Gibbs Sampling for Grayscale Image Denoising

Lili Chen, Kevin Cao, Brian Wu, and Estella Chen

December 6, 2019

1 Introduction

We explore the use of MCMC algorithms to tackle the problem of image denoising. Specifically, we denoise small grayscale images using the Gibbs sampling MCMC algorithm, a version of Metropolis-Hastings where the proposed next state is always accepted. Then, we treat each possible assignment of color to each pixel as one state, such that in each iteration of Gibbs sampling, we pick a pixel and reassign it a value based on the conditional probability distribution of neighboring pixels. The main problem is constructing such a distribution that works well. For this project we started with strictly black and white images and adapted our algorithm to deal with k-bit grayscale images.

2 Methods

2.1 Theory

Preparing the Image

For computational tractability we resize the image to be 128 x 128 or 256 x 256 pixels. We also changed it to have 2^k colors for $k = 1, 3, 4$.

Adding Noise

To help us verify that our denoising works, we constructed test cases by adding our own noise to unnoisy images, and comparing the original image to our denoised noisy image. To add noise, for every pixel of the image we will sample from a normal centered at the actual pixel value with a variance σ^2 that we will pick for a certain experiment. We round the sample pixel, since images require discrete pixel values.

Likelihood

Let X_i denote the color of pixel i. We use the equation

$$\mathbb{P}(X_{\text{observed}}|X_{\text{observed neighbors}}) = \frac{\mathbb{P}(X_{\text{observed}}|X_{\text{proposed}}) \cdot \prod_{n\in\text{neighbors}(X)} \mathbb{P}(X_{\text{proposed}}|X_{\text{n}})}{\sum \mathbb{P}(X_{\text{observed}}|X_{\text{proposed}}) \cdot \prod_{n\in\text{neighbors}(X)} \mathbb{P}(X_{\text{proposed}}|X_{\text{n}})} \tag{1}$$

and assign probabilities to X_{proposed} taking on any value. If $\mathbb{P}(X_{\text{observed}}|X_{\text{observed neighbors}})$ is large for a particular value of X_{proposed}, then in the distribution we use for Gibbs sampling, $\mathbb{P}(X_{\text{proposed}})$ is also large. However, because it is difficult to directly determine the conditional probabilities in 1, we use edge potentials that are proportional to the probabilities as stand ins. For a given pixel, we sample from its distribution of potential values.

Binary Images

We represent binary images as an array of 1's and -1's where 1's represent white pixels and -1's represent black. To convert from full color images to binary images we use PIL to convert images to grayscale and appropriately assign the 8 bit values to -1 and 1 depending on whether they are closer to 0 or 255.

Binary Edge Potential

For binary edges we use the edge potential

$$E(X_{\text{a}}, X_{\text{n}}) = e^{J \cdot X_a \cdot X_n} \tag{2}$$

Where J is a constant that we will set to control the strength of our denoising algorithm. When $X_a = X_n$, we have $(J \cdot X_a \cdot X_b) = 1$. When $X_a \neq X_n$, we have $J \cdot X_a \cdot X_b = -1$. Thus $E(X_a, X_b)$ is large when $X_a = X_b$, and small otherwise. [1]

Grayscale Images

We represent n bit grayscale images as an array of values between 0 and $2^n - 1$. The conversion from color images to n bit grayscale can be found in the code.

Grayscale Edge Potential

For grayscale images we will have to tweak our edge potential to handle nonbinary pixel values. To do this we modify our edge potential to be

$$E(X_{\text{a}}, X_{\text{n}}) = e^{J \cdot (\frac{\text{numcolors}}{2} - |X_{\text{a}} - X_{\text{n}}|)} \tag{3}$$

The range of $E(X_a, X_n)$ is approximately $e^{-J \cdot \frac{\text{numcolors}}{2}}$ to $e^{J \cdot \frac{\text{numcolors}}{2}}$ The exponent is large when the examined pixel has value close to its neighbor, and small if the values are farther apart.

Potential

To form the total potential of a pixel value we take the product of neighboring edge potentials.

$$\text{Pot}(X_{\text{a}}) = \prod_{n \in \text{neighbors}(a)} E(X_a, X_n) \tag{4}$$

The potential of a pixel is proportional to $\mathbb{P}(X_{\text{proposed}}|X_{\text{observed neighbors}})$ and so can be used as a proxy when picking the pixel value that maximizes $\mathbb{P}(X_{\text{observed}}|X_{\text{observed neighbors}})$.

Denoising

Once we have defined edge potentials, we can begin to denoise the image by sampling pixels. For each pixel, we form probabilities for every possible value it can take on, i.e. values in the range $[0, 2^n)$. The probability that the true value of the pixel is a certain value X_a is the probability density of seeing the observed value X_{observed} given that the actual value of the pixel is X_a, i.e. the pdf at X_a of a Gaussian centered at X_a with arbitrary variance that we set to 1, weighted by the product of the edge potentials of the pixel's neighbors. We then sample from this set of all X_a to determine the final proposed value for the pixel.

$$\mathbb{P}(X_a) = \mathcal{N}(X_a|X_{\text{observed}}, 1) \cdot \text{Pot}(X_a) = \mathcal{N}(X_a|X_{\text{observed}}, 1) \cdot \prod_{n \in \text{neighbors}(a)} E(X_a, X_n) \tag{5}$$

A note on notation: $\mathcal{N}(X_a|X_{\text{observed}}, 1)$ is the same as $\phi(\frac{X_a - X_{\text{observed}}}{1})$, where ϕ is the pdf of the standard Gaussian.

Because we assume random noise, we can model each pixel as a Gaussian. However as mentioned above, we do not know the variance of the noise and use 1 as a stand in. We found that this gives us results that are quite good. The variance needs to be large enough so that the pixels will actually change in the denoising process, but not too large since it is unlikely for a pixel to have changed to a completely different color as a result of noise. We found that 1 works pretty well. In the future we could do more experiments to find the best variance to set.

2.2 Code

Gibbs resampling of a random pixel from a distrubution based on neighboring pixels:

```
for _ in range(100000):
    i = random.randint(0, imagesize)
    j = random.randint(0, imagesize)
    pot = []
    for x in range(num_colors):
        edge_pot = edge(x, i, j)
        pot.append(edge_pot)
    total = sum([pot[x] * a[int(ising[i, j])][x] for x in range(num_colors)])
    probs = [pot[x] * a[int(ising[i, j])][x]/total for x in range(num_colors)]
    ising[i, j] = np.random.choice(range(num_colors), p = probs)
```

3 Experiments

We began with a black and white version where we drew a black circle on a white background, added noise, and then tried to denoise it. We then moved on to expanding to grayscale. Since real grayscale has 256 colors and would take too long for the code to run enough to see any meaningful changes, we used a smaller case of 8 grayscale colors. However, we immediately ran into some problems. When we were working with the black and white version, we added noise based on a probability p. This is not plausible with multiple colors, so we used the normal distribution to add noise, which accurately reflects most real-world image noise. Also, the edge potential had to be calculated differently since there were now eight different colors. If a pixel has neighbors of a certain color, it is more likely to be one of the colors near that color. Previously, we were only working with black and white; if a pixel had more white neighbors, it would be more likely to be white than black. With more colors, we needed a way to see if the pixel was more likely to be white, light gray, dark gray, or black (to list a few). Therefore, we defined a new function for edge potential, which took into consideration how close the colors were. We represented colors as 0, 1, 2, 3, 4, 5, 6, and 7, where 7 was white and 0 was black. With this scale, we were able to calculate how similar the colors were and determine which color the pixel was most likely to be. Additionally, while we primarily focused on Gaussian noise, we also experimented with impulsive noise, which randomly scatters black and white pixels (often occurs in analog-to-digital converters). Since our algorithm samples based on neighbors, we found that it very effectively removes impulsive noise, as extreme pixel values will quickly be sampled back to close to where they were originally.

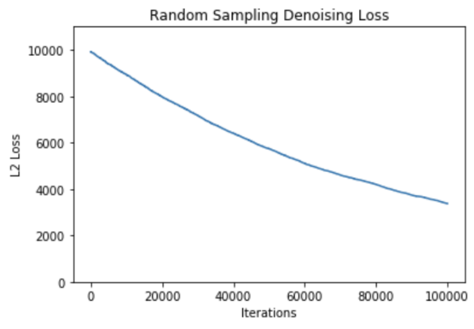
4 Results and Analysis

Evaluation

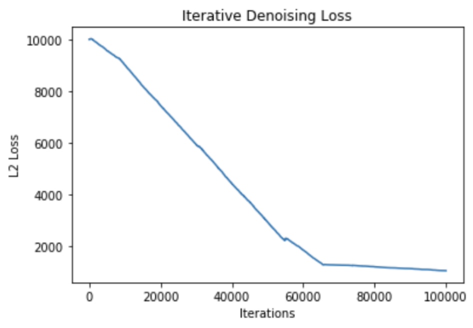
We tracked the L2 loss between the denoised image and the original image after every iteration.

Iterative Vs Random Sampling

We denoise the image one pixel at a time. There are two primary orderings of pixels - one is random sampling (random scan Gibbs sampling) from the entire image for each iteration. Another is a scan through every pixel in the image, evaluating every pixel column by column, row by row (systematic scan Gibbs sampling). [2] We then compared the loss curves for both methods.



(a) Loss curve from random sampling.



(b) Loss curve from iterative sampling.

Figure 1: Loss curves.

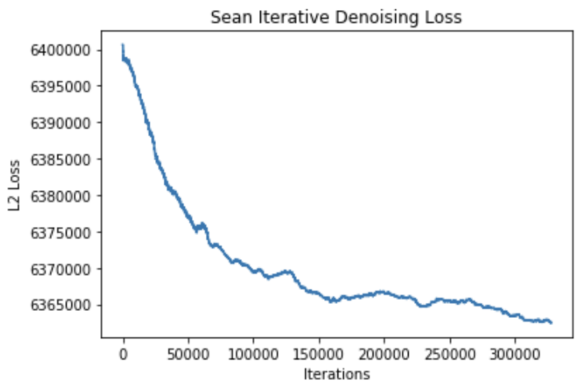


Figure 2: Sean Image Loss Curve. See Figure 5

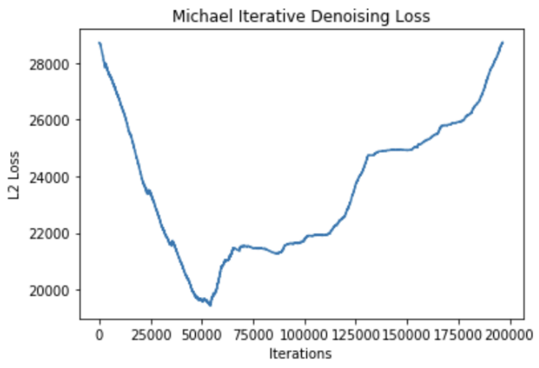
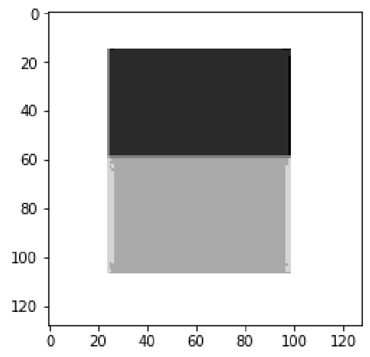


Figure 3: Michael Image Loss Curve. See Figure 6

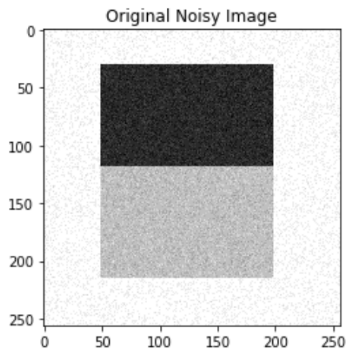
Looking at Figure 1, we find that iteratively sampling produced better results, as our denoising algorithm was able to converge faster. We also observed that the iterative sampling method loss curve converged sharply, while the random sampling loss curve converged smoothly. This is expected as iterative sampling guarantees every pixel is evaluated after 256^2 iterations, while random sampling does not guarantee that every pixel has been examined.

Excessive Denoising

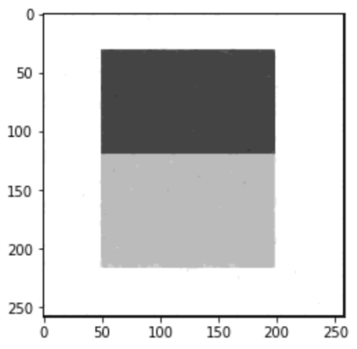
We ran 300,000 iterations of denoising on Sean (See Figure 5). This is equivalent to 5 passes over the 256 x 256 image. We found that the image loss converged early on around 100,000 iterations, and that further denoising does not give us much improvement in L2 loss. However qualitatively, we can see that excessive denoising resulted in a blurrier image, and increased banding artifacts. This is expected, since our algorithm tries to bring pixel values closer to those of its neighbors. This also tells us that maybe L2 loss is not the most accurate loss function to be using to evaluate the quality of the image, since images at 100,000 and 300,000 iteration have similar losses, but have different qualitative quality. Some further work can be done to find a better loss function for image denoising.



(a) Original square.



(b) Noisy square.



(c) Denoised square after 100000 iterations.

Figure 4: Lifetime of a square.

Stability

While for some scenes and noise parameters we saw converging loss, for others we found that our loss curve grew unstable after too many iterations. We think that some factors for instability are content of the scene and how much noise we injected into the image. We found better stability for simple scenes such as the box, where we have large patches of solid colors. The likelihood of exploding loss increases as the scene becomes more complex, such as some of our TA's profile pictures. We also saw unstable loss performance for scenes with high noise, for example converging loss for noise with $\sigma = 0.4$, but exploding loss for noise with $\sigma = 0.6$.

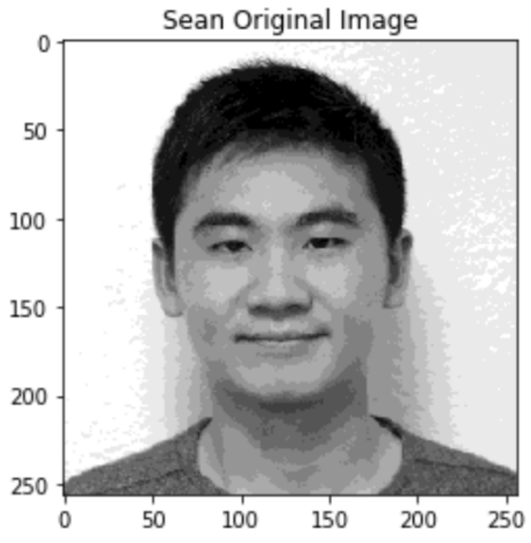
Video

https://youtu.be/HCSflFFoPgM

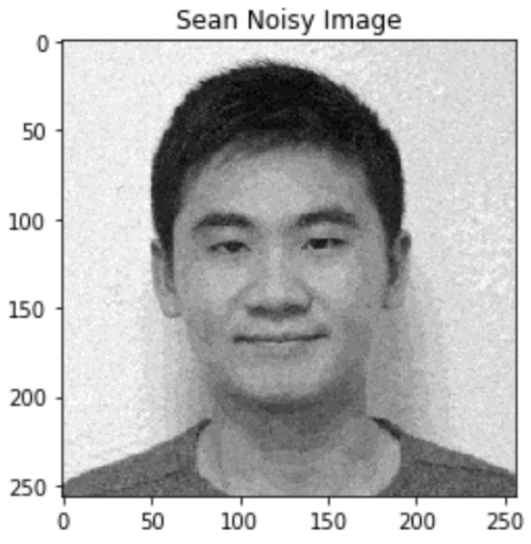
5 Limitations

The primary limitation with using Gibbs sampling for image denoising is the issue of computational tractability. Our code denoises images containing < 4 bit grayscale values, to allow a larger variety of shades than pure black and white. However, color images contain 3 channels of 8 bit values, and our method would not scale well to handle this. Our images also contain a relatively small number of pixels (128 x 128, 256 x 256) compared to the images we typically interact with. Another limitation with using Gibbs sampling is the inevitable imperfection of our denoising; evidently our images still contain some of the noise that we added and our technique didn't return some pixels to their original color. In addition, as we can see in some of the figures, our use of edge potential can lead to problems near the edges of an object. In order to use Gibbs sampling in more versatile and robust image denoising applications we would need to further tweak our parameters and explore efficiency tradeoffs.

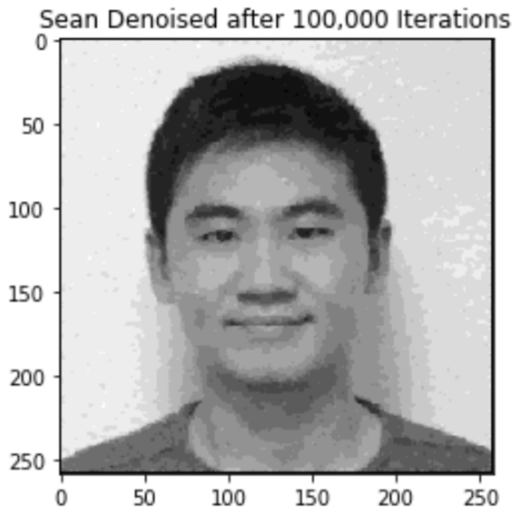
Appendix



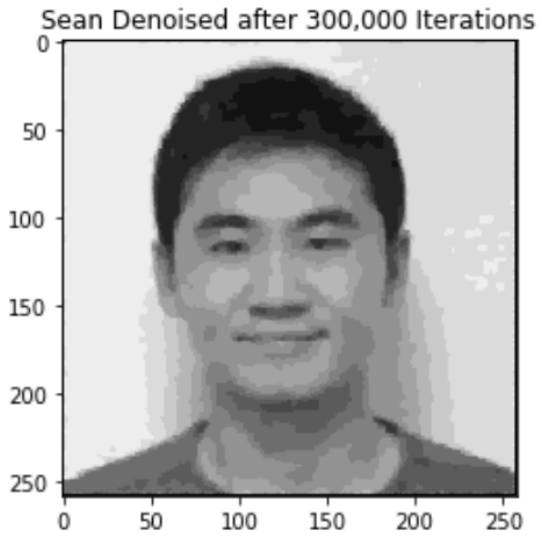
(a) Original Sean Image



(b) Noisy Sean Image with $\sigma = 0.4$

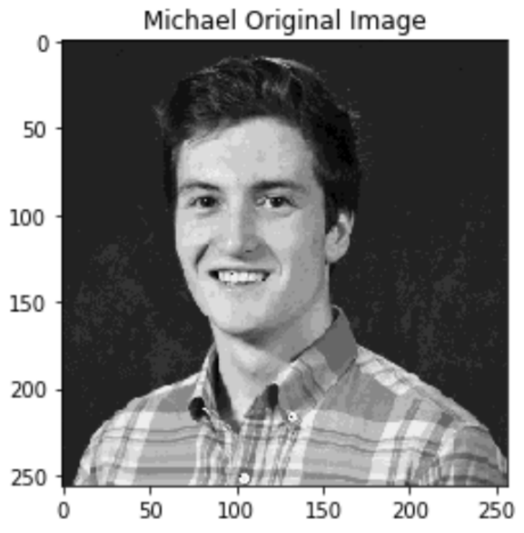


(c) Denoised Sean after 100,000 iterations

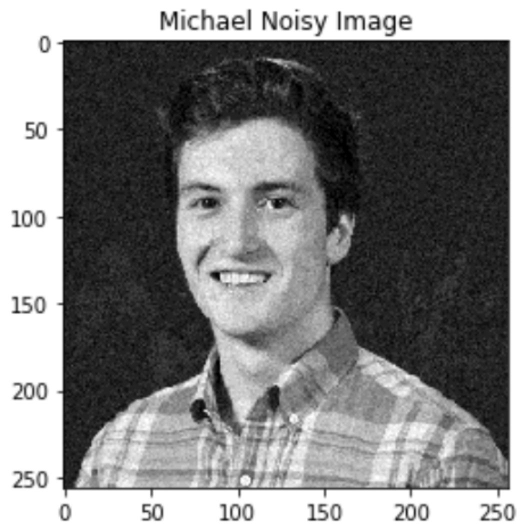


(d) Denoised Sean after 300,000 iterations

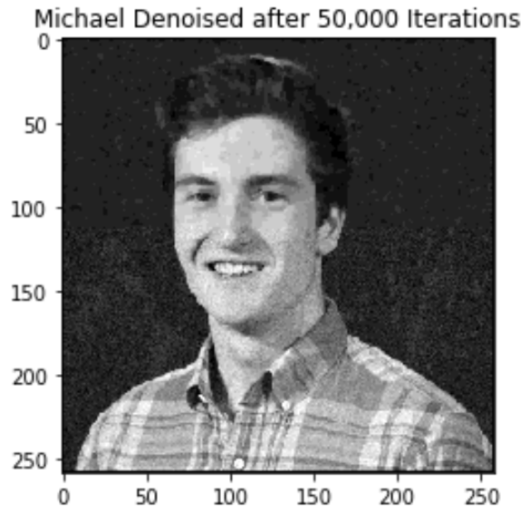
Figure 5: Sean



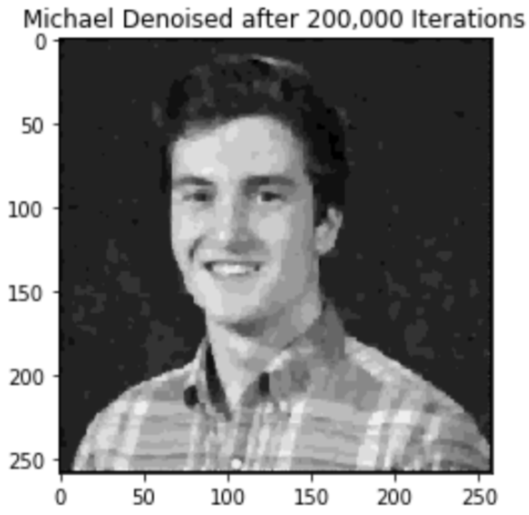
(a) Original Michael Image



(b) Noisy Michael Image with $\sigma = 0.6$



(c) Denoised Michael after 50,000 iterations



(d) Denoised Michael after 200,000 iterations

Figure 6: Michael

References

[1] https://towardsdatascience.com/image-denoising-by-mcmc-fc97adeaba9b

[2] https://ermongroup.github.io/cs323-notes/probabilistic/gibbs/