COMPUTATIONAL METHODS FOR DATA ANALYSIS COMPRESSED IMAGE RECOVERY

AVERY LEE

Applied Mathematics Department, University of Washington, Seattle, WA
leeave22@uw.edu
03/18/2022

Abstract. Compressed Image Recovery creates a Lasso regression model that utilizes sparsity and compressibility to recover a corrupted image that has a certain amount of random pixels missing. Discrete cosine transform will be used in the process. This paper will explain image compression and compressed image recovery, and go through the mathematical theories behind the calculation methods, the algorithms used, the final computational results, and conclusions.

1. Introduction and Overview

Natural images often use sparse recovery methods as it is useful when a matrix has a lot of values that do not significantly impact an outcome [1]. In terms of a regression model, this means that there are a lot of variables that do not impact the output as much as other variables. Lasso regression in particular can be utilized to find a sparse solution to regression problems as it decides which features are the most active for the model. In this paper, a corrupt image will be recovered through image compression using discrete cosine transform (DCT), and compressed image recovery using Lasso regression.

Section 2 of this paper will go over the mathematical background necessary to understand the computations. Section 3 will cover the algorithms and Python software packages. Then Section 4 will summarize the results from the computations, and Section 5 will summarize the learnings.

2. Theoretical Background

2.1. Sparsity.

Sparsity is utilized when many of the values in a matrix are insignificant and do not usefully impact the outcome of a calculation, meaning the value is 0. This is good for natural image processing. In mathematical terms, this is when $M < N$ in

$$A\beta = y \qquad\qquad A \in \mathbb{R}^{M\times N}, \beta \in \mathbb{R}^N, y \in \mathbb{R}^M$$

This means the system is underdetermined and cannot be solved uniquely. But if a dataset is sparse, the unimportant values can be identified and removed to get a system that hopefully is solvable. Using the number of non-zero values s, this is often called s-sparse. This is when the Lasso optimization approach comes into play.

2.2. Lasso Regression Model.

To account for the underdetermination of the system, a penalty or regularization component using norm 1 (not norm 2 like some other optimization methods), $\lambda||\beta||_1$, can be added to the normal minimization equation, so it would look like this:

$$\min_{\beta} ||A\beta - y||_2^2 + \lambda||\beta||_1 \qquad \hat{f}(x) = \sum_{j\in\{j|\hat{\beta}_j\neq 0\}} \hat{\beta}_j \psi_j(x)$$

This way, function $\hat{f}(x)$ only depends on s number of important features $\psi_j(x)$ with coefficient $\hat{\beta}_j$ [2]. Basically, Lasso determines which features are important to use. However, it is very important to note that Lasso will help identify important features for a given model and dataset, but these outcomes might change depending on the model or training dataset. So conclusions need to be drawn carefully.

Another way to write this optimization problem, which will be used in the computing section, is

$$\min_{x\in\mathbb{R}^N} ||x||_1 \qquad \text{subject to } Ax = y$$

where x is the minimizer, or the DCT vector, of image F.

2.3. **Discrete Cosine Transform (DCT).**

Before studying DCT, we need to understand the basic Fourier Transform. A Fourier Transform (FT) breaks a signal down into frequencies, where the discrete signal is often written as f. A Fourier Transform of f is often written as $\hat{f}$ and is a complex-valued function. The equation is written below, where e^{-ikx} represents the Fourier kernel [3].

$$F(f) = \hat{f}(\xi) = \frac{1}{\sqrt{2\pi}} \int_{-\infty}^{\infty} f(x)e^{-i\xi x}dx, \qquad e^{-i\xi x} = cos(\xi x) + isin(\xi x)$$

An inverse FT is also possible; in other words, turn $\hat{f}$ into f, with this equation below.

$$F^{-1}(\hat{f}) = f(x) = \frac{1}{\sqrt{2\pi}} \int_{-\infty}^{\infty} \hat{f}(\xi)e^{i\xi x}d\xi$$

A Discrete Fourier Transform (DFT) basically conducts Fourier Transform over a finite interval given equally spaced data points, with the given equation below.

$$\hat{x}_k = \frac{1}{N}\sum_{n=0}^{N-1} x_n e^{\frac{2\pi ikn}{N}}$$

Now that we understand DFT, we can observe Discrete Cosine Transform (DCT). DCT takes the real (not imaginary) part of the FT of f, meaning it becomes a sum of cosines, as shown below.

$$DCT(f)_k = \sqrt{\frac{1}{K}}\left[f_0 cos\left(\frac{\pi k}{2K}\right) + \sqrt{2}\sum_{j=1}^{K-1} f_j cos\left(\frac{\pi k(2j+1)}{2K}\right)\right]$$

Similar to an inverse FT, an inverse DCT reconstructs f from the DCT of f. The DCT and inverse DCT can be represented as a matrix D and D^{-1}. $vec(F)$ means the pixels of the image F have been "flattened", or vectorized. Then we can get these formulas:

$$Dvec(F) = vec(DCT(F)) \qquad D^{-1}vec(DCT(F)) = vec(F)$$

As explained earlier, natural images have sparse characteristics. This means many of the DCT coefficients are not extremely significant for image recovery. We can explore the top coefficients, meaning the highest absolute coefficient values. This will be done in Section 4.

3. Algorithm Implementation and Development

3.1. **Python Packages and Notable Functions.**

Python was used to compute the algorithms and produce plots. Below is a list of packages used.

`google.colab`: Gives access to the user's Google Drive account where they can access the data.
`cvxpy [4]`: Helps with computations for convex optimization problems.
`scikit-image [5]`: Allows display and processing of images.
`numpy [6]`: Allows usage of multidimensional arrays and functions to manipulate those arrays.
`scipy [7]`: Helps with scientific computing like discrete cosine transform.
`matplotlib.pyplot [8]`: Useful for graphics manipulation.
`cvx: Variable(), Minimize(), Problem(), solve()`: Functions in cvxpy used for optimization calculations given the objective and constraints.

3.2. **Setup.**

The given image (The Son of Man by Rene Magritte) has size 292x228. However, in order to reduce computing time during optimization, this will be reduced to 53x41. Also, grayscale will be used throughout to observe changes. This matrix representing the image is now vectorized; let's call this vecF. The number of pixels in the image N is dimension of rows N_y multiplied by dimension of columns N_x.

3.3. **Image Compression.**

Now, the DCT matrix D and inverse DCT matrix iD can be computed using the $dct()$ and $idct()$ functions in $scipy.fftpack$ in Python. This uses the DCT equation mentioned in Section 2.

As discussed previously, we can get a vector of DCT(F) using $vec(DCT(F)) = D * vecF$. This can be plotted to investigate the values (coefficients in the model) that are important or unimportant. To test how this works, a threshold of 5%, 10%, 20%, and 40% will be tested, where we only keep the top values within each threshold by absolute value, and change the rest to 0. Essentially, the less important values are getting removed. The new image is computed using $vecF = iD * vec(DCT(F))$. Now, the image is compressed.

3.4. **Compressed Image Recovery.**

Since we already have the full image that is not corrupt, we will select random observations of pixels to try to recover the image F. To construct a random measurement matrix B, $M = rN$ number of random rows of the identity matrix $I \in \mathbb{R}^{N\times N}$ will be chosen. We will try 3 trials each for $r = 0.2, 0.4, 0.6$ to make sure the random picks aren't too lucky or unlucky, and observe the change as r increases. The vector of measurements y is computed with $y = B * vecF$. Random pixels are selected.

Now, to solve the optimization problem mentioned in Section 2, we define $A = B * iD$, and the objective is to minimize $||x||_1$ subject to $Ax = y$ (the constraint). $cvxpy.Problem()$ can be used with parameters containing the objective and constraint, then solve the optimization using $solve()$. It can take a long time to converge, so having these parameters is helpful: solve(verbose=True, solver='CVXOPT', max_iter=1000, reltol=1e-2, featol=1e-2). With this, the new vecF can be computed again using $vecF = iD * x^*$ where x^* is the output DCT vector of the image.

4. Computational Results

4.1. **Image Compression.**

This is the original image we can compare to.

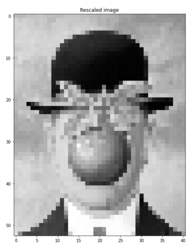

Figure 1. Original Image (53×41 pixels)

From Figure 2, we can observe that as predicted, not all the values are significant and are very close to zero; we can conclude that this is sparse and has room for compression. Since the absolute values are observed, the absolute values can also be plotted (on the right).

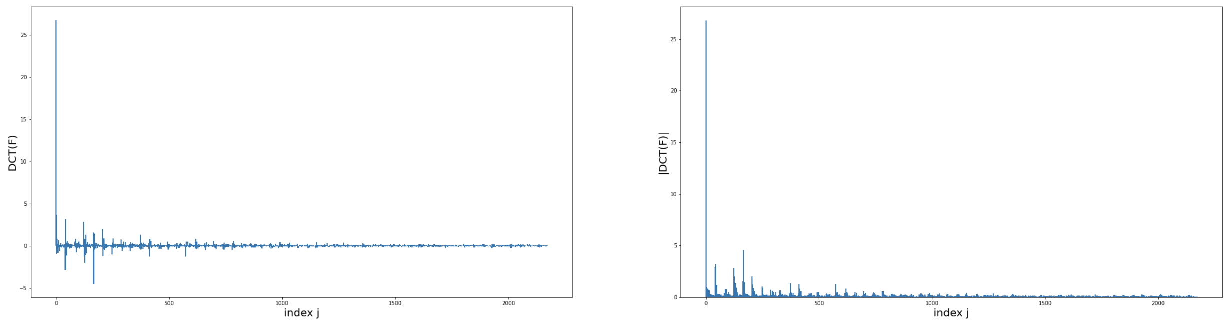

Figure 2. Discrete Cosine Transform Coefficients

We get the images below when we only use the top 5%, 10%, 20%, and 40% coefficients (by absolute value). Obviously, the more coefficients we use, the clearer the image gets. We can see that by just using 40% of the top coefficients, it still looks very similar to the original image.

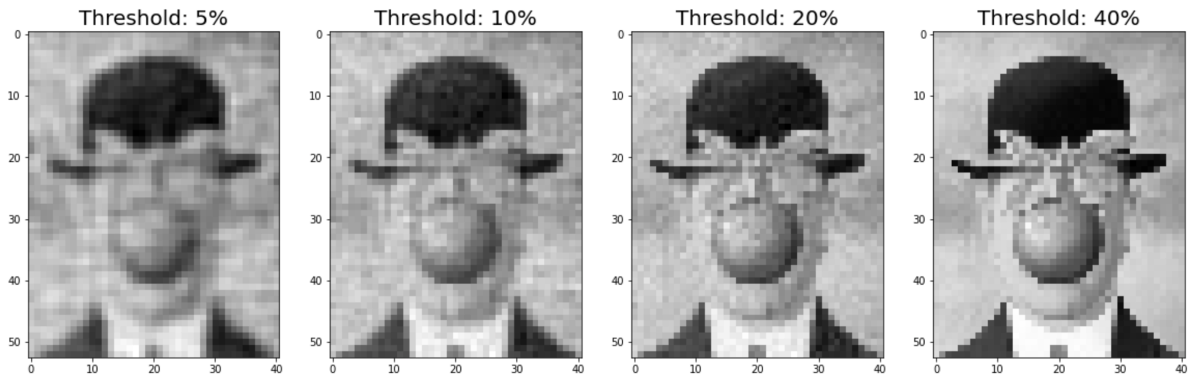

Figure 3. Images using Top n% of DCT Coefficients

4.2. **Compressed Image Recovery.**

Now following the algorithm to pick random pixels and restore the image, we get these recovery results. Each r parameter goes through 3 trials to make sure that we didn't get too lucky or unlucky with the randomly chosen pixels. We can see this come into play with Trial 1 of r=0.2 and Trial 3 of r=0.2; in Trial 1, it seems the image is blurrier than in Trial 3 that has a general outline of the man's face.

As predicted, the higher the r, the clearer the image. This makes sense since a higher r means more random pixels are selected from the start.

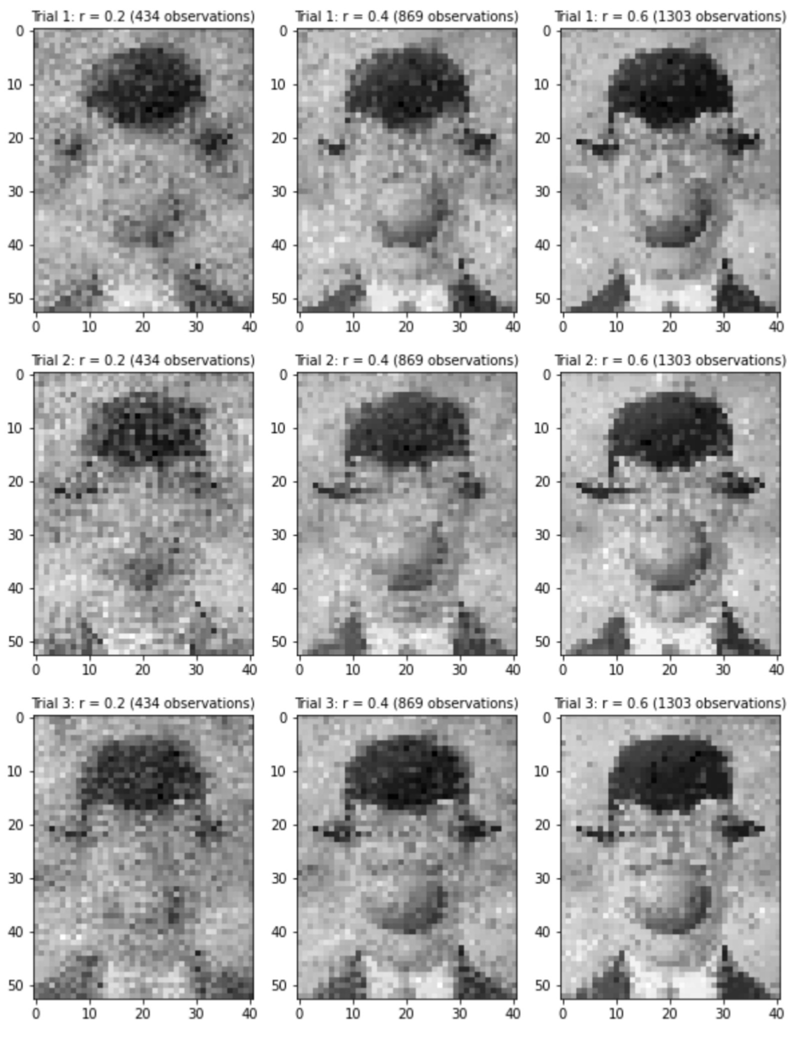

FIGURE 4. Recovered Images using r = 0.2, 0.4, 0.6 (3 Trials each)

4.3. **A Mysterious Image.**

At the end of the day, all this comes down to is matrix manipulation. For example, given only a random measurement matrix B and vector of measurements y, a "mysterious" image can be recovered. In our case, it will be an image of size 50×50 pixels. This is what we get!

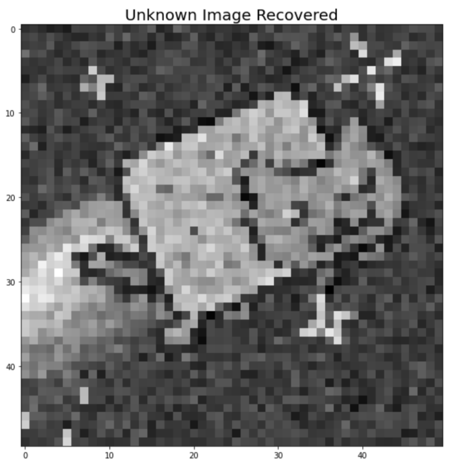

Figure 5. Recovered Image Only Given B and y

5. Summary and Conclusions

In this paper, sparsity, the Lasso regression model, and Discrete Cosine Transform (DCT) were used to investigate compressibility of an image, recover an image given random pixels, and recover a mysterious image only given its measurement matrix and vector of measurements. As we saw in "A Mysterious Image" in Section 4, this algorithm is extremely flexible and scalable. Any image can be used for same purposes, but it may require extensive computing power and time depending on the size of the image. It is also important to note that these results can be specific for a particular model or dataset, so multiple models or datasets need to be used to make a general conclusion.

Acknowledgements

The author is thankful to TA Katherine Owens for the general rundown and clarifications on calculations and code, and Professor Bamdad Hosseini for teaching most of the mathematical concepts mentioned in the paper.

References

[1] Bamdad Hosseini. Introduction to sparse recovery. University of Washington (LOW 216), March 2022.
[2] Bamdad Hosseini. Model selection with lasso. University of Washington (LOW 216), March 2022.
[3] Bamdad Hosseini. Signal processing with dft. University of Washington (LOW 216), Jan 2022.
[4] Steven Diamond and Stephen Boyd. CVXPY: A Python-embedded modeling language for convex optimization. *Journal of Machine Learning Research*, 17(83):1–5, 2016.
[5] Stefan Van der Walt, Johannes L Schönberger, Juan Nunez-Iglesias, François Boulogne, Joshua D Warner, Neil Yager, Emmanuelle Gouillart, and Tony Yu. scikit-image: image processing in python. *PeerJ*, 2:e453, 2014.
[6] Charles R. Harris, K. Jarrod Millman, Stéfan J. van der Walt, Ralf Gommers, Pauli Virtanen, David Cournapeau, Eric Wieser, Julian Taylor, Sebastian Berg, Nathaniel J. Smith, Robert Kern, Matti Picus, Stephan Hoyer, Marten H. van Kerkwijk, Matthew Brett, Allan Haldane, Jaime Fernández del Río, Mark Wiebe, Pearu Peterson, Pierre Gérard-Marchant, Kevin Sheppard, Tyler Reddy, Warren Weckesser, Hameer Abbasi, Christoph Gohlke, and Travis E. Oliphant. Array programming with NumPy. *Nature*, 585(7825):357–362, September 2020.
[7] Pauli Virtanen, Ralf Gommers, Travis E. Oliphant, Matt Haberland, Tyler Reddy, David Cournapeau, Evgeni Burovski, Pearu Peterson, Warren Weckesser, Jonathan Bright, Stéfan J. van der Walt, Matthew Brett, Joshua Wilson, K. Jarrod Millman, Nikolay Mayorov, Andrew R. J. Nelson, Eric Jones, Robert Kern, Eric Larson, C J Carey, İlhan Polat, Yu Feng, Eric W. Moore, Jake VanderPlas, Denis Laxalde, Josef Perktold, Robert Cimrman, Ian Henriksen, E. A. Quintero, Charles R. Harris, Anne M. Archibald, Antônio H. Ribeiro, Fabian Pedregosa, Paul van Mulbregt, and SciPy 1.0 Contributors. SciPy 1.0: Fundamental Algorithms for Scientific Computing in Python. *Nature Methods*, 17:261–272, 2020.
[8] J. D. Hunter. Matplotlib: A 2d graphics environment. *Computing in Science & Engineering*, 9(3):90–95, 2007.