Final Report for Implementation of “Efficient Sparse Coding Algorithms” by Lee et al.

Weixi Chen, Fanchen Kong, Haoyu Zha

Part1: Background

Nowadays, object recognition has always been a central research topic in machine learning such as computer vision. Unlike human-beings who perform object recognition effortlessly and instantaneously, machines have great difficulties recognizing objects in real world from images using object model. The overall object recognition problem is based on models of known objects. Formally, given an image containing one or more objects of interest and a set of labels corresponding to a set of models known to the system, the system should assign correct labels to regions, or a set of regions, in the image. A typical approach for object recognition will include the following steps: image representation, feature extraction, feature- model matching, hypotheses formation and finally object verification.

Sparse coding plays important part in encoding part of the whole process. It provides a class of algorithm for finding succinct representation given the unlabeled input data. It will finally get a convergent basis codebook that capture higher-level features in the data. Moreover, sparse coding can be applied to learning over complete basis set. Despite its advantages, the computation cost is huge. *Lee et al.* proposed an efficient algorithm on finding the sparse representation and a codebook by alternating optimization over two subsets, namely, L1-regularized least squares problem optimizing coefficient sparse matrix with fixed basis B, and L2-constrained least squares problem solving bases B with fixed coefficients S. In our project, we implement the algorithms in python, compare results with those from the Matlab code by the author. On top of the code migration, we also use some optimization techniques to further improve the efficiency of the algorithm, and as a result, we are able to achieve competing accuracy and significant speed improvement compared to the original Matlab implementation.

Part2: Algorithm Description

The goal of sparse coding is to represent each input vector $\vec{\xi} \in \mathbb{R}^k$ approximately using basis vectors $\overrightarrow{b_1}, \overrightarrow{b_2}, \dots \overrightarrow{b_n} \in \mathbb{R}^k$ and a sparse vector of weights such that $\vec{s} \in \mathbb{R}^n$ such that $\vec{\xi} \approx \sum_j \vec{b}_j s_j$. The prior distribution of each coefficient is defined as $\mathrm{P}(s_j) \propto \exp(-\beta\emptyset(s_j))$. For our experiment, we use L1 penalty function $\emptyset(s_j) = \|s_j\|_1$ as our sparsity function and β is a constant. Then we write our problem

$$\min_{\{b_j\},\{s^{(i)}\}} \sum_{i=1}^{m} \frac{1}{2\sigma^2} ||\xi^{(i)} - \sum_{j=1}^{n} b_j s_j^{(i)}||^2 + \beta \sum_{i=1}^{m} \sum_{j=1}^{n} \phi(s_j^{(i)})$$

$$\text{Subject to } ||b_j||^2 \leq c, \forall j = 1, \ldots\ldots n$$

(1)

in matrix form

$$\min_{B,S} \frac{1}{2\sigma^2} ||X - BS||_F^2 + \beta \sum_{i,j} \phi(S_{i,j})$$

$$\text{Subject to } \textstyle\sum_i B_{i,j}^2 \leq c, \forall j = 1, \ldots\ldots n$$

(2)

$X \in \mathbb{R}^{k \times m}$ is the input matrix, $B \in \mathbb{R}^{k \times n}$ is the basis matrix, and $S \in \mathbb{R}^{n \times m}$ is the coefficient matrix. When we use L1 penalty function, the optimization problem is convex in B when fixing and convex in S when fixing B, but not convex in both simultaneously. Therefore, we alternatively optimizing with respect to bases B and coefficient S while holding the other fixed iteratively to reach optimal B and S. We use feature-sign search algorithm to solve the L1- regularized least squares problem over coefficient $\{s_j^{(i)}\}$ of sparse coding.

$$\min_{\vec{s}} || \xi^{(i)} - \sum_j b_j s_j^{(i)}||^2 + (2\sigma^2\beta) \sum_j |s_j^{(i)}|$$

(3)

Considering only nonzero coefficients, this reduce (3) into a standard, unconstrained quadratic optimization problem:

$$\min_x f(x) \equiv ||y - Ax||^2 + \gamma ||x||_1$$

(4)

The feature sign maintains an active set of potential nonzero coefficient and their corresponding signs while all other coefficient must be zero, then we search for the optimal active set and coefficient sign. It begin with initializing $x := 0, \theta := 0, and\ active\ set\ := \{\ \ \}\ where\ \theta_i \in \{-1,0,1\}\ denote\ the\ sign,$ selecting $i =$ arg $max_i \left| \frac{\partial ||y - Ax||^2}{\partial x_i} \right|$ and activating x_i if it locally improve the objective.

Then it proceeds in a series of "feature-sign steps": on each step, it is given a current guess for the active set and the signs: and it computes the analytical solution $\hat{x}^{new}$ to the resulting unconstrained QP: $\hat{x}^{new} := (\hat{A}^T\hat{A})^{-1}(\hat{A}^T y - \frac{\gamma\hat{\theta}}{2})$

It then updates the solution, the active set and the signs using an efficient discrete line search

between the current solution and $\hat{x}^{new}$, and removes zero coefficient of $\hat{x}$ from the active set and update θ. Finally it check the optimality conditions for zero coefficient and nonzero coefficient to iterate each step. The overall algorism will always converge after each iteration reducing the objective of f(x).
We use Lagrange dual algorithm for to solve L2-constrained least squares problem over learning bases.

$$\min ||\mathrm{X} - \mathrm{BS}||_F^2$$
$$\text{Subject to } \sum_{i=1}^{k} B_{i,j}^2 \le c, \forall j = 1, \ldots .. n \tag{5}$$

This constrained optimization problem can be solved efficiently using a Lagrange dual:

$$\mathcal{D}(\lambda) = \min_B \mathcal{L}(\mathrm{B}, \lambda) = trace(X^T X - XS^T (SS^T + \wedge)^{-1} (XS^T)^T - c \wedge) \tag{6}$$

After maximizing $\mathcal{D}(\lambda)$, we get the optimal base B:

$$B^T = (SS^T + \wedge)^{-1} (XS^T)^T \tag{7}$$

Part3: Results and Analysis

The experiment is conducted on an Intel Core i5-3470 Quad-Core 3.2GHz, 4G memory, Ubuntu 15.10 machine, the Python interpreter version is 2.7, and the Matlab version is r2015a. Due to the limit of time, we only run 20 iterations. The following are three plots regarding the accuracy and the speed of our experiment.
Figure 3.1 is a plot of reconstructed error between obtained basis *B* and the collections of coding vectors *S*. It is defined as

$$E = \frac{\|X - BS\|_F}{\|X\|_F} \tag{7}$$

where *X* is the collection of original feature vectors, and $\|\cdot\|_F$is the Frobenius Norm. The reconstruction error drops significantly after the first 10 iterations, and gradually stabilize. However, as we could see that the lower bound of the reconstructed error stays above 0.6. Two reasons cause this situation. First, the codebook is only 196*128 dimension, there is a natural limitation by the code book. Second, the penalty constant γ is 0.4, as provided in the Matlab implementation, which restricts the density of the coding each feature vector.

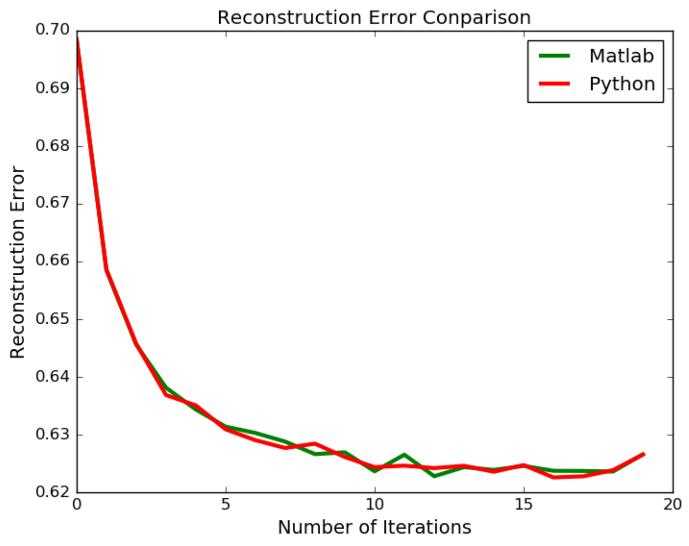

Figure 3.1 Reconstruction Error Comparison

Some fluctuations occur between the results from Matlab and those from Python. We investigate the internal computation results, and minimization from the l2 optimization problem results show slight variance in each round. Consequently, the codebook *B* is found different, and subsequent *S* and *B* are therefore affected. Figure 3.2 shows the the difference, defined by (8) and (9), propagates as the algorithm goes on, although the reconstructed errors maintained steadily.

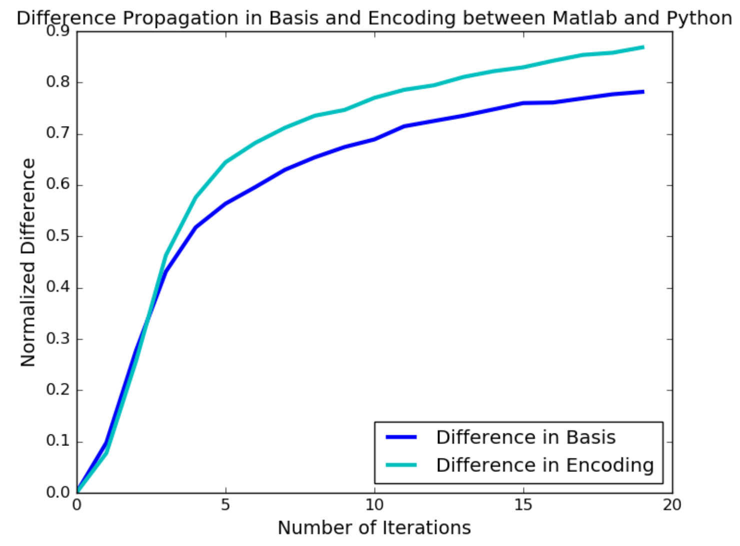

Figure 3.2 Difference Propagation in Basis and Encoding between Matlab and Python

$$D_B = \frac{\left\|B_{matlab} - B_{python}\right\|_F}{\left\|B_{matlab}\right\|_F} \tag{8}$$

$$D_S = \frac{\left\| S_{matlab} - S_{python} \right\|_F}{\| S_{matlab} \|_F}$$

(9)

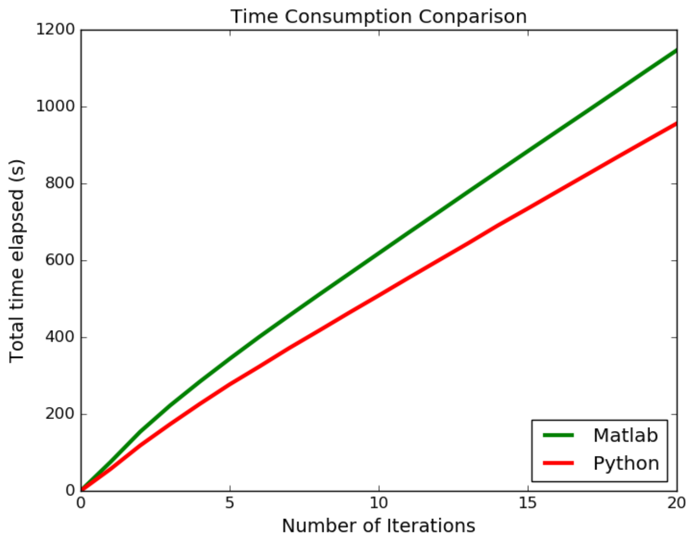

Figure 3.3 Time Consumption Consumption

Figure 3.3 is a plot of the time consumption by the Python implementation and MATLAB implementation for each iteration. The cause for the speed increase in python may credit to the underlying multi-threading math library. Since the CPU usage shows constant full loads on all four cores for the python implementation, and alternating full load on individual cores for Matlab implementation. This may suggest that Matlab is not using multi-threading. Figure 3.4 and 3.5 show the CPU usage statistics by the Python and Matlab implementation.

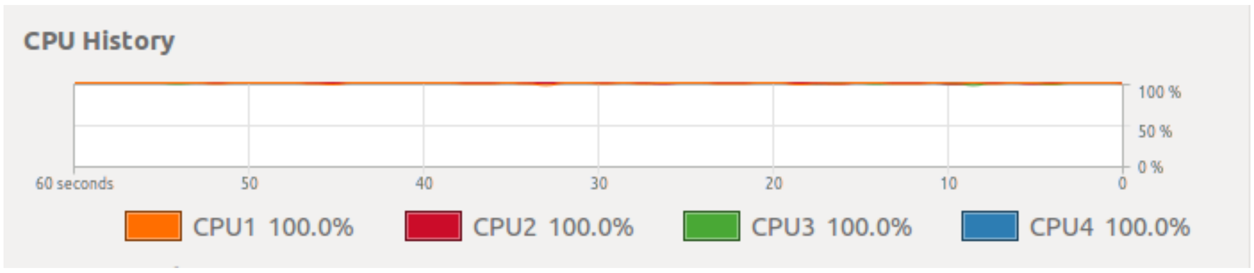

Figure 4.4 CPU History of Python Implementation

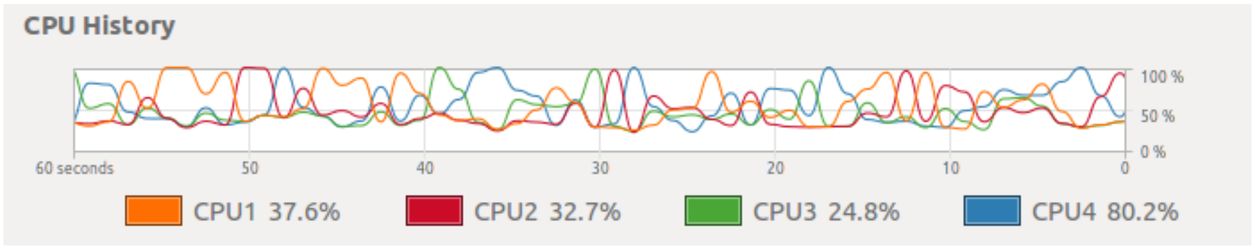

Figure 4.5 CPU History of Matlab Implementation

Figure 4.6 is a visualization of the codebook after 20 iterations. The codebook seems to capture simple line textures.

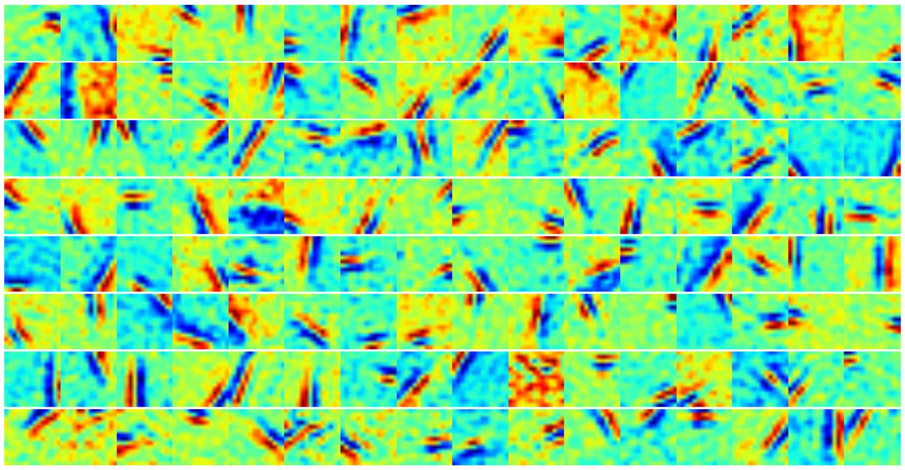

Figure 4.6 Visualization of the codebook after 20 iterations

Part4: Improvements

Several attempts are made to improve the efficiency in the Python implementation. As Python is inherently dynamic scripting language, and community-based, certain Matlab functions and data types are not supported by Python, so we have made some compromises when we implement in Python. Nevertheless, certain python features are exploited improve the overall efficiency of the algorithm, and with some successful attempts, we obtain descent improvements.

4.1 Using Non- Sparse Matrix

The Matlab implementation uses sparse matrix when never possible. However, in Python, whether to use sparse matrix and built-in multiplication function from Scipy depends on the density of the matrix. The density is defined as the number of nonzero terms divided by the the size of the matrix. For example, if a m*n matrix has a nonzero terms, then the density of the matrix is a/(m*n). We implement the multiplication of dense matrix D and sparse matrix S using different methods.

We create a random 1000*1000 dense matrix D and a random 1000*1000 sparse matrix with different density in COO format. Then I transform the sparse matrix from COO format to CSC format and LIL format. Results are shown in the Table 4.1 and Table 4.2.

Formats of Sparse Matrix	Formula	Speed for 10% density (average of 10 times)	Speed of 1% density (average of 10 times)	Speed of 0.1% density (average of 10 times)
COO Format	D*S	10 loops, best of 3: 135 ms per loop	10 loops, best of 3: 28.7 ms per loop	10 loops, best of 3: 17.4 ms per loop
CSC Format	D*S	10 loops, best of 3: 92.3 ms per loop	**10 loops, best of 3: 18 ms per loop**	**10 loops, best of 3: 8.48 ms per loop**
LIL Format	D*S	10 loops, best of 3: 131 ms per loop	10 loops, best of 3: 24 ms per loop	10 loops, best of 3: 11.6 ms per loop
toarray()	D.dot(SD) (SD = S.toarray())	**10 loops, best of 3: 27.4 ms per loop**	10 loops, best of 3: 27.1 ms per loop	10 loops, best of 3: 27.1 ms per loop

Table 4.1 Comparison of Dense Matrix * Sparse Matrix in Different Formats

Formats of Sparse Matrix	Formula	Speed for 10% density (average of 10 times)	Speed of 1% density (average of 10 times)	Speed of 0.1% density (average of 10 times)
COO Format	S*D	10 loops, best of 3: 138 ms per loop	10 loops, best of 3: 32.9 ms per loop	10 loops, best of 3: 20.5 ms per loop
CSC Format	S*D	10 loops, best of 3: 98 ms per loop	10 loops, best of 3: 14.2 ms per loop	10 loops, best of 3: 4.41 ms per loop
LIL Format	S*D	10 loops, best of 3: 103 ms per loop	10 loops, best of 3: 12.7 ms per loop	10 loops, best of 3: 5.08 ms per loop
toarray()	SD*S (SD = S.toarray())	**10 loops, best of 3: 4.4 ms per loop**	**10 loops, best of 3: 5.08 ms per loop**	**10 loops, best of 3: 4.01 ms per loop**

Table 4.2 Comparison of Sparse Matrix in Different Formats * Dense Matrix

The density of matrix is affected by the penalty value gamma used. In Lee's demo Matlab code, gamma is set to be 0.4, resulting sparse matrices of density between 14-10%. In our experiment, we employed the same gamma value in the demo code, therefore based on the results we obtained, it is rather more efficient using dense matrix for manipulation. In addition, sparse matrices have overhead to store, therefore, we use dense matrices as final saved format as well. For future study, we will compare the time and space complexity of sparse matrix manipulation of lower densities.

4.2 Parallelization

Since the vectors are processed one by one, there could be some room for parallelized processing that could further improve the speed of the algorithm. Two methods can be used, the python built-in multiprocessing module, and the openBLAS implementation of multi-threading matrix-vector multiplication. However, as python's multiprocessing module involving allocating separate memory space for variables which could cause some memory allocation issues, we decide to choose the openBLAS library. The openBLAS library has low-level support for parallelization, and as a result of profiling most time consuming part occurs in the process of matrix multiplication and solving, which are both very well improved by openBLAS library. We obtain full loads on all four cores of the CPU using openBLAS.

4.3 Faster Numerical Evaluation

One other attempt is using Numepxr library to speed up complex matrix evaluation. In the unit experiment we conducted involving only the matrix evaluation, it is about 3 times faster if the evaluation contains multiple matrix multiplication and addition. The underlying logic is that it gets rids of the intermediate steps between assigning temporary results to a variable. However, as we profile the our running time, most time spent is on solving the matrices and multiplication itself, cost adding the intermediate results was negligible. There for code readability and maintainability, we decide not to use Numexpr.

4.4 Array Creation

We store array in Fortran style because it allocates column-wise continuous memory. Compared to arrays in C style which has row-wise continuous memory allocation, arrays stored in Fortran style can be accessed and competed faster due to spatial proximity, thereby memory caching. However, for 1-D array, it does not matter to store in Fortran style or C style.