that specifies how causes create samples. This can be quite useful in vision to check whether one's model is capturing the regularities seen in the data. Sampling can be used to generate complex structured stimuli to test theories of human learning and recognition (e.g. Lake, B. M., Salakhutdinov, R., & Tenenbaum, J. B., 2015), and in principal, to characterize response properties of neurons.



```
procedure GENERATETYPE
  $\kappa \leftarrow P(\kappa)$                          ▷ Sample number of parts
  for $i = 1 \ldots \kappa$ do
    $n_i \leftarrow P(n_i|\kappa)$                ▷ Sample number of sub-parts
    for $j = 1 \ldots n_i$ do
      $s_{ij} \leftarrow P(s_{ij}|s_{i(j-1)})$ ▷ Sample sub-part sequence
    end for
    $R_i \leftarrow P(R_i|S_1, \ldots, S_{i-1})$          ▷ Sample relation
  end for
  $\psi \leftarrow \{\kappa, R, S\}$
  return @GENERATETOKEN($\psi$)          ▷ Return program

procedure GENERATETOKEN($\psi$)
  for $i = 1 \ldots \kappa$ do
    $S_i^{(m)} \leftarrow P(S_i^{(m)}|S_i)$                ▷ Add motor variance
    $L_i^{(m)} \leftarrow P(L_i^{(m)}|R_i, T_1^{(m)}, \ldots, T_{i-1}^{(m)})$
                                   ▷ Sample part's start location
    $T_i^{(m)} \leftarrow f(L_i^{(m)}, S_i^{(m)})$ ▷ Compose a part's trajectory
  end for
  $A^{(m)} \leftarrow P(A^{(m)})$              ▷ Sample affine transform
  $I^{(m)} \leftarrow P(I^{(m)}|T^{(m)}, A^{(m)})$            ▷ Sample image
  return $I^{(m)}$
```

Figure from Lake et al., 2015.

Texture synthesis, demonstrated below using Gibbs sampling, is an example of sampling given an undirected graph.

Learning

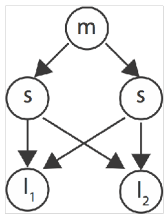
Given a model, one use stochastically generated (synthesized) samples in order to compare with data so as to adjust parameters (i.e. weights). We saw this in the original Boltzmann machine.

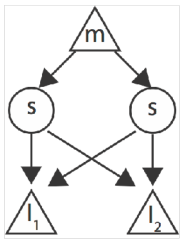
Optimization

We already saw an example using the Boltzmann machine to do optimization, where we had multiple constraints (the stereo problem). For distributions with multiple modes, one may need annealing.

Inference

Even with low-dimensional architectures (graphical models), given non-gaussian distributions, the math for determining simple messages of the sort we saw in our belief propagation example can be too difficult, and Monte Carlo simulations are required. For example, it is one thing to posit and write down the joint distribution corresponding to this graph:





but quite another to calculate needed values for unknown variables, with others fixed, and still others unwanted. Recall that unwanted variables need to be integrated out in order to calculate ideal performance. Practical solutions to solving these kinds of problems is what we are aiming for here and in the later material on MCMC sampling.

Integration (summing over) in very high dimensions

Integration or summing over many variables is a recurring problem. Sampling provides a method to do numerical integration.

Summing over many variables is required to find the *normalization constant* for a distribution. In equation 1), the normalization constant is the discrete estimate of the 3D volume of p:

$$Z = \sum_{x_3=1}^{N_3} \sum_{x_2=1}^{N_2} \sum_{x_1=1}^{N_1} p(x_1, x_2, x_3) \qquad (1)$$

You may want to calculate the *expectation of a function*, f(x) of a random vector, e.g. when you calculate the mean, f(x) = x. You have an expression for the joint distribution, say p(x), where x = $(x_1,x_2,x_3,\ldots,x_N)$.

$$E[f(x)] = \sum_{x_1}\sum_{x_2}\sum_{x_3}\ldots\sum_{x_M} f(x_1, x_2, x_3, \ldots, x_M)\, p(x_1, x_2, x_3, \ldots, x_M)$$

Marginalization also requires summing over many variables

$$p(x_1 \mid x_2, x_3, \ldots, x_M) = \sum_{x_2}\sum_{x_3}\ldots\sum_{x_M} p(x_1, x_2, x_3, \ldots, x_M)$$

Even if one had formulas for the functions, M can be very, very big. Summing over gets computationally expensive to the point of intractability the more variables we need to sum over. If we have $p(x_1, x_2, x_3, \ldots, x_M)$ and we would like to sum over M variables, each of which can take on N values. We have to calculate and sum N^M values. For example, this would be impossible for even a small image representation.

Also, there is *model selection*, where we want to compare the probability of two models given some data, we need to integrate out the parameters, θ, that determine the models, m. For example, one model might be a quadratic function with three parameters, and another the sum of three sinusoids, whose parameters are the coefficients. Think of the parameters as values to be estimated (or explanations of the data) under model m. Bayes rule says that given data x:

$$p(\theta \mid x, m) = \frac{p(x \mid \theta, m)\, p(\theta \mid m)}{p(x \mid m)}$$

The denominator is called the “evidence”. Calculating it is the above normalization problem. For a fixed model, we usually don't need to know the denominator. But if we want to decide which model is better we need to evaluate p(x|m) for the various m's. And this may not be easy:

$$p(x \mid m) = \sum_{\theta} p(x, \theta \mid m) = \sum_{\theta} p(x \mid \theta, m)\, p(\theta \mid m)$$

As above, $\theta = (\theta_1, \theta_2, \ldots,)$ may have many dimensions. Nested sampling is one method to do model selection.

Sampling as a metaphor for bottom-up, top-down neural processing

One can either fix causes and draw samples of the data (top-down, "dreaming"), or fix the data and draw samples of possible causes (inference).

Neural sampling as the basis for representations and decisions.

See Sundareswara & Schrater (2008), Battaglia et al. (2011); Vul et al. (2012).

► 1. Show that if you have to sum over M indices (e.g. M = 3 in equation 1) above), each ranging from 1 to N, you would have to do ~ N^M additions.

If you needed to calculate a marginal distribution over an image representation, say of really tiny pictures, 8 x 8 pixels, and each pixel could take on 256 gray levels (i.e. 8 bit pixels) how many sums would you have to do? Just for comparison, what can a 64 bit register on your computer count up to?

Sampling

What is Monte Carlo?

Let's consider the Monte Carlo approach to the problem of calculating the area under a continuous function, or the volume of a surface, etc.. In practice, approximations are estimated through numerical integration, and one method of numerical integration is to use random sampling. For example, given a line that divides a table into two unequal parts, one could estimate the relative proportions of one of the areas by randomly throwing lots of balls on to the table and recording the proportion that settled in the area of interest.

More generally, we can approximate a density p(x) with a weighted set of samples from a potentially very high dimensional space by:

We will see below, that in contrast to deterministic integration, Monte Carlo integration positions samples in regions of high probability.

With a large number of random samples $\{x^i\}$ drawn from p(x), we can approximate p(x) as:

$$p_N(x) \sim \frac{1}{N} \sum_{i=1}^{N} \delta\left(x^i - x\right)$$

By the definition of expectation we have:

$$E(f) = \int f(x)\, p(x)\, dx \simeq \int f(x)\, p_N(x)\, dx =$$

$$\int f(x) \frac{1}{N} \sum_{i=1}^{N} \delta\left(x^i - x\right) dx = \frac{1}{N} \sum_{i=1}^{N} \int f(x)\, \delta\left(x^i - x\right) dx = \frac{1}{N} \sum_{i=1}^{N} f\left(x^i\right)$$

A familiar case is f(x) = x, where $\frac{1}{N} \sum_{i=1}^{N} x^i$ is the approximation of the mean.

▶ 2. Derive the above relationship starting with a weighted sum of delta functions:

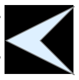
$$p_N(x) \sim \sum_{i=1}^{N} \delta\left(x^i - x\right) p\left(x^i\right)$$

Using the definition of the delta function, one can show that we can approximate high-dimensional integrals with computable sums. Suppose we want to compute the expectation of a function f(x):

$$S_N(f) = \sum_{i=1}^{N} p\left(x^i\right) f\left(x^i\right) \sim \frac{1}{N} \sum_{i=1}^{N} f\left(x^i\right) \xrightarrow[N \to \infty]{a.s.} S(f) = \int f(x)\, p(x)\, dx \qquad (2)$$

"a.s." is a mathematical notion of "almost surely" converges as N approaches infinity

▶ 3. Use Monte Carlo sampling to calculate the area of the white area in the figure below. Monte Carlo is one option in routines to do numerical integration.

```
pannter = ImageData[Image[ColorConvert[
```

```
, GrayLevel], "Bit"]];
```

```
n = 100 000;
Total[Table[If[pannter[[RandomInteger[{1, Dimensions[pannter][[1]]}],
        RandomInteger[{1, Dimensions[pannter][[2]]}]]] == 1, 1, 0], {n}]] / n // N
```

Compare with a direct count of bright points, 1s, with dark points (0s):

```
Count[Flatten[pannter], 0];
N[Count[Flatten[pannter], 1] / Length[Flatten[pannter]]]
```

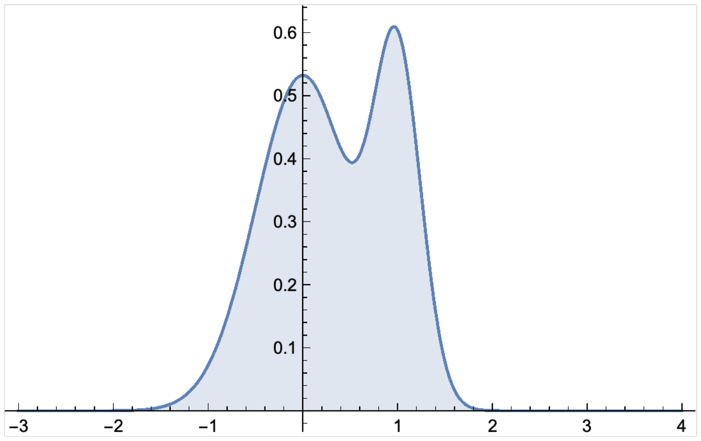
Let's look at some univariate examples to illustrate concepts that can be carried to more complicated cases. Of course to draw samples, we could use the *inverse CDF* method in an earlier lecture, but let's try something different, something that is more broadly applicable, and extensible to high dimensions.

Assume that we want to draw samples from a target distribution, p(x) that is hard or impossible to directly sample from. But we have a method to draw samples from another distribution that is easy to sample from, called the *proposal distribution,* q(x). Given a sample from the proposal distribution, we can calculate its probability from the target distribution and then keep the sample if it is a true draw, but reject it if not. How do we know if it is representative of a true draw from the target distribution?

Here's a concrete example. We want to draw samples from this target distribution p$\mathcal{D}$:

```
Clear[p𝒟, q𝒟]
```

```
p𝒟 = MixtureDistribution[{2, 1},
   {NormalDistribution[0, 1 / 2], NormalDistribution[1, 1 / 4]}];
Plot[PDF[p𝒟, x], {x, -3, 4}, Filling → Axis]
```



but the only algorithm we have to draw samples from is this one:

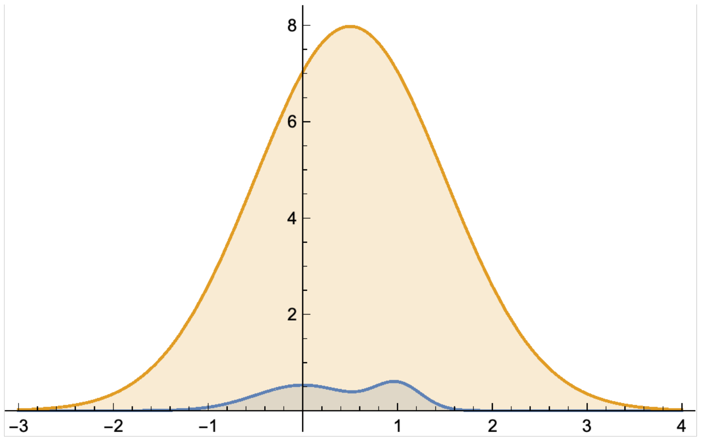
```
q𝒟 = NormalDistribution[.5, 1]
NormalDistribution[0.5, 1]
```

How can we do it? One way is to use *rejection sampling*.

Rejection sampling

The idea is to first find M such that p(x) < M × q(x), i.e. that PDF[p𝒟,x] < M* PDF[q𝒟,x]. Graphically, we want M × q(x) to be higher than p(x) over the range where p(x)>0.

```
Plot[{PDF[p𝒟, x], 20 * PDF[q𝒟, x]}, {x, -3, 4}, Filling → Axis]
```



Now draw a sample x' from the proposal distribution, q𝒟, and another sample, u, from the uniform distribution (between 0 and 1).

Only count the sample x’, if u*M*PDF[q𝒟,x’] < PDF[p𝒟,x’].

In effect, we are drawing a sample from a uniform distribution between 0 and M × q(x), and testing if it’s probability is below p(x).

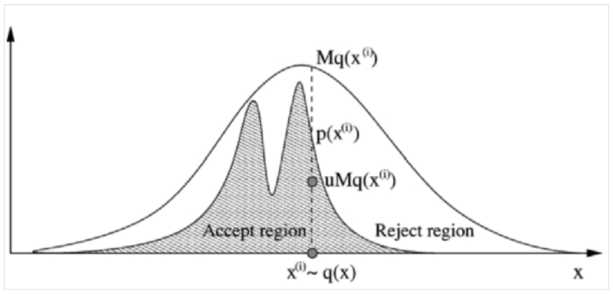
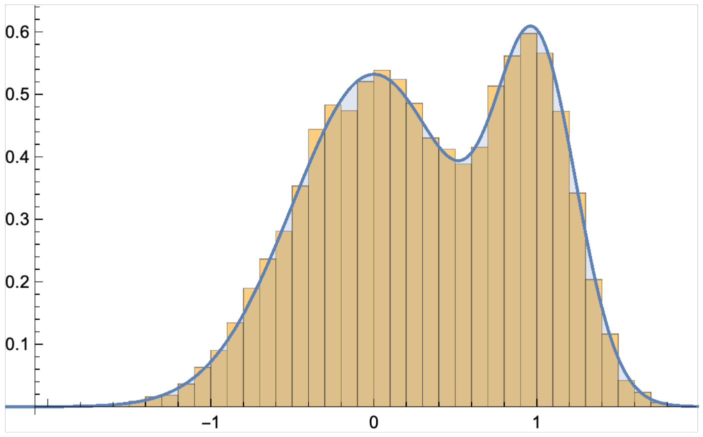


Figure from Andrieu, C., De Freitas, N., Doucet, A., & Jordan, M. I. (2003).

```
x := RandomVariate[q𝒟 ];
u := RandomVariate[UniformDistribution[]];
M = 2;
samples =
  Table[If[(u < PDF[p𝒟, x0 = x] / (M * PDF[q𝒟, x0]) && Abs[x] < 3), x0, Null], {50 000}];
```

```
gH = Histogram[samples, 48, "PDF"];
```

```
gp𝒟 = Plot[{PDF[p𝒟, x]}, {x, -3, 4}, Filling → Axis];
Show[{gH, gp𝒟}]
```



▶ 4. Try M = 100. You can see that the rejection sampling is most efficient if M times the proposal distribution isn’t too much higher than the target distribution over most of the range.

Rejection sampling becomes increasingly inefficient as the dimensionality increases when too many samples get rejected.

Importance “sampling”

Perhaps should be called “importance approximation”, because we don’t use it to directly produce samples from a target distribution, but use samples from a proposal distribution to estimate expectations with respect to the target distribution. If we sample x from p(x):

$$E[f] = \int f(x)\, p(x)\, dx \sim \frac{1}{N} \sum_{i}^{N} f\left(x^{i}\right)$$

But if we sample x from q(x), then the expectation of f(x)p(x)/q(x) (with respect to sampling from q) is :

$$E[f] = \int \left(f(x) \frac{p(x)}{q(x)}\right) q(x)\, dx = \int (f(x)\, w(x))\, q(x)\, dx \sim \frac{1}{N} \sum_{i}^{N} f\left(x^{i}\right) w\left(x^{i}\right)$$

Lets look at an example where we find the average of a random variable from target distribution p𝒟, but we don’t sample from it directly--as before, we sample from a proposal distribution, q𝒟. In this case, let f(x) = x.

```
q𝒟 = NormalDistribution[0, 3];
p𝒟 = NormalDistribution[3, 1];
```

```
u := RandomVariate[q𝒟];
weights = Table[{x0 = u, PDF[p𝒟, x0] / PDF[q𝒟, x0]}, {1000}];
Mean[weights[[ ;; , 1]] * weights[[ ;; , 2]]]
```

```
0.284649
```

Compare with the mean when we are allowed to directly sample from p𝒟:

```
x := RandomVariate[p𝒟 ];
xsamples = Table[{x}, {1000}];
```

```
Mean[xsamples]
```

```
{0.302341}
```

What is the importance of importance sampling?

Suppose you need to sample from a truncated normal distribution with support from 0 to 1--call it p_T(x). You could draw samples from the normal and throw out ones outside of the (0,1) range. This could waste many samples.

Alternatively, you could use the uniform distribution over (0,1) as the proposal distribution, and weight each sample by the truncated normal, $p_T(x^i)/\mathcal{U}[0,1] = p_T(x^i)/1$. You can show that the mean is:

$$(1/N) \sum_{i}^{N} x^{i}\, p_T\left(x^{i}\right)$$

And every sample gets used.

An example of using importance sampling to estimate the mean of a truncated normal

```
𝒟trunc = TruncatedDistribution[{0, 1}, NormalDistribution[]];
u := RandomVariate[UniformDistribution[{0, 1}]];
```

In this case Mathematica provides an analytic formula for the mean:

```
Mean[𝒟trunc]
N[%]
```

$$\frac{\left(-1+\sqrt{e}\right)\sqrt{\frac{2}{e\pi}}}{\mathrm{Erf}\left[\frac{1}{\sqrt{2}}\right]}$$

```
0.459862
```

But we can also estimate the mean using importance sampling:

```
Table[{x0 = u , x0 * PDF[𝒟trunc, x0]}, {10 000}];
Mean[%]
```

```
{0.499209, 0.459676}
```

Importance sampling can be used to reduce estimator variance.

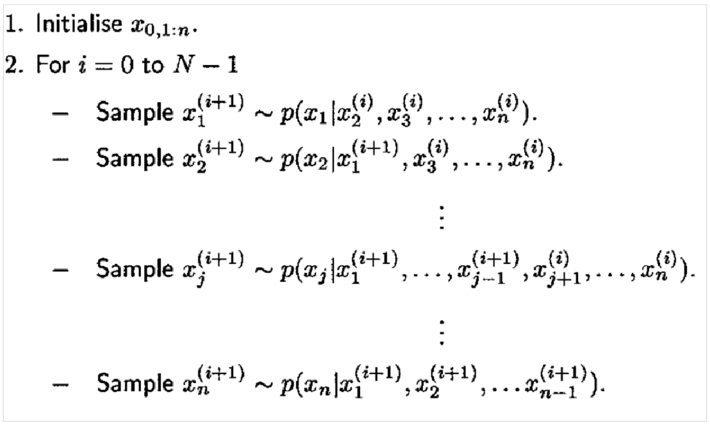
- 5. Estimate the variance of the target distribution of the above truncated normal using importance sampling
- 6. Show that we don’t need to know the normalizing constants for either the target or proposal distributions.

Gibbs sampling

Sometimes we can calculate the conditionals:

$$p\left(x_j \mid x_{-j}\right) = p\left(x_j \mid x_1, x_2, \ldots x_{j-1}, x_{j+1}, \ldots . x_N\right)$$

where “x_{-j}” means all x_i such that $i \neq j$. In this case, we can start with some initial condition, and proceed by replacing each value by a sample drawn from the conditionals/ As we repeatedly iterate through all the nodes, the samples approach the distribution of $p\left(x_1, x_2, \ldots x_j, \ldots x_N\right)$



```
1. Initialise $x_{0,1:n}$.
2. For $i = 0$ to $N-1$
    – Sample $x_1^{(i+1)} \sim p(x_1|x_2^{(i)}, x_3^{(i)}, \ldots, x_n^{(i)})$.
    – Sample $x_2^{(i+1)} \sim p(x_2|x_1^{(i+1)}, x_3^{(i)}, \ldots, x_n^{(i)})$.
                              ⋮
    – Sample $x_j^{(i+1)} \sim p(x_j|x_1^{(i+1)}, \ldots, x_{j-1}^{(i+1)}, x_{j+1}^{(i)}, \ldots, x_n^{(i)})$.
                              ⋮
    – Sample $x_n^{(i+1)} \sim p(x_n|x_1^{(i+1)}, x_2^{(i+1)}, \ldots x_{n-1}^{(i+1)})$.
```

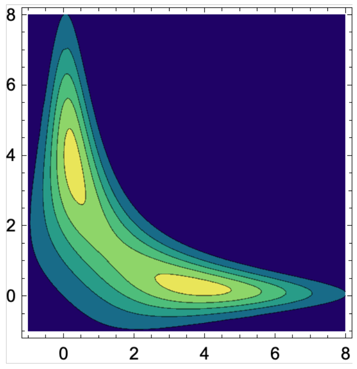
From Andrieu et al.

The iterations don't have to be in sequence, but can be chosen at random, as we do below in the texture synthesis example. Given a Markov property, the conditionals can be simple e.g. $p(x_j \mid x_{-j}) = p(x_j \mid x_{j-1})$ and Gibbs sampling becomes practical in high-dimensional spaces. For Gibbs sampling, we need the probabilities, i.e. normalized. This is illustrated in the texture synthesis demo below

Gibbs sampling simple bivariate: demo

This demo takes advantage of a "weird" bivariate distribution, weird because its conditionals are gaussian, which makes for a simple gibbs sampler.

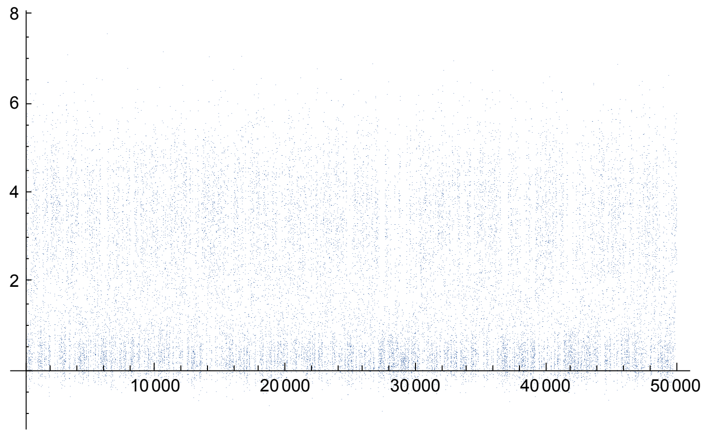
```
weird2ddist[x_, y_] := Exp[- (x * x * y * y + x * x + y * y - 8 * x - 8 * y) / 2];
gcontour = ContourPlot[weird2ddist[x, y] ^ .2, {x, -1, 8},
  {y, -1, 8}, ColorFunction → "BlueGreenYellow", ImageSize → Small]
```



```
n = 50 000;
x = ConstantArray[0.0, n];
y = ConstantArray[0.0, n];

x[[1]] = 1.;
y[[1]] = 6.;
sig[z_, i_] := Sqrt[1. / (1. + z[[i]] * z[[i]])];
mu[z_, i_] := 4. / (1. + z[[i]] * z[[i]]);

For[i = 2, i < n - 1, i = i + 2,
 sigx = sig[y, i - 1];
 mux = mu[y, i - 1];

 x[[i]] = RandomVariate[NormalDistribution[mux, sigx]];
 y[[i]] = y[[i - 1]];

 sigy = sig[x, i];
 muy = mu[x, i];

 y[[i + 1]] = RandomVariate[NormalDistribution[muy, sigy]];
 x[[i + 1]] = x[[i]];(**)

]
```

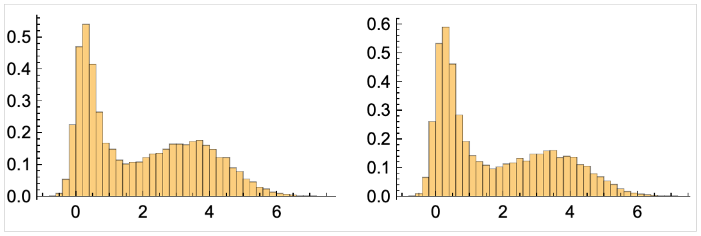
Plot the evolution of the x sample values as a function of the iteration (or time step):

```
ListPlot[x]
```



Plot up the marginal distribution estimates:

```
GraphicsRow[{Histogram[x, 50, "PDF"], Histogram[y, 50, "PDF"]}]
```



Estimate the means of the marginals. E.g. for x:

```
x
```

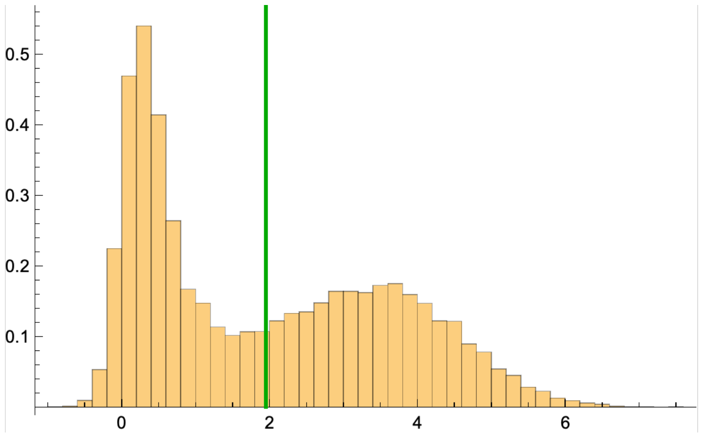
```
{1., 0.225781, 0.225781, -0.137131, -0.137131,
 ··· 49 990 ···, 4.11481, 4.11481, 3.5323, 3.5323, 0.}

large output    show less    show more    show all    set size limit...
```

```
Mean[x]
```

```
1.94803
```

```
{bins, heights} = HistogramList[x, 50, "PDF"];
maxFreq = N[Max[heights]]
Histogram[x, {bins}, heights &,
 Epilog → {{Thick, Darker[Green], Line[{{Mean[x], 0}, {Mean[x], maxFreq + 2}}]}}]
```

```
0.5402
```



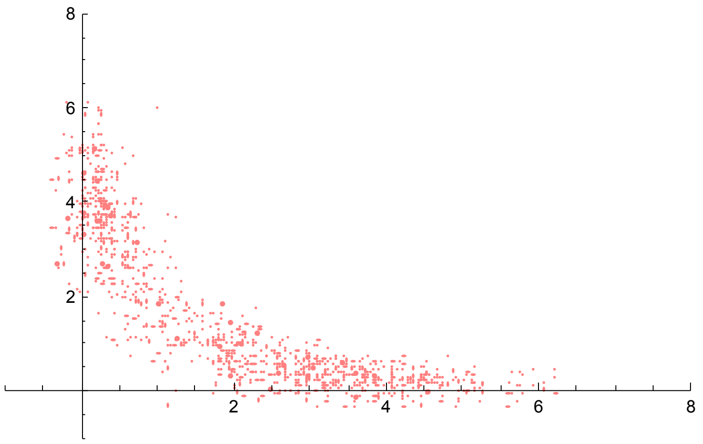
And for both:

```
{Mean[x], Mean[y]}
```

```
{1.94803, 1.78225}
```

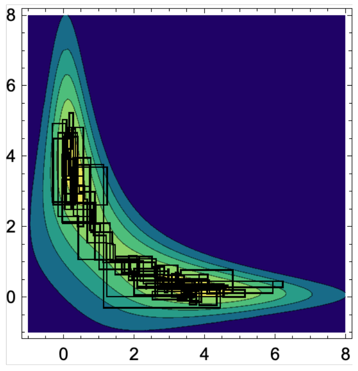
Here’s a scatter plot of the samples:

```
traj = Transpose[Join[{x[[1 ;; 1000]], y[[1 ;; 1000]]}]];
gtraj0 = ListPlot[traj, Joined → False,
  PlotRange → {{-1, 8}, {-1, 8}}, PlotStyle → Pink, Axes → True]
```



And finally, visualize how the samples jump around during the last 300 iterations.

```
traj2 = Take[traj, {Dimensions[traj][[1]] - 300, Dimensions[traj][[1]], 1}];
gtraj = ListPlot[traj2, Joined → True, PlotRange → {{-1, 8}, {-1, 8}},
   PlotStyle → {Black, Thickness[.005]}, Axes → False];
Show[{gcontour, gtraj}]
```



Metropolis-Hastings

We’ve gone from special cases to more general. It can be shown that Gibbs Sampling, for example, is a special case of a more general class of samplers called Metropolis-Hastings.

As before assume we have target distribution, p(x) that may only be specified up to a normalizing constant. We can't directly sample from it,
but we have a proposal distribution, q(x) that we can sample from. Given a possible sample we can evaluate its target probability up to a normalizing constant. Further, we can calculate the conditional probabilities $q(x' \mid x'')$.

Here’s a summary of the algorithm:

```
Initialize $x^0$ to some arbitrary value
for $i = 0$ to $n-1$
   sample $u \sim \mathcal{U}[0, 1]$
   sample $x^* \sim q(x^* \mid x^i)$
   if $u < \min\left\{1, \frac{p(x^*)\, q(x^i \mid x^*)}{p(x^i)\, q(x^* \mid x^i)}\right\}$
      $x^{i+1} = x^*$,
   otherwise
      $x^{i+1} = x^i$
```

$\mathcal{A}(x^i, x^*) = \min\left\{1, \frac{p(x^*)\, q(x^i \mid x^*)}{p(x^i)\, q(x^* \mid x^i)}\right\}$ is called the acceptance function.

The acceptance rule is equivalent to setting $x^{i+1} = x^*$ with probability $\mathcal{A}(x^i, x^*)$.

Demonstration

```
Clear[x, x1, xx, uu, x0, u, p𝒟, q𝒟, A, A0];
```

The first function, p𝒟 defines the target distribution, p(x). Recall that any function, constant × p(x), will also work.

```
p𝒟 = MixtureDistribution[{2, 1},
    {NormalDistribution[], NormalDistribution[2, 1 / 4]}];
gp𝒟 = Plot[{PDF[p𝒟, x]}, {x, -3, 3}, Filling → Axis];
```

This next function below specifies the proposal distribution. We can sample from it and given a possible sample value as input, we
can calculate its value.

```
q𝒟[x_] := NormalDistribution[x, 4];
```

```
x0 = -10.0; (*initial value*)
uu := RandomVariate[UniformDistribution[]];
```

We could define the “acceptance function” $\mathcal{A}$ = A0 as:

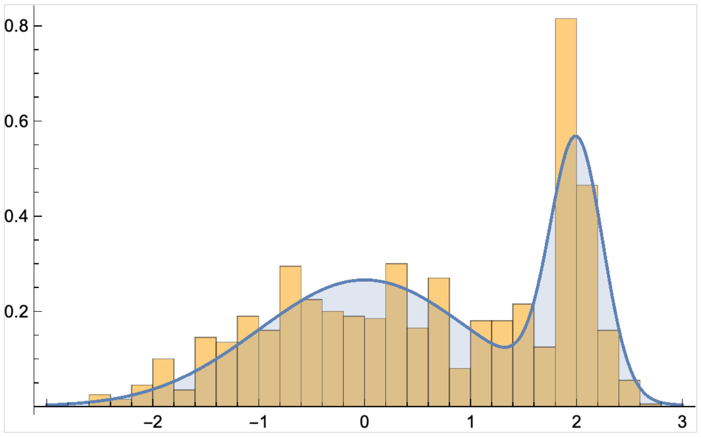
```
A0[x1_, x0_] :=
  Min[1.0, (PDF[p𝒟, x1] * PDF[q𝒟[x0], x1]) / (PDF[p𝒟, x0] * PDF[q𝒟[x1], x0])];
```

The above acceptance function here is the general case we introduced above-- it puts the Hastings in Metropolis-Hastings.

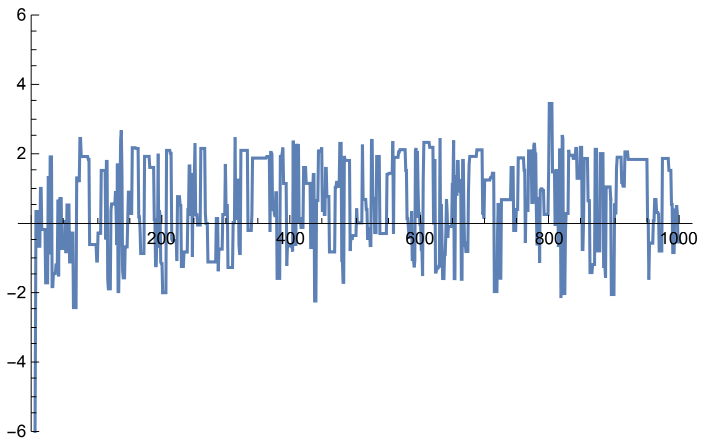
But if you examine the above **A0[ ]** function, you’ll see that the proposal distribution is symmetric (you can swap x1 and x0, and the computed
value stays the same). So in our case using a symmetric normal distribution for proposal sampling, we can simplify the acceptance function:

$$\mathcal{A}\left(x^i, x^*\right) = \min\left\{1, \frac{p\left(x^*\right)}{p\left(x^i\right)}\right\}$$

```
A[x1_, x0_] := Min[1.0, (PDF[p𝒟, x1]) / (PDF[p𝒟, x0])];

n = 1000;
samples = ConstantArray[0.0, n];
samples[[1]] = x0;
For[i = 1, i < n, i++,
      x1 = RandomVariate[q𝒟[samples[[i]]]];
      Pause[.01];

  If[uu < A[x1, samples[[i]]], samples[[i + 1]] = x1, samples[[i + 1]] = samples[[i]]]];

(*gH=Dynamic[Histogram[samples,48,"PDF",PlotRange→{{-3,3},{0,.8}}]]*)

gH =
 Dynamic[Show[{Histogram[samples, 48, "PDF", PlotRange → {{-3, 3}, {0, .8}}], gp𝒟}]]
```

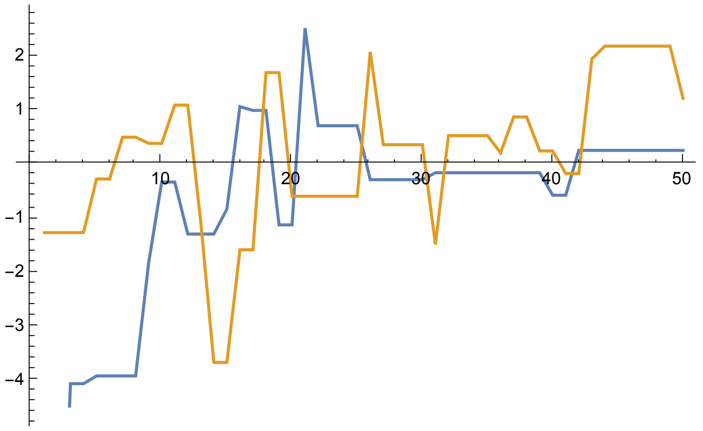


```
Dynamic[ListPlot[samples, Joined → True, PlotRange → {-6, 6}]]
```



This is sometimes called random-walk Metropolis. Let’s compare the first 50 samples (blue) with the last 50 (tan).

```
ListLinePlot[{samples[[1 ;; 50]], Take[samples, -50]}]
```



The samples are eventually representative of $x^i \sim p(x)$. Because it takes a while for the samples to settle, there is an initial “burn in” period--these samples are typically thrown out. Also note because the acceptance rule is conditional on the previous sample, that the samples are not independent.

The original Metropolis et al. paper and simulated annealing papers are among the top 100 most cited papers according to Web of Science and Nature: http://www.nature.com/news/the-top-100-paper-s-1.16224

Metropolis, N., Rosenbluth, A. W., Rosenbluth, M. N., Teller, A. H. & Teller, E. , (1953) Equation of state calculations by fast computing machines. J. Chem. Phys. 21,1087–1092.

Kirkpatrick. S., Gelatt, C. D. & Vecchi, M. P. (1983) Optimization by simulated annealing. Science, 220, 671–680.

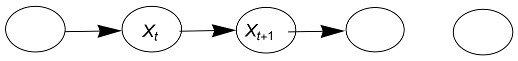
To get a sense of their scientific impact, look up the above papers in scholar.google. Also, for Gibbs sampling, type in geman.

So what is a Markov Chain?

We’ve demonstrated several Monte Carlo methods of drawing samples, but haven’t explained the “Markov Chain” bit.

Gibbs Sampling and the Metropolis-Hastings methods are both examples of Markov Chain Monte Carlo. Where does the Markov Chain come in?

As a function of time, the iterations determine a sequence of states that have the Markov Property characterized by the graph:

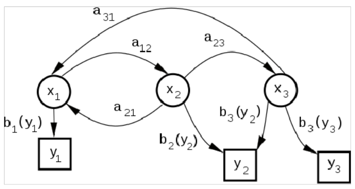


Note that a draw from the proposal distributions only depended on the previously drawn value:

$$p\left(x^i \mid x^{i-1}, \ldots, x^1\right) = T\left(x^i \mid x^{i-1}\right)$$

$T\left(x^i \mid x^{i-1}\right)$ is a transition matrix, which can also be represented by a transition graph.

Here’s an example from the wiki page:



With this condition and the assumption that the transition matrix T stays fixed, the stochastic process is a Markov Chain. The sequence of draws we simulated above are the samples from Markov Chains.

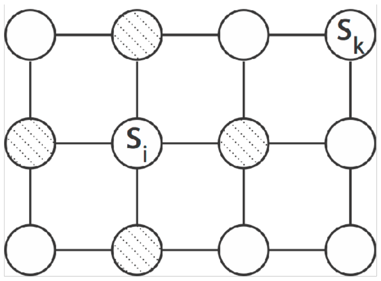
Under certain "regularity" conditions, with large enough i, the evolution of samples x^i will come to reflect the distribution of an "invariant" or "stationary distribution", which with the right proposal design corresponds to our desired target distribution. Once we have an estimate of that, represented by many samples, we can calculate various statistics on it, such as mean values of marginals. Because it can take many iterations before the samples are representative of draws from the target distribution, one usually drops early samples (from the burn in period) and calculates the mean (or other statistics) based on a criterion number of later samples.

Although we don't describe the details of the convergence conditions, they can be illustrated with searches on the web (Andriew et al., 2003). A transition takes us from one node (link) to the next. One would like to avoid getting trapped in loops and one would like to be able to access any link in the net. These two constraints correspond to two conditions, *aperiodicity* and *irreducibilty* which in the case of Markov Chains are required for the chain to evolve such that samples are drawn from the target distribution. Google's original PageRank search algorithm used a transition matrix with added noise to ensure aperiodicity and irreducibility.

Gibbs sampling texture synthesis demo

Let’s look at an application to pattern synthesis. We’ve already looked at the problem of visual interpolation. This was an example of a more general principle of perception in which similar features reinforce each other, providing the basis for grouping. We discuss this more later when we talk about lateral organization in neural architectures for vision.

Grouping similar features can be represented by an undirected graph:



Suppose the nodes represent pixel values over some probability model of images, such as all the images of “grass”. In general, the probability of activation, s_i, of node i will depend on the neighbors j in some neighborhood of i: $j \in N_i$. s_i. A Markov Random Field (MRF) has the following property:

$p\left(s_i \mid s_j;\ \text{all } j \neq i\right) = p\left(s_i \mid s_j;\ j \in N_i\right)$

In other words, to draw samples from $p\ (s_i \mid \text{given all the other pixels})$, it is sufficient to draw from $p\ (s_i \mid \text{in a neighborhood of } i)$. This locality restriction makes it tractable to calculate marginals and use Gibbs sampling.

Texture perception relies in part on local regularities in images: MRFs

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).

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.

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)

Often it isn’t important to be sure that one has a “true sample”, and other techniques can be used to produce random patterns. E.g. one can first draw sample of white noise, and then iteratively adjust the texture to match statistics of the model, e.g. http://www.cns.nyu.edu/~lcv/texture/

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.

We drew the analogy to dreaming or conscious imagery. A potentially related phenomenon is hypnagogic imagery.

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:

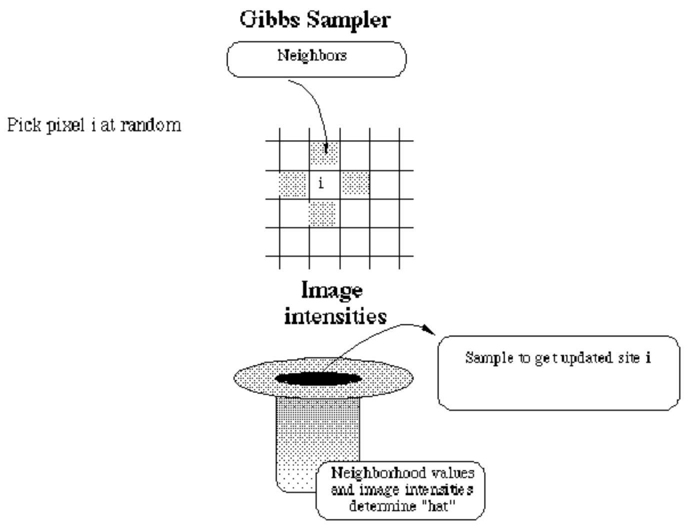

Gibbs Sampler
Neighbors
Pick pixel i at random
i
Image intensities
Sample to get updated site i
Neighborhood values and image intensities determine "hat"

Figure from Kersten (1991)

Demo

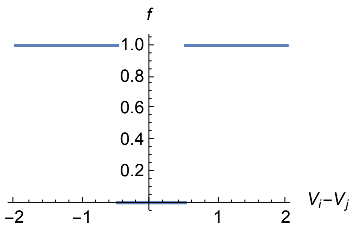
Set up image arrays and useful functions

```
size = 64; 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;
```

We can try several types of potentials.

Ising potential

```
Clear[f];
f[x_, s_, n_] := If[Abs[x] < 0.5, 0, 1];
s0 = 1.`; n0 = 5;
Plot[f[x, s0, n0], {x, -2, 2}, AxesLabel →
  {"\!\(\*SubscriptBox[\(V\), \(i\)]\)-\!\(\*SubscriptBox[\(V\), \(j\)]\)", f}]
```

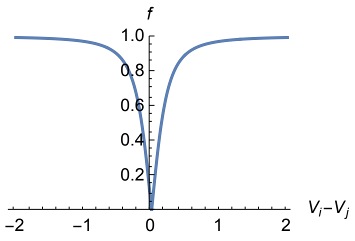


Geman & Geman potential

```
Clear[f];
```

$$f[x\_, s\_, n\_] := N\left[\sqrt{\left(\text{Abs}\left[\frac{x}{s}\right]^n \Big/ \left(1 + \text{Abs}\left[\frac{x}{s}\right]^n\right)\right)}\right];$$

```
s0 = 0.25`; n0 = 2;
Plot[f[x, s0, n0], {x, -2, 2}, PlotRange → {0, 1}, AxesLabel →
  {"\!\(\*SubscriptBox[\(V\), \(i\)]\)-\!\(\*SubscriptBox[\(V\), \(j\)]\)", 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. **avg** is a list of the levels at the neighbors of the site i.

```
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]];
```

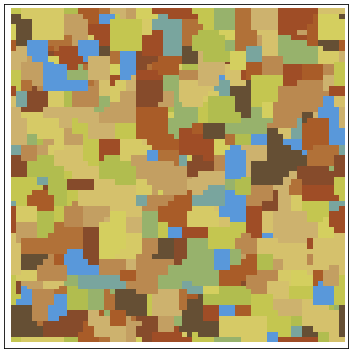
Define a function to draw a single pixel gray-level sample from a conditional distribution determined by pixels in neighborhood

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).

```
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}];];
```

FoldList [] allows us to effectively draw a sample from a normalized distribution using the inverse CDF method.

```
gg = Dynamic[
  ArrayPlot[brown, Mesh → False, ColorFunction → ColorData["SouthwestColors"],
   PlotRange → {1, ngray}, ImageSize → Small]]
```



```
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];
```

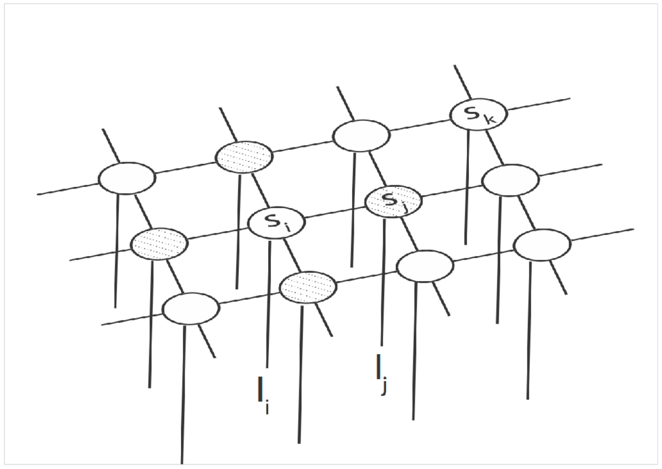
"Drawing" a pattern sample

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?

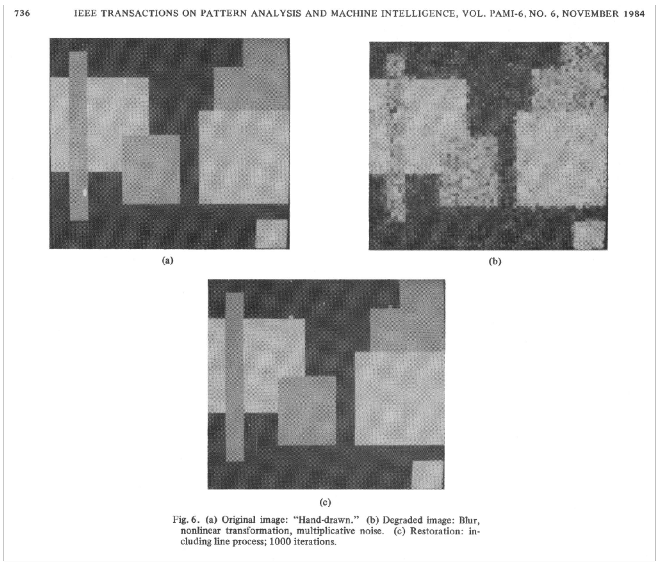
Denoising, and grouping features contingent on lower level input

A MRF can be constrained by image data. For example, suppose the true image is corrupted by noise. Fidelity to input image data can be constrained by prior assumptions about what image pixel relations are typically like--i.e. that image color patterns tend to be piece-wise constant.

Alternatively, the nodes, s, could represent some useful world property to be estimated, such as a material’s intrinsic reflectivity (or surface color).



Here’s a figure from the now classic paper by Geman, S., & Geman, D. (1984) illustrating sharpening and noise removal.


736 IEEE TRANSACTIONS ON PATTERN ANALYSIS AND MACHINE INTELLIGENCE, VOL. PAMI-6, NO. 6, NOVEMBER 1984

(a) (b) (c)

Fig. 6. (a) Original image: "Hand-drawn." (b) Degraded image: Blur, nonlinear transformation, multiplicative noise. (c) Restoration: including line process; 1000 iterations.

Next time

Scientific computing with Python and Jupyter/IPython notebooks. Examples of neural modeling, and Bayesian inference using MCMC sampling.

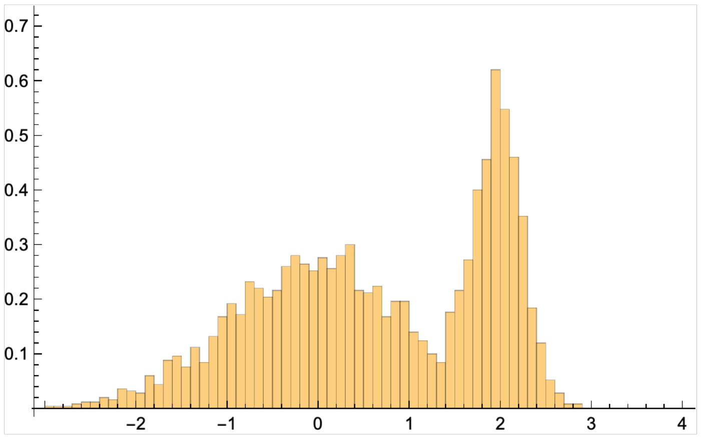
Appendix

Rejection sampling with uniform proposal distribution

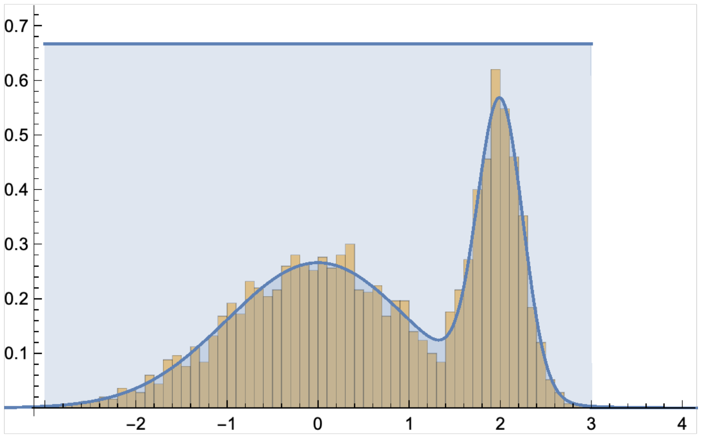
The uniform distribution doesn't cover the whole range of the target distribution, but may be good enough.

```
M = 4;

p𝒟 = MixtureDistribution[{2, 1},
   {NormalDistribution[], NormalDistribution[2, 1 / 4]}];

q𝒟2 = UniformDistribution[{-3, 3}];
gq𝒟 = Plot[{M * PDF[q𝒟2 , x]}, {x, -3, 4}, Filling → Axis, PlotRange → {-5, 6}];

Clear[u, x];
u := RandomVariate[UniformDistribution[]];
x := Module[{n}, n = 1;
   While[Abs[xr = RandomVariate[q𝒟2]] > 3, xr;
    n++]; Return[xr]];


samples = Table[If[(u < PDF[p𝒟, x0 = x] / (M * PDF[q𝒟2, x0])), x0, Null], {10 000}];
```

```
gH = Histogram[samples, 48, "PDF", PlotRange → {{-3, 4}, {0, .7}}]
```



```
gp𝒟 = Plot[{PDF[p𝒟, x]}, {x, -5, 6}, Filling → Axis, PlotRange → {-3, 4}];
Show[{gH, gp𝒟, gq𝒟}]
```



References

Andrieu, C., De Freitas, N., Doucet, A., & Jordan, M. I. (2003). An introduction to MCMC for machine learning. Machine Learning, 50(1), 5–43.

Battaglia, P. W., Kersten, D., & Schrater, P. R. (2011). How haptic size sensations improve distance perception. PLoS Computational Biology, 7(6), e1002080. doi:10.1371/journal.pcbi.1002080

Geman, S., & Geman, D. (1984). Stochastic relaxation, Gibbs distributions, and the Bayesian restoration of images. Pattern Analysis and Machine Intelligence, IEEE Transactions on, (6), 721–741.

Gershman, S. J., Vul, E., & Tenenbaum, J. B. (2012). Multistability and perceptual inference. Neural Computation, 24(1), 1–24.

Kersten. (1991). Transparency and the cooperative computation of scene attributes. In M. S. Landy

(Ed.), Computational models of visual processing (pp. 209–228). The MIT Press.

Lake, B. M., Salakhutdinov, R., & Tenenbaum, J. B. (2015). Human-level concept learning through probabilistic program induction. Science, 350(6266), 1332–1338. http://doi.org/10.1126/-science.aab3050

Shi, L., Griffiths, T. L., Feldman, N. H., & Sanborn, A. N. (2010). Exemplar models as a mechanism for performing Bayesian inference. Psychonomic Bulletin & Review, 17(4), 443–464. http://-doi.org/10.3758/PBR.17.4.443

Sundareswara, R., & Schrater, P. R. (2008). Perceptual multistability predicted by search model for Bayesian decisions. Journal of Vision, 8(5), 12–12. doi:10.1167/8.5.12

Vul, E., Goodman, N. D., Griffiths, T. L., & Tenenbaum, J. B. (2012). One and done? Optimal decisions from very few samples. Proceedings of the 31st Annual Meeting of the Cognitive Science Society, 148–153.