ECE 510: Mathematical Foundations of Machine Learning **Spring 2018**

Course Project: Sparse Image Reconstruction of Half-Tone Images

Student Names: Robert Holt, Scott Matsuo, Chris Neighbor

This was a group final project for my Mathematical Foundations of Machine Learning. Below are my sections and code for the project demonstrating how to convert binary images back into grey scale images using sparse dictionary reconstruction.

1 Algorithm Implementation & Testing

We have chosen to implement the K-SVD algorithm from [1] to the half-tone regression problem from [2]. We chose this algorithm because we found the presentation of it to be very clear and to be grounded firmly in topics which were covered throughout this semester. Additionally, the KSVD algorithm and presentation in paper [1] are heavily cited in other works (more than 8000 citations!) which we have interpreted to mean that it is well regarded and valuable for learning. We chose the half-tone problem because it could be treated similar to a denoising application and would have very visually presentable results which give a clear intuition for whether we are developing a functional algorithm. Additionally, the half-tone images looked impressively similar to the original image despite being binary. Apparently this technique of half-toning was developed for computer display screens which could only display binary values.

The algorithm was developed in Python 3 using Jupyter Notebooks, but was also condensed into .py files for clarity. Code will be presented in Appendix for the reviewer's convenience.

1.1 Results on Benchmark Data

For our benchmark data, we chose to use images which have been classically used for evaluating image processing algorithms. Our main criterion for determining how well our algorithm is able to successfully perform the non-linear mapping of the half-tone images to their original reference image is the peak signl-to-noise ration. The equation for the PSNR is:

$$PSNR = 10log_{10}\Big(\frac{1}{MSE}\Big)$$

Where MSE is the mean squared error of the reconstructed image and the ground truth image. The units of the PSNR are in dB. We also explored the difference norm calculated here as the square root of the sum of the squared error of the reconstructed image and the reference image. This would be equivalent to the Frobenius norm between the two image matrices.

We gather the data for the reference and half-tone images from the database described in [3]. While our algorithm could be applied to a larger subset, for the sake of computational time we constrained our task to training on the patches from one reference image and reconstructing the half-tone images. In our case we selected *lena* and *boat* in part due to our having a reference table in [2] for PSNR and to demonstrate that the dictionary can be used on test data which is seemingly unrelated to the original training data. In this case we are reconstructing an image of a boat using a dictionary learned from patches extracted from the image of a female face. This is in part due to the patch reconstruction formulation of the problem and the natural aspects of the images. The images are 512x512 and can be seen below next to their half-tone counterparts.

The half-tone images are generated using the Floyd-Steinberg algorithm, but knowledge of how this algorithm functions is not necessary in order to make the dictionary learning still effective.

The images are first normalized to be grayscale with values from 0 to 1. Then overlapping patches were extracted from the *lena* reference image and a shuffled subset were used to generate the dictionary using our

Figure 1: Reference images for *lena* and *boat* are on the left. The images on the right are the half-tone images and are binary matrices. This can best be viewed by zooming in on a computer. PSNR=6.8 between images

K-SVD code. Once the dictionary has been learned. The half-tone patches are extracted and the orthogonal matching pursuit algorithm is used to find the dictionary element which can best be used to reconstruct each patch. These reconstructed overlapping patches are then reshaped back into the original image size and each pixel value is averaged. For example a pixel in the middle of an image which has 10x10 patches extracted will have 100 different values from the different patches it was a part of and those values are then averaged. Some dip in performance can occur at the edges of the images since these pixels are not overlapped by as many patches. The parameters we explore were patch size $= \{6 \times 6, 10 \times 10, 15 \times 15\}$.

Shown in Figure 2 are the 100 dictionary atoms learned using all of the 6x6 patches from the reference image *lena*. This consisted of $> 250,000$ patches and was processed in 50 minutes using our algorithm.

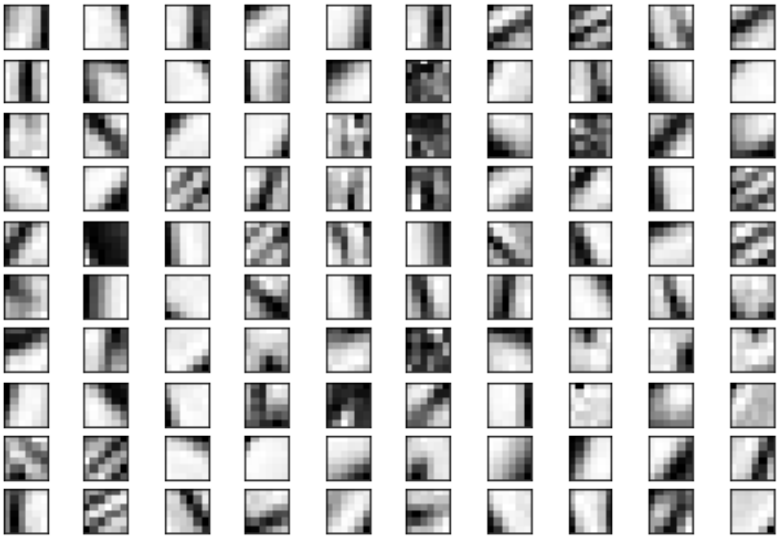

Figure 2: Dictionary elements learned from 6x6 patches extracted from *lena*

Something to note about the dictionary elements is that because this dictionary learning and sparse construction technique is not shift invariant many of the patches look very similar to each other but are slightly shifted.

Additionally we used as an initial benchmark test the reconstruction of different shades of gray using the dictionary. This was extremely helpful in diagnosing issues we had when we were merging our code together and the images were not turning out the way we expected. Once our code had been debugged we were able to demonstrate the reconstruction of a uniform gray image which had been half-toned. These results can be seen in Figure 3.

Figure 3: Reconstruction of a uniform gray image from its half-tone conterpart, PSNR=40.36

Now that we demonstrated our algorithms effectiveness on a simple gray image we explore its performance on the natural images.

Seen in Figure 4 we can see our best reconstruction of the *boat* by reconstructing the noisy patches using our dictionary. This was accomplished using the parameters patches=6x6, training samples=10,000, dictionary atoms = 100, reconstuction atoms=2, number non-zero(nnz) for K-SVD = 1.

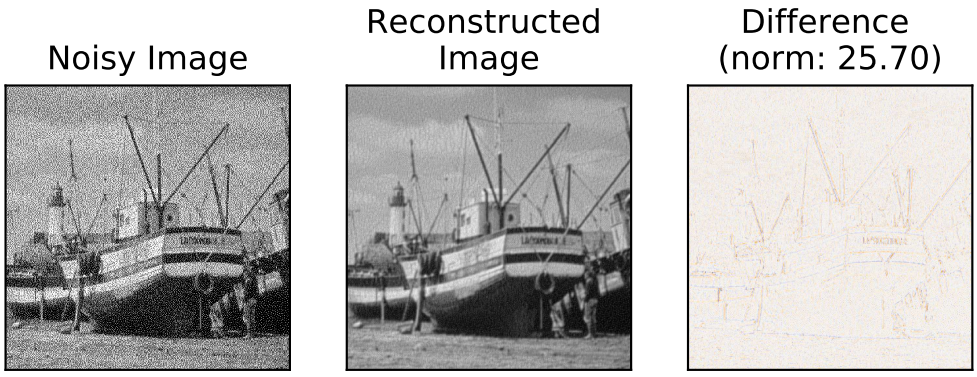

Figure 4: Best Performance of our algorithm on the *boat* image, PSNR=25.99

We can see in the difference image where exactly our algorithm and dictionary are failing to reconstruct the image properly. Many of the edges are difficult to reconstruct using our dictionary and patch methodology. It could be possible to improve this performance if we were either increase our dictionary size or to incorporate more data into our training which is representative of the failure cases we observe. An example of our *lena* reconstruction is included in Appendix C

We explored a variety of hyperparameters and measured the average performance on our test image. The results can be seen in Table 1.

	Reconstruction Atoms		**Patch Size**			**Training Samples**			**Dictionary Atoms**		
Value	1	2	6x6	10x10	15x15	100	1000	10000	10	100	200
PSNR	23.43	25.00	25.81	24.1	23.47	21.47	24.51	24.53	24.33	24.11	24.86

Table 1: Average PSNR for different parameters

In general our algorithm was able to reconstuct the image for *lena* with +3dB PSNR over our test image. This could in part be due to our dictionary being trained on *lena*, but the trend is also present in the results presented in [2]. So it could be that *boat* is in general a more difficult image task.

Our final best results on reconstructing each of the half-tone images and comparisons can be seen in Table 2.

	lena	**boat**
half-tone	6.70	6.88
Random Dictionary	15.04	14.32
Our Dictionary	28.06	25.99
Paper Table	31.4-33.0	28.6-30.2

Table 2: Comparison of our results with controls and the results achieved and mentioned in [2], All values are in PSNR

Our results are not as accurate as what was achieved in the paper[2] or the other benchmark algorithms mentioned. There are a number of reasons why this could be, some of the most likely differences are the size of the training data. We used 250,000 patches vs. 9 million in the paper. They also extracted their training data from 24 images rather than our method which only used a single image's patches for training.

They performed more hyperparameter searches both in terms of patches, dictionary size, and regularization terms.

It is of note, that our algorithm was able to achieve within 3.5dB of other algorithms which are designed for this image processing problem. We also performed much better than a random dictionary and improved the starting image. Achieving these results required us to be able to effectively integrate the functions of our group together and was a learning experience which required intelligent debugging, communication about variable dimensions, using each team members expertise, and understanding a Python library's API. After working together to solve these sticking points, we were able to generate the final results discussed above.

Our results demonstrate that our implementation of the K-SVD algorithm was effective at achieving a dictionary which could be used to effectively to solve the inverse half-tone task.

References

[1] M. Aharon, M. Elad, A. Bruckstein *et al.*, "K-svd: An algorithm for designing overcomplete dictionaries for sparse representation," *IEEE Transactions on signal processing*, vol. 54, no. 11, p. 4311, 2006.

[2] J. Mairal, F. Bach, and J. Ponce, "Task-driven dictionary learning," *IEEE transactions on pattern analysis and machine intelligence*, vol. 34, no. 4, pp. 791–804, 2012.

[3] J. M. Guo and S. Sankarasrinivasan, "Digital halftone database (DHD): A comprehensive analysis on halftone types," in *2018 Asia-Pacific Signal and Information Processing Association Annual Summit and Conference (APSIPA ASC)*. IEEE, pp. 1091–1099. [Online]. Available: https://ieeexplore.ieee.org/document/8659732/

A Appendix

B Benchmark Test Code

Presented here is the code we used for the image inverse half-tone task. The Python library scikit learn was useful for patch extraction and reconstruction, an example of which can be found here. The main focus of our project was the dictionary learning rather than the pursuit algorithms for finding the coefficients so the sklearn implementation of orthogonal matching pursuit was used to save time.

```
"""ML_Final_Project_Halftone_submit.ipynb

Automatically generated by Colaboratory.

"""

import numpy as np
import matplotlib.pyplot as plt

from PIL import Image

from sklearn.feature_extraction.image import extract_patches_2d
from sklearn.feature_extraction.image import reconstruct_from_patches_2d

from time import time

# mounts my google drive in colab allowing me to access shared folders
from google.colab import drive
```

```
drive.mount('/content/drive')

path = "/content/drive/My Drive/Half-Tone Images"

""" Code for loading in all of the images in the folder
import glob
image_list = []
for filename in glob.glob(path+'/Ref/*.tif'):
im=Image.open(filename)
    image_list.append(im)
"""

#Import all of the reference images as greyscale, they should already be that
img_0_ref = Image.open(path + '/Ref/Ref  (163).tif').convert('L')
img_1_ref = Image.open(path + '/Ref/Ref  (23).tif').convert('L')
img_2_ref = Image.open(path + '/Ref/Ref  (95).tif').convert('L')
img_3_ref = Image.open(path + '/Ref/Ref  (165).tif').convert('L')

# import the half-tone images
img_0_ht = Image.open(path + '/Floyd-Steinberg/11 (163).tif').convert('L')
img_1_ht = Image.open(path + '/Floyd-Steinberg/11 (23).tif').convert('L')
img_2_ht = Image.open(path + '/Floyd-Steinberg/11 (95).tif').convert('L')
img_3_ht = Image.open(path + '/Floyd-Steinberg/11 (165).tif').convert('L')

# convert the images to np arrays for manipulation, normalize from 0-1
img_0_ref = np.asarray(img_0_ref)/255
img_1_ref = np.asarray(img_1_ref)/255
img_2_ref = np.asarray(img_2_ref)/255
img_3_ref = np.asarray(img_3_ref)/255

img_0_ht = np.asarray(img_0_ht)/255
img_1_ht = np.asarray(img_1_ht)/255
img_2_ht = np.asarray(img_2_ht)/255
img_3_ht = np.asarray(img_3_ht)/255

# check the images look like what we expect
fig, (ax1, ax2) = plt.subplots(1, 2, figsize=(40,40))
ax1.imshow(img_3_ref, cmap='gray', vmin=0, vmax=1)
ax2.imshow(img_3_ht, cmap='gray', vmin=0, vmax=1)
plt.show()

"""### Toy data

We can split the shaded image into pieces in order to more clearly see how well we are
↪  doing and to speed up the trials when we are initially testing.
"""

size = img_0_ht.shape
height = int(size[0]/7)
width = int(size[1]/3)
```

```
shade_ht = []
# there are 7x3 shades in the image
for i in range(3):
  for j in range(7):
    shade = img_0_ht[height*j:height*(j+1), width*i+1:width*(i+1)]
    shade_ht.append(shade)

shade_ref = []
# there are 7x3 shades in the image
for i in range(3):
  for j in range(7):
    shade = img_0_ref[height*j:height*(j+1), width*i+1:width*(i+1)]
    shade_ref.append(shade)


shade_ht = np.asarray(shade_ht)
shade_ref = np.asarray(shade_ref)

# Check and see how the shade separations look
fig, (ax1, ax2) = plt.subplots(1, 2, figsize=(20,30))
ax1.imshow(shade_ref[12], cmap='gray', vmin=0, vmax=1)
ax2.imshow(shade_ht[12], cmap='gray', vmin=0, vmax=1)
plt.show()

"""## Extract the Patches for training the dictionary

We could also change our images to be more similar in terms of content.
For now I'm extracting all of the patches and using them to learn the dictionary in the
↪  next step.
"""

print('Extracting reference patches...')
t0 = time()

patch_size = (10,10)

data0 = extract_patches_2d(img_0_ht, patch_size)
data1 = extract_patches_2d(img_1_ht, patch_size)
data2 = extract_patches_2d(img_2_ht, patch_size)
data3 = extract_patches_2d(img_3_ht, patch_size)

data0_ref = extract_patches_2d(img_0_ref, patch_size)
data1_ref = extract_patches_2d(img_1_ref, patch_size)
data2_ref = extract_patches_2d(img_2_ref, patch_size)
data3_ref = extract_patches_2d(img_3_ref, patch_size)

data = np.vstack((data3_ref))

# decided to only use the patches in one image to save computation time
data = data3_ref
```

```
#Flatten the patches and normalize them to have mean=0
data = data.reshape(data.shape[0], -1)
intercept = np.mean(data, axis=0)
data -= intercept

print(f'done in {round(time() - t0, 2)}s')
print('Number of patches:', data.shape)

"""##Dictionary Learning

Initial demo using the example from sklearn here:
↪  https://scikit-learn.org/stable/auto_examples/decomposition/plot_image_denoising.html

We will replace these steps here with our K-SVD algorithm
"""

# shows that it works on the toy dataset
#

from sklearn.decomposition import MiniBatchDictionaryLearning

# this was the example used in sklearn, much faster than ours,
# and provides a point of reference for debugging
def learn_dictionary(data, n_components=100, alpha=0.1):
  print('Learning the dictionary via Sklearn MiniBatchingDictLearning...')
  t0 = time()
  dico = MiniBatchDictionaryLearning(n_components=n_components, alpha=alpha,
                                     n_iter=500)
  V = dico.fit(data).components_
  dt = time() - t0
  print(f'done in {dt}')
  plt.figure(figsize=(7, 7))
  for i, comp in enumerate(V[:100]):
      plt.subplot(10, 10, i + 1)
      plt.imshow(comp.reshape(patch_size), cmap=plt.cm.gray_r,
                 interpolation='nearest')
      plt.xticks(())
      plt.yticks(())
  plt.suptitle('Dictionary learned from image patches\n' +
               'Train time %.1fs on %d patches' % (dt, len(data)),
               fontsize=16)
  plt.subplots_adjust(0.08, 0.02, 0.92, 0.85, 0.08, 0.23)

  return dico, V

# here is our dictionary learning implementation
import KSVD
def learn_dictionary_ksvd(data, n_components=100, nnz=1):
  print(f'Learning dictionary via our KSVD algorithm')
  t0 = time()
  Y = data.T
```

```
  D = KSVD.batching_KSVD(Y, batch_size=15000, K=n_components, nnz=nnz)
  V = D.T

  dt = time() - t0
  print(f'done in {dt}')

  for i, comp in enumerate(V[:100]):
    plt.subplot(10, 10, i + 1)
    plt.imshow(comp.reshape(patch_size), cmap=plt.cm.gray_r,
               interpolation='nearest')
    plt.xticks(())
    plt.yticks(())
  plt.suptitle('Dictionary learned from image patches\n' +
               'Train time %.1fs on %d patches' % (dt, len(data)),
               fontsize=16)
  plt.subplots_adjust(0.08, 0.02, 0.92, 0.85, 0.08, 0.23)

  return _, V

# randomizes the data and samples the first
training_samples = 10000
np.random.shuffle(data)
input_data = data[:training_samples]
print('Training data shape:', input_data.shape)

# Learn the dictionary from the training samples
dico, V_sk = learn_dictionary(data, n_components=100, alpha=0.1 )
_, V = learn_dictionary_ksvd(input_data, n_components=100, nnz=1)

def show_with_diff(noisy, image, reference, title):
    """Helper function to display denoising"""
    plt.figure(figsize=(10, 7))
    plt.subplot(1, 3, 1)
    plt.title('Noisy Image')
    plt.imshow(noisy, vmin=0, vmax=1, cmap=plt.cm.gray,
               interpolation='nearest')
    plt.xticks(())
    plt.yticks(())
    plt.subplot(1, 3, 2)
    plt.title('Reconstructed Image')
    plt.imshow(image, vmin=0, vmax=1, cmap=plt.cm.gray,
               interpolation='nearest')
    plt.xticks(())
    plt.yticks(())
    plt.subplot(1, 3, 3)
    print(f'Shape of Reconstructed image is {image.shape}, '
          f'Shape of Reference image is {reference.shape}')
    difference = image - reference

    plt.title('Difference (norm: %.2f)' % np.sqrt(np.sum(difference ** 2)))
    plt.imshow(difference, vmin=-0.5, vmax=0.5, cmap=plt.cm.PuOr,
```

```
                interpolation='nearest')
    plt.xticks(())
    plt.yticks(())
    plt.suptitle(title, size=16)
    plt.subplots_adjust(0.02, 0.02, 0.98, 0.79, 0.02, 0.2)

    psnr = calculate_psnr(image,reference)
    print('PSNR: ', psnr)

import numpy
import math

def calculate_psnr(img1, img2):
  """Calculates the Peak signal-to-noise ratio, units are dB"""
    mse = numpy.mean( (img1 - img2) ** 2 )
    if mse == 0:
       return 100
    PIXEL_MAX = 1.0
    return round(20 * math.log10(PIXEL_MAX / math.sqrt(mse)),2)

from sklearn.linear_model import OrthogonalMatchingPursuit
from sklearn.decomposition import SparseCoder

def reconstruct_image_sparsely(noisy_image, ref_image, V, patch_size, dico=None):
  print('Extracting noisy patches... ')
  t0 = time()
  data = extract_patches_2d(noisy_image, patch_size)
  data = data.reshape(data.shape[0], -1)
  intercept = np.mean(data, axis=0)
  data -= intercept
  print('done in %.2fs.' % (time() - t0))

  transform_algorithms = [
      ('Orthogonal Matching Pursuit\n1 atom', 'omp',
       {'transform_n_nonzero_coefs': 1}),
      ('Orthogonal Matching Pursuit\n2 atoms', 'omp',
       {'transform_n_nonzero_coefs': 2})]

  reconstructions = {}
  for title, transform_algorithm, kwargs in transform_algorithms:
      print(title + '...')
      reconstructions[title] = noisy_image.copy()
      t0 = time()

      # find the sparse coefficients to reconstruct the patches
      coder = SparseCoder(dictionary=V, transform_algorithm='omp',
                          transform_n_nonzero_coefs=kwargs['transform_n_nonzero_coefs'])
      code = coder.transform(data)

      patches = np.dot(code, V)
      patches += intercept
```

```
        patches = patches.reshape(len(data), *patch_size)

        reconstructions[title] = reconstruct_from_patches_2d(
            patches, (noisy_image.shape))
        dt = time() - t0
        print('done in %.2fs.' % dt)
        show_with_diff(noisy_image, reconstructions[title], ref_image,
                       title + ' (time: %.1fs)' % dt)

# reconstruct the half-tone images using dictionary
reconstruct_image_sparsely(shade_ht[10], shade_ref[10], V, patch_size)

reconstruct_image_sparsely(img_3_ht, img_3_ref, V, patch_size)

reconstruct_image_sparsely(img_2_ht, img_2_ref, V, patch_size)

# Shows that our dictionary is better than random or nothing

random_dict = np.random.randn(100*100).reshape(100,100)

for i, comp in enumerate(random_dict[:100]):
  plt.subplot(10, 10, i + 1)
  plt.imshow(comp.reshape(patch_size), cmap=plt.cm.gray_r,
             interpolation='nearest')
  plt.xticks(())
  plt.yticks(())
plt.subplots_adjust(0.08, 0.02, 0.92, 0.85, 0.08, 0.23)

reconstruct_image_sparsely(img_3_ht, img_3_ref, random_dict, patch_size)

print(calculate_psnr(img_3_ref, img_3_ht))
```

C Appendix: Lena reconstucted image

Figure 5: Performance of our algorithm on the *lena* image, PSNR=28.06, Showing half-tone image, reconstructed image, reference image, difference image(reference-reconstruction). Zoom in on computer for comparison