GMDL, HW #1

Oren Freifeld and Ron Shapira Weber
The Department of Computer Science, Ben-Gurion University

Release Date: 24/3/2023
Submission Deadline: 1/6/2023, 23:59

Contents

Version Log

- 1.02, 19/4/2023. Fixed some typos.
- 1.01, 18/4/2023. Updated the submission deadline.
- 1.00, 24/3/2023. Initial release.

Contents

1 Exact sampling in the Ising model via Dynamic Programming

This section is dedicated to building an exact sampler for the Ising model, on an 8×8 2D regular lattice, using dynamic programming. At each of three temperatures, ten random samples are displayed and two empirical expectations are computed. The main computer exercise is Computer Exercise 7. Those before it are shorter, easier, and are used as building blocks or debugging tools for that computer exercise. Computer Exercise 8 boils down to using the implementation from Computer Exercise 7. Finally, in Problem 2 you are asked to analyze the results from Computer Exercise 8.

1.1 The Clique Functions in the Ising Model

Computer Exercise 1 *Implement a function, G, whose input is*

1. *a 1D numpy array, denoted here by r and in your code by* `row_s`*;*
2. *a scalar* Temp*,*

and whose output is the scalar

$$\exp\left(\frac{1}{\text{Temp}}\sum_{i=1}^{\text{length}(r)-1} r_i r_{i+1}\right). \tag{1}$$

It should be a single line of code, and you must not use a loop (nor a list/dict comprehension). ♢

Computer Exercise 2 *Implement a function, F, whose input is*

1. *two 1D numpy arrays of the same length, denoted here by r and $\tilde{r}$ and in your code by* `row_s` *and* `row_t`*;*
2. *a scalar* Temp*,*

and whose output is the scalar

$$\exp\left(\frac{1}{\text{Temp}}\sum_{i=1}^{\text{length}(r)} r_i \tilde{r}_i\right). \tag{2}$$

Again, it should be a single line of code, you must not use a loop (nor a list/dict comprehension). ♢

1.2 Brute Force on Small Lattices

Some of the computer exercises in this section require you to use an absurd number (from a programmer's perspective) of nested loops. Thus, you can (but do not have to) exploit Python's itertools module to make this a bit more elegant.

Computer Exercise 3 *In the case of a 2×2 lattice,* i.e.

$$S = \{(i,j)\}_{1\le i\le 2, 1\le j\le 2}, \tag{3}$$

compute Z_{Temp} (for three different values of Temp *where* Temp $\in \{1, 1.5, 2\}$*) using brute force (use 4 nested For loops, one for each of the x_s's, the looping is done over the values that x_s can take: $\{-1, 1\}$). To help you debug: For* Temp $= 1$*, your result should be $Z_{\text{Temp}} = 121.23\ldots$.* ♢

Computer Exercise 4 *In the case of a 3×3 lattice,* i.e.

$$S = \{(i,j)\}_{1\le i\le 3, 1\le j\le 3}, \tag{4}$$

compute Z_{Temp} (for three different values of Temp *where* Temp $\in \{1, 1.5, 2\}$*) using brute force (use 9 nested For loops, one for each of the x_s's, the looping is done over the values that x_s can take: $\{-1, 1\}$). To help you debug: For* Temp $= 1$*, your result should be $Z_{\text{Temp}} = 10^5 \cdot 3.65\ldots$.* ♢

Obviously, 9 For loops is pretty ugly. We will see below we can use fewer loops, even if we remain in a brute-force approach.

The most computationally-efficient approach, to achieve the main goal of this section (exact sampling from the Ising model), is to use a raster or boustrophedonic[1] site-visitation schedule and then compute conditional probabilities, site-by-site, in a backward ordering. The number of computations is $O(64 \cdot 2^9) = O(2^{15})$. **However, this is NOT what you are asked to do. Instead, you are requested to do something much simpler (and suboptimal):** the programming burden becomes substantially easier if the 8×8 lattice is represented as a Markov chain with 8 sites (where each of them encodes an entire row from the original 8×8 lattice) and 8 corresponding variables, $y_1, \ldots, y_8$. Although the computational complexity becomes $O(8 \cdot 2^{16}) = O(2^{19})$, the programming complexity is vastly reduced – so we will go with that. Each site variable, y_s (where $s = 1, \ldots, 8$) has 2^8 possible states, $y_s \in \{0, 1, \ldots, 255\}$, corresponding to the 2^8 possible configurations of row s in the Ising lattice. The correspondence is thus $y_s \leftrightarrow (x_{(s,1)}, \ldots, x_{(s,8)})$. A convenient mapping between y_s and $(x_{(s,i)})_{i=1}^8$ is to use the 8-bit binary representation of y_s, together with the conversion of 0's to -1's.

Example 1 *Suppose $y_s = 136$. The 8-bit binary representation of 136 is '10001000'. Therefore, the corresponding row (*i.e.*, $(x_{(s,i)})_{i=1}^8$) is [1 -1 -1 -1 1 -1 -1 -1].* ◊

The following helper function computes the $y_s \mapsto (x_{(s,i)})_{i=1}^8$ mapping.

```
import numpy as np
def y2row(y,width=8):
    """
    y: an integer in (0,...,(2**width)-1)
    """
    if not 0<=y<=(2**width)-1:
        raise ValueError(y)
    my_str=np.binary_repr(y,width=width)
    # my_list = map(int,my_str) # Python 2
    my_list = list(map(int,my_str)) # Python 3
    my_array = np.asarray(my_list)
    my_array[my_array==0]=-1
    row=my_array
    return row
```

Remark 1 *Note that in this section (§ 1), the y's merely stand for groups of x's, used as a mechanism for simplifying the programming. These y's should not be confused with "observations". In this section we use only the prior, $p(x)$, and there are no noisy observations;* i.e.*, this section does not involve a posterior distribution. However, if we let z denote hypothetical observations, the sampling mechanism from this section can be easily adjusted to sampling from the posterior, $p(x|z)$, in the case that each data point is connected in the graph to a single site in the original graph of $p(x)$;* i.e.*, if the likelihood, $p(z|x)$, factorizes as $p(z|x) = \prod_{s \in S} p(z_s|x_s)$. As we saw in class, the only thing that will have*

[1] Namely, from right to left and from left to right in alternate lines. The word came from Greek and refers to how an ox turns when plowing a field.

to change is the functional form of each clique function, without changing the graph/cliques. To clarify, we reiterate that this is not something you are asked to do.

Computer Exercise 5 *In the case of a* 2×2 *lattice, converted to a chain of length 2, compute* Z_{Temp} *(for three different values of* Temp *where* $\text{Temp} \in \{1, 1.5, 2\}$*) using brute force (use 2 nested For loops, one for each of the* y_s*'s, the looping is done over the values that* y_s *can take:* $\{0, 1, 2, 3\}$*). In effect,*

$$p(y) = p(y_1, y_2) = \frac{1}{Z_{\text{Temp}}} G(y_1)G(y_2)F(y_1, y_2) \tag{5}$$

and

$$Z_{\text{Temp}} = \sum_{y} G(y_1)G(y_2)F(y_1, y_2) = \sum_{y_1}\sum_{y_2} G(y_1)G(y_2)F(y_1, y_2) \tag{6}$$

where the singleton function

$$G(y_s) = \exp\left(\frac{1}{\text{Temp}} x_{(s,1)}x_{(s,2)}\right), \qquad s = 1, 2 \tag{7}$$

representing, for each of the two rows, s*, the intra-row pair clique from the original graph, and a pair clique function*

$$F(y_1, y_2) = \exp\left(\frac{1}{\text{Temp}} \sum_{i=1}^{2} x_{(1,i)}x_{(2,i)}\right) \tag{8}$$

representing the product of the two inter-row pair cliques from the original graph.

In your implementation, use the provided helper function ***y2row*** *(in the second argument, pass width=2 since here there are only two* x_s*'s in each row) and your implemented* G *and* F *from the previous exercises.*

To help you debug: For $\text{Temp} = 1$*, your result should be* $Z_{\text{Temp}} = 121.23\ldots$ *(which is of course the same result from before).* ♢

Computer Exercise 6 *In the case of a* 3×3 *lattice, converted to a chain of length 3, compute* Z_{Temp} *(for three different values of* Temp *where* $\text{Temp} \in \{1, 1.5, 2\}$*) using brute force (use 3 nested For loops, one for each of the* y_s*'s, the looping is done over the values that* y_s *can take:* $\{0, 1, \ldots, 7\}$*). In effect,*

$$p(y) = p(y_1, y_2, y_3) = \frac{1}{Z_{\text{Temp}}} G(y_1)G(y_2)G(y_3)F(y_1, y_2)F(y_2, y_3) \tag{9}$$

and

$$\begin{aligned} Z_{\text{Temp}} &= \sum_{y} G(y_1)G(y_2)G(y_3)F(y_1, y_2)F(y_2, y_3) \\ &= \sum_{y_1}\sum_{y_2}\sum_{y_3} G(y_1)G(y_2)G(y_3)F(y_1, y_2)F(y_2, y_3) \end{aligned} \tag{10}$$

where the singleton function

$$G(y_s) = \exp\left(\frac{1}{\text{Temp}} \sum_{i=1}^{2} x_{(s,i)}x_{(s,i+1)}\right), \qquad s = 1, 2, 3 \tag{11}$$

representing, for each of the three rows, s, the product of the two intra-row pair cliques from the original graph, and pair clique functions

$$F(y_s, y_{s+1}) = \exp\left(\frac{1}{\text{Temp}} \sum_{i=1}^{3} x_{(s,i)} x_{(s+1,i)}\right) \qquad s = 1, 2 \tag{12}$$

representing, for each of the two pairs of rows, $(s, s+1)$, the product of the three inter-row pair cliques from the original graph.

In your implementation, use the provided helper function ***y2row*** *(in the second argument, pass width=3 since here there are three x_s's in each row) and your implemented G and F from the previous exercises.*

To help you debug: For $\text{Temp} = 1$*, your result should be* $Z_{\text{Temp}} = 10^5 \cdot 3.65\ldots$ *(which is of course the same result from before).* ♢

1.3 Dynamic Programming on an 8×8 Lattice

The following Computer Exercise is the main task in § 1.

Computer Exercise 7 *Using dynamic programming and an 8×8 lattice, build an exact sampler for each of the three temperatures,* $\text{Temp} \in \{1, 1.5, 2\}$*.* ***The details for how to do it appear at § 1.4.*** *Use the samplers to generate ten independent samples (to clarify, each such sample is an entire 8×8 image) at each of the three temperatures, and display these as thirty images all in a single figure (three rows, one for each temperature, and ten columns of 8×8 black-and-white images; the plt.subplot command should be useful here). When displaying your images using plt.imshow (in Python), you should use the Interpolation="None" option (*i.e.*, passing the string "None", not to be confused with Interpolation=None, which passes the Python built-in None).* ♢

Remark 2 *In the problem above, when* $\text{Temp} = 1$*, some of you may be puzzled to see that most of the samples (where each sample is an entire 8-by-8 image) are "all blue" (or "all black", depending on the colormap you are using) and you will be wondering how come this is not symmetric. In effect, you will feel that it makes sense that most of the images are unicolor, but you would have expected that about half of them will be "all blue" ("-1") and about half of them will be "all red" ("+1"). Well, this is not how imshow works by default. Here is what is going on: if you hover with the mouse over the pixels in these unicolor images, you will see (at the bottom of the figure) that, in fact, some of them are all "-1" and some of them are all "+1", even though both cases are "all blue". This is because by default, imshow(img) scales the display such that the blue stands for img.min(), and red stands for img.max(). If var(img)=0 (*i.e.*, all pixels share the same value), then img.min()=img.max() so only a single color (blue) is used. This is regardless whether all the pixels are -1 or all the pixels are +1. One way to overcome this visualization issue is using the vmin and vmax optional parameters: imshow(...,vmin=-1, vmax=+1). Read imshow's documentation about it. That said, later, when you show images corrupted by real-valued noise, you should avoid this [-1,1] range because it will trim the values which may be outside that range.* ♢

Problem 1 *A student, who used Python 2 (as opposed to Python 3), encountered a weird problem when working on Computer Exercise 7. While the resulting*

samples for Temp $= 1$ *and* Temp $= 1.5$ *look as one might expect, for* Temp $= 2$ *the samples that the student obtained looked completely random with no structure whatsoever. Explain what programming bug the student probably had, and from which distribution (instead of the Ising model) over binary images that student really sampled from. Hint: compare the outputs of the following commands (the syntax below assumes a Linux terminal):*

```
python2 -c "print [1/Temp for Temp in [1,1.5,2]]"
python3 -c "print ([1/Temp for Temp in [1,1.5,2]])"
```

$\Diamond$

Computer Exercise 8 *Using the three samplers you implemented above, at each of the three temperatures, draw 10,000 samples,* $x(1), \ldots, x(10000)$ *(each such sample is an* 8×8 *binary image) and compute two empirical expectations:*

$$\widehat{E}_{\text{Temp}}(X_{(1,1)}X_{(2,2)}) \triangleq \frac{1}{10000}\sum_{n=1}^{10000} x_{(1,1)}(n)x_{(2,2)}(n) \tag{13}$$

$$\widehat{E}_{\text{Temp}}(X_{(1,1)}X_{(8,8)}) \triangleq \frac{1}{10000}\sum_{n=1}^{10000} x_{(1,1)}(n)x_{(8,8)}(n) \tag{14}$$

$\Diamond$

where Temp $= 1, 1.5,$ *and* 2 *and where* $x_{(i,j)}(n)$ *is the value at the* (i,j)*-th lattice site of sample* n*. To help you debug here are the values you should get for* Temp $= 1$*:* $\widehat{E}_{\text{Temp}}(X_{(1,1)}X_{(2,2)}) \approx 0.95$*;* $\widehat{E}_{\text{Temp}}(X_{(1,1)}X_{(8,8)}) \approx 0.9$*.*

Problem 2 *Using the results of the Computer Exercise above, explain the relative values of* $\widehat{E}$*, in terms of the spatial distance of the lattice sites and in terms of the temperature.* $\Diamond$

1.4 Detailed Notes and How to Debug

1. You should do the sampling in terms of the y_s's, not the x_s's, since it is considerably simpler.
2. In this representation, where we use an 8-length Markov chain (with each of its variables taking values in $\{0, 1, \ldots, 255\}$) to represent a binary MRF defined over an 8×8 lattice, there are two kinds of clique functions:
   (a) Singletons of the form

   $$G(y_s) = \exp\left(\frac{1}{\text{Temp}}\sum_{i=1}^{7} x_{(s,i)}x_{(s,i+1)}\right), \qquad s = 1, 2, \ldots, 8 \tag{15}$$

   representing, for each of the eight rows, s, the product of the seven intra-row pair cliques from the original graph;
   (b) pair cliques of the form

   $$F(y_s, y_{s+1}) = \exp\left(\frac{1}{\text{Temp}}\sum_{i=1}^{8} x_{(s,i)}x_{(s+1,i)}\right) \qquad s = 1, 2, \ldots, 7 \tag{16}$$

   representing, for each of the seven pairs of rows, $(s, s+1)$, the product of the eight inter-row pair cliques from the original graph.

Thus,

$$p(y_1, \ldots, y_8) \propto \left(\prod_{s=1}^{8} G(y_s) \right) \left(\prod_{s=1}^{7} F(y_s, y_{s+1}) \right) . \tag{17}$$

By now, you should already have these two types of functions implemented.

The dynamic programming iterations, adopting the site-visitation schedule $1, 2, \ldots, 8$, are then

$$T_1(y_2) = \sum_{y_1=0}^{255} G(y_1) F(y_1, y_2) \qquad y_2 \in \{0, 1, \ldots, 255\} ; \tag{18}$$

$$T_k(y_{k+1}) = \sum_{y_k=0}^{255} T_{k-1}(y_k) G(y_k) F(y_k, y_{k+1}) \qquad 2 \le k \le 7 \quad y_{k+1} \in \{0, 1, \ldots, 255\} ; \tag{19}$$

$$\mathbb{R} \ni T_8 = \sum_{y_8=0}^{255} T_7(y_8) G(y_8) \quad (= Z_{\text{Temp}}) . \tag{20}$$

Working backwards:

$$p_8(y_8) = \frac{T_7(y_8) G(y_8)}{Z_{\text{Temp}}} \qquad y_8 \in \{0, 1, \ldots, 255\} ; \tag{21}$$

$$p_{k|k+1}(y_k | y_{k+1}) = \frac{T_{k-1}(y_k) G(y_k) F(y_k, y_{k+1})}{T_k(y_{k+1})} \qquad k = 7, 6, \ldots, 2 \quad y_k, y_{k+1} \in \{0, 1, \ldots, 255\} ; \tag{22}$$

$$p_{1|2}(y_1 | y_2) = \frac{G(y_1) F(y_1, y_2)}{T_1(y_2)} . \tag{23}$$

3. On a computer:
   (a) p_8 is a 1D array of length $2^8 = 256$;
   (b) for each $k \in \{1 \ldots, 7\}$, $p_{k|k+1}$ is a $2^8 \times 2^8 = 256 \times 256$ array (so you should have 7 such 2D arrays: $p_{7|8}$; $p_{6|7}$; $p_{5|6}$; $p_{4|5}$; $p_{3|4}$; $p_{2|3}$; $p_{1|2}$).

4. Sampling is fast; computing the conditional probabilities is slow. So you may want to save the conditional probabilities (each of which is a 256×256 array as mentioned above) after they are computed (or, at least save the "T functions").

5. Debug your dynamic-programming implementation on 2×2 and 3×3 lattices by computing Z_{Temp}, the normalizing constant, and the associated T functions. Here are some results for small lattices (with Temp = 1):

```
2x2 lattice results:
T1: [ 21.18917525   8.20463255   8.20463255  21.18917525]
T2 = Z: 121.232931344

3x3 lattice results:
T1: [ 155.37102759   46.44297052   31.70116107   46.44297052   46.44297052
   31.70116107   46.44297052  155.37102759]
T2: [ 23416.16435187   4634.76802124   3916.10003703   4634.76802124
   4634.76802124   3916.10003703   4634.76802124  23416.16435187]
T3 = Z: 365645.749136
```

In other words, before you start the sampling (which is done in the "backward pass", in effect, in ordering 8,7,6,...,1), you should debug on a small lattice (2×2 or 3×3), making sure you get the correct results for the "forward pass". In effect, computing the T's, the last of which is Z_{Temp}.

To be more concrete:

- For a 2×2 lattice, each y_s (for $s \in \{1, 2\}$), *i.e.*, a row, is a 1D array of length $2^2 = 4$, whose entries take integral values between 0 and $3 = 2^2 - 1$, inclusive. T_1 is a function of y_2. Since y_2 has 4 different values, it follows that T_1 should be represented as a 1D array of length 4. Particularly, since $T_1(y_2) = \sum_{y_1=0}^{3} G(y_1)F(y_1, y_2)$, it follows that the 4 entries of the T_1 array are:

$$T_1(y_2 = 0) = \sum_{y_1=0}^{3} G(y_1)F(y_1, 0)\,; \tag{24}$$

$$T_1(y_2 = 1) = \sum_{y_1=0}^{3} G(y_1)F(y_1, 1)\,; \tag{25}$$

$$T_1(y_2 = 2) = \sum_{y_1=0}^{3} G(y_1)F(y_1, 2)\,; \tag{26}$$

$$T_1(y_2 = 3) = \sum_{y_1=0}^{3} G(y_1)F(y_1, 3)\,. \tag{27}$$

  T_2 is a scalar because it is the last T (since we have only two rows). In fact, $T_2 = Z_{\text{Temp}}$, the normalizing constant:

$$T_2 = Z_{\text{Temp}} = \sum_{y_2=0}^{3} T_1(y_2)G(y_2)\,. \tag{28}$$

  Now you can compare it with the brute-force result.

- For a 3×3 lattice, each y_s (for $s \in \{1, 2, 3\}$), *i.e.*, a row, is a 1D array of length $2^3 = 8$, whose entries take integral values between 0 and $7 = 2^3 - 1$, inclusive. T_1 is a function of y_2. Since y_2 has $2^3 = 8$ different values, it follows that T_1 should be represented as a 1D array of length 8. Particularly, since $T_1(y_2) = \sum_{y_1=0}^{7} G(y_1)F(y_1, y_2)$, it follows that the

8 entries of the T_1 array are:

$$T_1(y_2 = 0) = \sum_{y_1=0}^{7} G(y_1)F(y_1, 0)\,; \tag{29}$$

$$T_1(y_2 = 1) = \sum_{y_1=0}^{7} G(y_1)F(y_1, 1)\,; \tag{30}$$

$$\vdots \tag{31}$$

$$T_1(y_2 = 7) = \sum_{y_1=0}^{7} G(y_1)F(y_1, 7)\,. \tag{32}$$

Now, T_2 is a function of y_3. Since y_3 has $2^3 = 8$ different values, it follows that T_2 should be represented as a 1D array of length 8. Particularly, since $T_2(y_3) = \sum_{y_2=0}^{7} T_1(y_2)G(y_2)F(y_2, y_3)$, it follows that the 8 entries of the T_2 array are:

$$T_2(y_3 = 0) = \sum_{y_2=0}^{7} T_1(y_2)G(y_2)F(y_2, 0)\,; \tag{33}$$

$$T_2(y_3 = 1) = \sum_{y_2=0}^{7} T_1(y_2)G(y_2)F(y_2, 1)\,; \tag{34}$$

$$\vdots \tag{35}$$

$$T_2(y_3 = 7) = \sum_{y_2=0}^{7} T_1(y_2)G(y_2)F(y_2, 7)\,. \tag{36}$$

T_3, since it is the last T (since we have only 3 rows) is a scalar. In fact, $T_3 = Z_{\text{Temp}}$, the normalizing constant.

$$T_3 = Z_{\text{Temp}} = \sum_{y_3=0}^{7} T_2(y_3)G(y_3)\,. \tag{37}$$

Now you can compare it with the brute-force result.

- For an 8×8 lattice, each y_s (for $s \in \{1, 2, 3, 4, 5, 6, 7, 8\}$), *i.e.*, a row, is a 1D array of length 8, whose entries take integral values between 0 and $255 = 2^8 - 1$, inclusive. So $T_1, T_2, ..., T_7$ are arrays of length $2^8 = 256$, while $T_8 = Z_{\text{Temp}}$ since it is the last one.

  Once you are done with this forward pass, use these T's to compute the p's.

  p_8 is a function of y_8 (in fact it is the pmf over the 256 states y_8 can take). So p_8 should be represented as a 1D array of length $2^8 = 256$.

  $p_{7|8}(y_7, y_8)$ is the pmf of y_7 given y_8. So for every value of y_8 – and we have 256 of these – we should have 256 values to cover the probability of all possible 256 states y_7 may assume. So we represent $p_{7|8}$ using a 256 by 256 array.

Similarly for $p_{6|7}(y_6, y_7)$, $p_{5|6}(y_5, y_6)$, $\ldots$, $p_{1|2}(y_1, y_2)$.
Finally, to produce the entire sample, we first sample y_8 from p_8 (one way to sample from a distribution of a discrete RV is using np.random.choice – do not forget to pass the distribution as an input argument to that function). Then, we pick the row (or column, depends how you built your $p_{7|8}$) that corresponds to that y_8 we drew, and use it to sample $y_{7|8}$. And so on till we end with sampling $y_{1|2}$.

Remark 3 *As usual, when writing p, we could have also absorbed singletons into the pairwise functions,* e.g.*:*

$$\begin{aligned} p(y_1,\ldots,y_8) \propto & \underbrace{G(y_1)F(y_1,y_2)}_{H_{1,2}(y_1,y_2)}\underbrace{G(y_2)F(y_2,y_3)}_{H_{2,3}(y_2,y_3)}\underbrace{G(y_3)F(y_3,y_4)}_{H_{3,4}(y_3,y_4)} \\ & \times \underbrace{G(y_4)F(y_4,y_5)}_{H_{4,5}(y_4,y_5)}\underbrace{G(y_5)F(y_5,y_6)}_{H_{5,6}(y_5,y_6)}\underbrace{G(y_6)F(y_6,y_7)}_{H_{6,7}(y_6,y_7)} \\ & \times \underbrace{G(y_7)G(y_8)F(y_7,y_8)}_{H_{7,8}(y_7,y_8)} = \prod_{s=1}^{7} H_{s,s+1}(y_s,y_{s+1})\,. \end{aligned} \tag{38}$$

♢

2 Approximated Sampling in the Ising model via MCMC

We will test the Gibbs sampler by computing expectations and comparing with the exact results (which we obtained using dynamic programming). We will also pretend that the Ising model is a good model for images, and study image restoration from Ising-model samples corrupted by noise.

Remark 4 ***Before you start coding*** *a single line in this section, do yourself a favor and* ***read all of*** § *2* ***first.*** ♢

Reminder ($p(x_s|_s x)$ in Ising's model) *The Ising-model prior,*

$$p(x) \propto \exp\left(\frac{1}{\text{Temp}}\sum_{s\sim t} x_s x_t\right) \qquad \text{Temp} > 0\,, \tag{39}$$

implies that

$$p(x_s|_s x) \propto \begin{cases} \exp\left(\frac{1}{\text{Temp}}\sum_{t:t\in\eta_s} x_t\right) & \text{if } x_s = 1 \\ \exp\left(-\frac{1}{\text{Temp}}\sum_{t:t\in\eta_s} x_t\right) & \text{if } x_s = -1 \end{cases}. \tag{40}$$

So sampling from this conditional distribution, $p(x_s|_s x)$, is just flipping a biased coin, where the bias is affected by the states of the neighbors of x_s. ♢

Reminder ($p(x_s|_s x, y)$ in Ising's model, assuming Gaussian *i.i.d.* additive noise) *Under this assumption, letting x denote the sample from the Ising-model prior and y denote the noisy observations (where σ^2 is the variance of the Gaussian* i.i.d. *zero-mean noise), we have:*

$$p(x|y) \propto \exp\left(\frac{1}{\text{Temp}}\left(\sum\nolimits_{s\sim t} x_s x_t\right) - \frac{1}{2\sigma^2}\sum\nolimits_s (y_s - x_s)^2\right) \tag{41}$$

and

$$p(x_s|_s x, y) \propto \begin{cases} \exp\left(\frac{1}{\text{Temp}}\left(\sum_{t:t\in\eta_s} x_t\right) - \frac{1}{2\sigma^2}(y_s-1)^2\right) & \text{if } x_s = 1 \\ \exp\left(\frac{1}{\text{Temp}}\left(-\sum_{t:t\in\eta_s} x_t\right) - \frac{1}{2\sigma^2}(y_s+1)^2\right) & \text{if } x_s = -1 \end{cases} . \tag{42}$$

Again, the sampling is nothing more than flipping a biased coin, where here the bias is affected by both (i) the states of the neighbors of x_s and (ii) y_s, the measurement associated with x_s. ♢

2.1 Estimation: Comparisons with the Nearly-exact Values

In § 1 we used exact sampling to estimate

$$E_{\text{Temp}}(X_{(1,1)}X_{(2,2)}) \text{ and } E_{\text{Temp}}(X_{(1,1)}X_{(8,8)})$$

through their empirical expectations,

$$\widehat{E}_{\text{Temp}}(X_{(1,1)}X_{(2,2)}) \triangleq \frac{1}{10000}\sum_{n=1}^{10000} x_{(1,1)}(n)x_{(2,2)}(n) \tag{43}$$

$$\widehat{E}_{\text{Temp}}(X_{(1,1)}X_{(8,8)}) \triangleq \frac{1}{10000}\sum_{n=1}^{10000} x_{(1,1)}(n)x_{(8,8)}(n) \tag{44}$$

where $x(1), x(2), \ldots, x(10000)$ were 10,000 exact samples from the Ising model. We refer to these expectations as "nearly-exact" for the following reason: the term "exact" stems from the fact the samples were obtained using exact sampling, while the "nearly" qualifier is added since an empirical expectation, *i.e.*, the average over the samples (*i.e.*, AKA the sample mean) is a random quantity which, by the Law of Large Numbers, converges to the true expectation as the number of samples tends to infinity.

Now, in this section, instead of using exact sampling, we will produce the samples using MCMC, particularly the Gibbs-sampling method.

Computer Exercise 9 *Build a Gibbs sampler for the 8×8 Ising model at temperatures* $\text{Temp} \in \{1, 1.5, 2\}$. *Compute the empirical expectations of*

$$E_{\text{Temp}}(X_{(1,1)}X_{(2,2)}) \text{ and } E_{\text{Temp}}(X_{(1,1)}X_{(8,8)}) \tag{45}$$

using two different methods; see below. Compare the estimates to the nearly-exact values computed in § 1; see Table 1.
Programming Note: *A convenient way to handle boundaries is to embed the $n \times n$ lattice in the interior of an $(n+2) \times (n+2)$ lattice whose boundary values are set to zero, and then visit only the $n \times n$ interior sites of the larger lattice.*♢

Method 1. **Independent Samples.** At each temperature, draw 10,000 (approximated) samples from the Ising-model prior,

$$p(x) \propto \exp\left(\frac{1}{\text{Temp}}\sum_{s\sim t} x_s x_t\right) , \tag{46}$$

using the Gibbs sampler. For each sample, initiate the Gibbs sampler at a random configuration; *i.e.*, sample all the pixels *i.i.d.* using a fair coin, and assign the values $\{-1, 1\}$ accordingly. One way to do it in Python is using
`np.random.randint(low=0,high=2,size=(8,8))*2-1)`.
Next, update sites in a deterministic, raster-scan, order (a hfixed site-visitation schedule). The sample is the obtained configuration after 25 passes through the entire graph (*i.e.*, after 25 "sweeps" – where each sweep involves $8 \times 8 = 64$ single-site updates). Putting it differently, you are asked to run 10,000 such Markov Chains, where each chain has 25×64 iterations. To be clear, the initialization should always be done using *i.i.d.* coin flips as mentioned above, but please use a different realization of that random initialization for each Markov chain. Let the obtained 10,000 (approximated) samples be denoted by $x(1), \ldots, x(10000)$. Your estimates are then

$$\frac{1}{10000} \sum_{n=1}^{10000} x_{(1,1)}(n) x_{(2,2)}(n) \text{ and } \frac{1}{10000} \sum_{n=1}^{10000} x_{(1,1)}(n) x_{(8,8)}(n) \,. \tag{47}$$

Method 2. **Ergodicity.** Beginning with a random configuration (*i.e.*, sampling all the pixels *i.i.d.* using a fair coin, as before), run the Gibbs sampler for 25,000 sweeps of the lattice (*i.e.*, a single Markov Chain of 25,000 $\times$ 64 iterations). Use the empirical averages of $x_{(1,1)}x_{(2,2)}$ and $x_{(1,1)}x_{(8,8)}$ over all but the first 100 sweeps (*i.e.*, following a so-called "burn-in period" of 100 sweeps, use the 24,900 configurations obtained at the end of each of the remaining sweeps to compute the empirical averages).

Remark 5 *Note that for computing the average of, say, such 24,900 quantities, you do not need to store 24,900 values in memory. In effect, suppose you have a sequence of N values, denoted by $z_1, z_2, \ldots, z_N$ and you want to compute $\frac{1}{N}\sum_{n=1}^{N} z_i$. Then you can discard each z_i once you iteratively update the current estimate of the mean:*

$$\mu^{[1]} = z_1 \tag{48}$$

$$\mu^{[n]} = \frac{1}{n} \sum_{m=1}^{n} z_m = \frac{1}{n} \left(\left(\sum_{m=1}^{n-1} z_m \right) + z_n \right) \tag{49}$$

$$= \frac{1}{n} \left(\frac{n-1}{n-1} \left(\sum_{m=1}^{n-1} z_m \right) + z_n \right) = \frac{(n-1)\mu^{[n-1]} + z_n}{n} \qquad n \in 2, 3, \ldots, N \,. \tag{50}$$

$\Diamond$

Problem 3 *Compare the results obtained from the two methods, and also compare these results with the values from exact sampling (obtained using dynamic programming). Speculate about the reasons for any substantial discrepancies.* $\Diamond$

2.2 Binary Image Restoration

These experiments are to be performed on a 100×100 lattice.

temperature	$\hat{E}_{\text{Temp}}(X_{(1,1)}X_{(2,2)})$	$\hat{E}_{\text{Temp}}(X_{(1,1)}X_{(8,8)})$
1	0.95	0.9
1.5	your result from § 1	your result from § 1
2	your result from § 1	your result from § 1

Table 1: Expectations estimated from exact samples using dynamic programming

Computer Exercise 10 *At each of the three temperatures* $\text{Temp} \in \{1, 1.5, 2\}$*:*

1. *Using Gibbs sampling, generate a single* 100×100 *sample,* x*, from the* 100×100 *Ising-model prior using 50 raster-scan sweeps.*

2. *Add to the sample an array of* 100×100 *Gaussian noise values, sampled* i.i.d. *from* $\mathcal{N}(0, 2^2 = 4)$*. We denote this noise array by* η*. Here is one way to achieve this in Python:*

```
eta = 2*np.random.standard_normal(size=(100,100))
y = x + eta
```

3. *Using Gibbs sampling, generate a sample from the posterior distribution of the uncorrupted sample given the corrupted sample,*

$$x \sim p(x|y)\,, \tag{51}$$

again using 50 sweeps. To remove any doubt, even though the y *values are continuous, the* x *values are still discrete. Thus, even when you are using the Gibbs sampler for sampling from* $p(x|y)$*, you should still initialize your* x *values using random* i.i.d. *flips of a fair coin, to produce values in* $\{-1, 1\}$*. Particularly, do* ***not*** *initialize* x *by setting it to* y*.*

4. *For comparison, compute the "ICM" (Iterated Conditional Modes) restoration: upon a random initialization, visit sites, in a raster order, and replace at each site* s *the current value of* x_s *by its most-likely value (of the two possibilities,* ± 1*), under the posterior distribution and conditioned on the four neighbors:*

$$x_s^{new} = \underset{x_s \in \{-1,1\}}{\arg\max}\; p(x_s|_s x, y) \tag{52}$$

where

$$p(x_s|_s x, y) = p(x_s|x_{\eta_s}, y) = p(x_s|x_{\eta_s}, y_s)\,. \tag{53}$$

This greedy algorithm will converge within a few raster cycles.

5. *For yet another comparison, display the maximum-likelihood estimate,* $\arg\max_x p(y|x)$*. Recall that each pixel in* x *takes values in* $\{-1, 1\}$*;* i.e.*, the maximum-likelihood estimate should also take only these discrete values. Particularly, it is* ***not*** *simply (the continuous)* y*.*

6. *Display all five images (from the previous 5 steps) on a single plot. (see the subplot and imshow commands). When showing x or its estimates, set the imshow optional parameters vmin and vmax to -1 and $+1$, respectively. When showing y, however, do not do that since it may contain values outside the range $[-1, 1]$.* ◊

Remark 6 *Note well: the ICM result is (usually)* ***not*** *the argmax of the posterior.* ***So usually ICM is not the MAP solution*** *(where MAP stands for the maximum-a-posteriori); rather, ICM converges to a usually-local maximum of the posterior. Also, just like in the case of Gibbs sampling, when one can apply it to distributions that are not Gibbs distributions (*i.e.*, not MRFs) but using Gibbs sampling to Gibbs distributions is especially convenient (due to the structure of the conditionals), we have a similar situation for ICM: one can also apply ICM to distributions that are not Gibbs distributions, but using it for Gibbs distributions will be especially convenient, and for the same reason.* ◊