# Data C102: Image Denoising with a Gibbs-Sampling Algorithm

Keon Etebari

March 11, 2022

## 1 Introduction

Derive a Gibbs sampling algorithm to restore a corrupted image. Our image can be represented by a 2-dimensional array X of shape n × m, where the intensity of the (i, j)-th pixel is Xij . In this problem, we are given an image X whose pixels have been corrupted by noise, and the goal is to recover the original image Z.

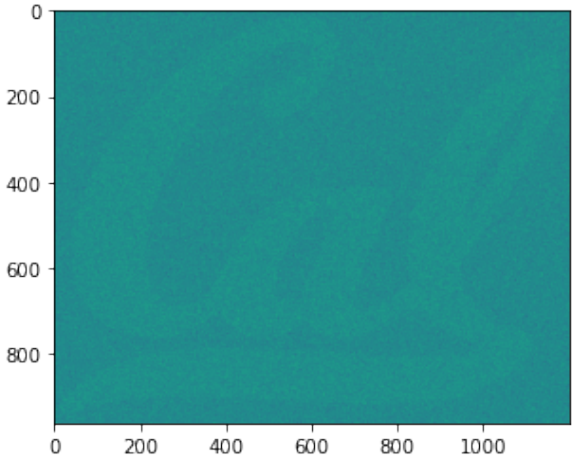


Figure 1: Figure 1.1: Corrupted Image

## 2 Theory

Load the image X.pkl as a numpy array X. Visualize the image.From plotting the image X, it is clear that it has been corrupted with noise. Let Z denote the original image, which we also represent as an n × m array. Let $I = \{(i, j) : 1 \leq i \leq n \text{ and } 1 \leq j \leq m\}$ denote the collection of all pixels in the image,represented by the corresponding index of the array. Given a pixel (i, j), define the set of neighboring pixels to be:

$$N_(i, j) = \{(i', j') \in \mathcal{I} : (i = i' \text{ and } |j - j'| = 1) \textbf{ or } (|i - i'| = 1 \text{ and } j = j'\} \tag{1}$$

To capture the fact that, in natural images, neighboring pixels are likely be similar, we consider the following prior over the original image:

$$p(Z) \propto \exp\left(-\frac{1}{2}\sum_{(i,j)\in\mathcal{I}}\left[aZ_{ij}^2 - b\sum_{(i'\,j')\in N_{(i,j)}} Z_{ij}Z_{i'j'}\right]\right) \tag{2}$$

Assuming the image has been corrupted with Gaussian noise $X_{(i,j)} \mid Z_{(i,j)} \sim \mathcal{N}\left(Z_{(i,j)}, \tau^{-1}\right)$

(independently across pixels $(i,j) \in \mathcal{I}$), the complete posterior can be written as

$$p(X \mid Z) \propto \exp\left(-\frac{1}{2}\sum_{(i,j)}\left[(a+\tau)\, Z_{ij}^2 - 2\tau Z_{ij}X_{ij} - b\sum_{(i'\,j')\in N_{(i,j)}} Z_{ij}Z_{i'j'}\right]\right) \tag{3}$$

Let $S_{ij} = \sum_{(i'\,j')\in N_{(i,j)}} Z_{i'j'}$ By completing the square in the posterior (3), we have:

$$Z_{ij} \mid (Z_{i'j'})_{(i'j')\neq(i,j)}, X \sim \mathcal{N}\left(\frac{\tau X_{ij} + bS_{ij}}{a+\tau}, \frac{1}{a+\tau}\right) \tag{4}$$

# 3 Naive Gibbs-Sampling Algorithm

- Initialize $Z^{(0)} = X$
- For $t = 1, ..., T$ :
  - Sample $Z_{(1,1)}^{(t)} \sim p\left(Z_{(1,1)} | Z_{(1,2)} = Z_{(1,2)}^{(t-1)}, Z_{(1,3)} = Z_{(1,3)}^{(t-1)}, ..., Z_{(n,m)} = Z_{(n,m)}^{(t-1)}, X\right)$
  - Sample $Z_{(1,2)}^{(t)} \sim p\left(Z_{(1,2)} | Z_{(1,1)} = Z_{(1,1)}^{(t)}, Z_{(1,3)} = Z_{(1,3)}^{(t-1)}, ..., Z_{(n,m)} = Z_{(n,m)}^{(t-1)}, X\right)$
  - ...
  - Sample $Z_{(n,m)}^{(t)} \sim p\left(Z_{(n,m)} | Z_{(1,1)} = Z_{(1,1)}^{(t)}, Z_{(1,2)} = Z_{(1,2)}^{(t)}, ..., Z_{(n,m-1)} = Z_{(n,m-1)}^{(t)}, X\right)$

Implement the Naive Gibbs sampler with a = 250, b = 62.5, and $\tau = 0.01$. Run our code for $T = 1$ iteration, i.e. update each coordinate exactly once. Visualize the resulting image $Z^{(1)}$. Time our code and estimate how long it would take to compute $Z^{(100)}$

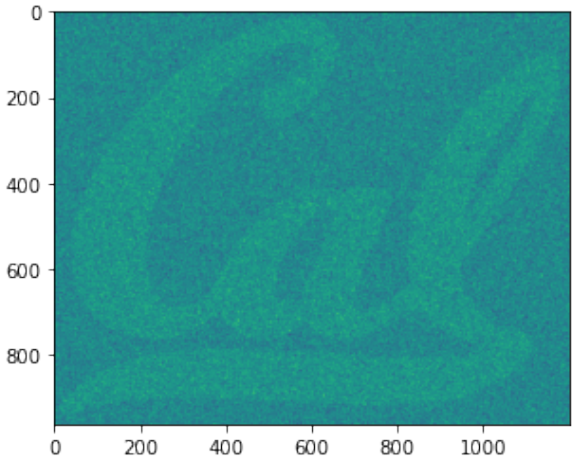


Figure 2: Figure 1.2: Naive Gibbs Sampler Image Output

  - Time to run one iteration of Gibbs Sampling: 10.713757514953613
  - Prediction for the time to calculate $Z^{(100)}$ would be: 1071.3757514953613 seconds

# 4 Blocked Gibbs-Sampling Algorithm

The bottleneck in running the Naive Gibbs sampler stems from sampling each single pixel $Z_{ij}$ with the values of all others held fixed. To speed up the sampling process, we implement an improvement known as blocked Gibbs sampling. Specifically, define two subsets of the pixels $\mathcal{I}_{\text{even}} = \{(i,j) : (i+j) \text{ is even}\}$ and $\mathcal{I}_{\text{odd}} = \{(i,j) : (i+j) \text{ is odd}\}$ The blocked Gibbs sampler proceeds as follows:

- Initialize $Z^{(0)} = X$.

- For $t = 1, ..., T$:
  - Let $Z = Z^{(t-1)}$
  - Let $\delta$ be an n x m matrix with $\mathcal{N}\left(0, \frac{1}{a+\tau}\right)$ entries
  - For $(i, j) \in \mathcal{I}_{\text{even}}$:
    - Let $S_{ij} = \sum_{(i',j') \in N_{(i,j)}} Z_{i'j'}$
  - Update $Z_{\mathcal{I}_{\text{even}}} = \frac{\tau}{a+\tau} X_{\mathcal{I}_{\text{even}}} + \frac{b}{a+\tau} S_{\mathcal{I}_{\text{even}}} + \Delta_{\mathcal{I}_{\text{even}}}$
  - For $(i, j) \in \mathcal{I}_{\text{odd}}$:
    - Let $S_{ij} = \sum_{(i',j') \in N_{(i,j)}} Z_{i'j'}$
  - Update $Z_{\mathcal{I}_{\text{odd}}} = \frac{\tau}{a+\tau} X_{\mathcal{I}_{\text{odd}}} + \frac{b}{a+\tau} S_{\mathcal{I}_{\text{odd}}} + \Delta_{\mathcal{I}_{\text{odd}}}$
  - Let $Z^{(t)} = Z$

Hence, our algorithm is more efficient as the inner for-loops are now vectorized. Moreover, because of the way we have defined our neighborhoods, we can vectorize and calculate our even  odd pixels all at once because our even and odd $(i, j)$ entries will never be neighbors. Thus, when we calculate $S_{i,j}$ we will not be double computing an already updated entry.

We now implement the Blocked Gibbs-Sampler using a = 250, b = 62.5 and $\tau = 0.01$. Run our code for T = 100 iterations, and visualize the resulting image $Z^{(100)}$.

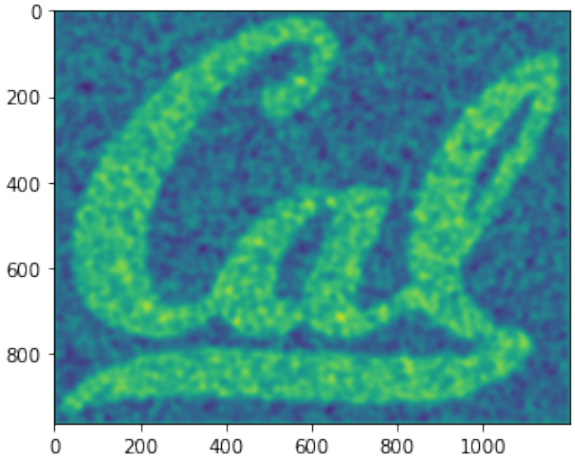


Figure 3: Figure 1.3: Blocked Gibbs Sampler Image Output

  - Gibbs Blocked Sampler took: 200.70132732391357 seconds

.

# References

[1] Stuart Geman and Donald Geman (1984). Stochastic relaxation, Gibbs distributions, and the Bayesian restoration of images. IEEE Transactions on Pattern Analysis and Machine Intelligence, (6), 721-741.