A2

Sean Pavlak

Mar 19 2018

1 Exercise 1

Plot the basis functions of a 16x16 discrete cosine transform (DCT).

I attempted to implement a transform matrix solution for the DCT. There were two clear options to compute DCT as documented in MathWorks documentation files for DCT, either the FFT-based algorithm or a transform matrix.

The optimal option appeared to be matrix transformation, as we were only computing a 16x16 matrix of basis functions. The mathematics behind DCT using matrix transformation is as follows.

$$B_{pq} = \begin{cases} \frac{1}{\sqrt{M}} & \text{if } n \text{ p} = 0 \\ \sqrt{\frac{2}{M}} \cos \frac{\pi(2q+1)p}{2M} & \text{if } n \text{ } 1 \leq \text{p} \leq \text{M - 1} \end{cases} \tag{1}$$

Discrete cosine transform represents an image as a sum of sinusoids of varying magnitudes and frequencies and has the property that most of the visually significant information about the image is concentrated in just a few coefficients of the matrix.

1.1 Code Description

For the exercise I created a function that performs a matrix transformation returning a set of basis functions given an integer. I attempted to post process the matrix in order to separate the basis functions into distinct blocks; however, this was unsuccessful.

```
def dct_basis_function(basis_count):
    basis_array = np.zeros((basis_count, basis_count))

    for p in range(basis_count):
        for q in range(basis_count):
            if (p == 0):
                basis_array[p, q] = np.sqrt(1/basis_count)
            else:
                basis_array[p, q] =
                 ↪  np.sqrt(2/basis_count)*np.cos((np.pi*(2*q+1)*p)/(2*basis_count))

    basis_array_flat = basis_array.flatten()
    basis_array_flat.shape = (basis_array_flat.shape[0], 1)
    dct_basis_function =
     ↪  np.transpose(basis_array_flat)*basis_array_flat

    return dct_basis_function
```

After defining this function all that remained was to call it and it would generate any nxn basis function matrix.

The returned matrix does not have a clear definition of where one basis function begins and the other ends, this presents a minor issue with viewing. I made attempts to post-process the matrix to separate the basis functions, but this was unsuccessful.

1.2 Results

After defining this function all that remained was to call it and it would generate any nxn basis function matrix.

I ran the function to generate a 4x4, 8x8, 16x16, 32x32, and 64x64 matrix of basis functions. My intent for this was to show that I truely have succesffully defined an nxn basis function generator, but also to test the MathWorks statement that FFT-based algorithm's out perform transform matrix at large values of n. (n <16)

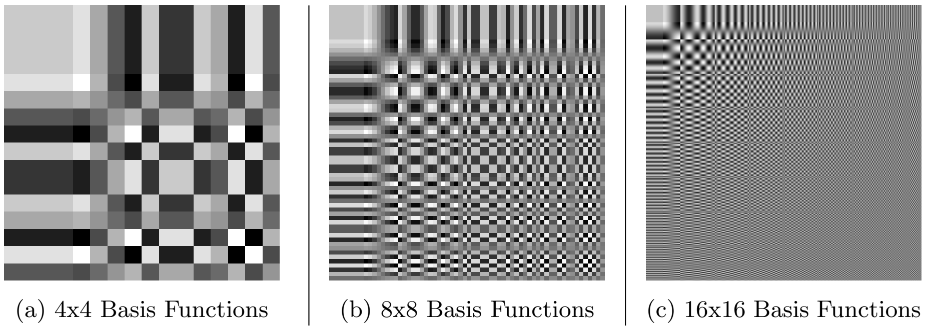
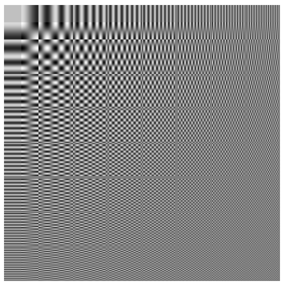

(a) 4x4 Basis Functions (b) 8x8 Basis Functions (c) 16x16 Basis Functions

Figure 1: (a),(b), and (c) are the generated basis functions from the definition above in equation (1) and code block 1. These functions were generated very quickly and efficiently.


(a) 16x16 Basis Functions

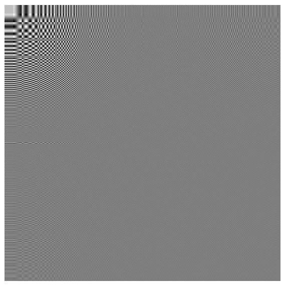

(b) 32x32 Basis Functions

Figure 2: Both of these are also the generated basis functions from the definition above in equation (1) and code block 1. However, these functions took significantly longer to generate giving credence to the statement provided by MathWorks.

2 Exercise 2

Implement low- and high-pass image filters by zeroing different ranges of the DCT coefficients.

As described in Exercize 1, discrete cosine transform represents an image as a sum of sinusoids of varying magnitudes and frequencies and has the property that most of the visually significant information about the image is concentrated in just a few coefficients of the matrix.

So it is possible to perform DCT on an image to generate a set of basis functions, coefficients, and use them to recreate the original image. Empirically I was able to determine that the coefficients of greatest significance are located near (0,0) of the matrix and the coefficients of least significance are located near (n,n).

In order to implement a low pass filter I focused on the most significant coefficients, and for the high pass I focused mainly on the least significant coefficients.

I was curious as to how both of these filters evolved over time and over a larger set of basis functions so I set out to generate GIFs to show just how they change with respect to time.

2.1 Code Description

For my code I defined two functions to extend the utility of jupyter notebooks to handle GIFs well.

```python
def show_gif(filename):
    file = open(filename , "rb")
    image = file.read()
    progress= Img(
        value=image,
        format='gif',
        width=400,
        height=400)
    display.display(progress)

    return progress
```

```python
def create_gif(filename, path):
    with imageio.get_writer(filename, mode='I') as writer:
        glob_path = path + '/*.png'
        for filename in sorted(glob(glob_path)):
            image = imageio.imread(filename)
            writer.append_data(image)
```

Once I had that functionality, I needed to generate each frame as a different set of basis functions. In order to calculate the DCT I implemented Scipy's fftpack which includes functions for DCT.

```python
    image = Image.open('E2_image.png')
    image = image.resize((256,256), 1)
    image = image.convert('L')
    pixels = np.array(image.getdata(),
    ↪   dtype=np.float).reshape((dctSize, dctSize))
    dct = fftpack.dct(fftpack.dct(pixels.T, norm='ortho').T,
    ↪   norm='ortho')
```

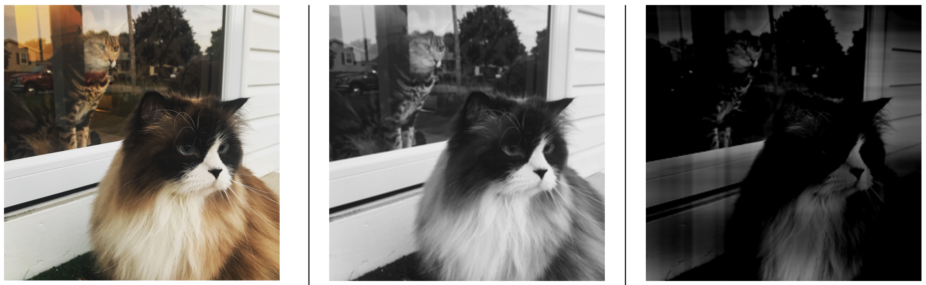
Now that I have the DCT of my image I can zero out parts and then perform inverse DCT to generate the image. That would suffice for a single filtered image, but in order to generate the entire image I have to zero out less and less over a loop.

```python
for i in range(0, dctSize):
    dct2 = dct.copy()

    dct2[i:,:] = 0
    dct2[:,i:] = 0

    idct = fftpack.idct(fftpack.idct(dct2.T, norm='ortho').T, norm='ortho')

    idct = idct.clip(0, 255)
    idct = idct.astype('uint8')
    img = Image.fromarray(idct)

    img.save()
```

This successfully generated 255 frames for the high and low pass filters. Then I just had to call my GIF functions and I was able to show how more, or less, coefficients effects the resulting image.

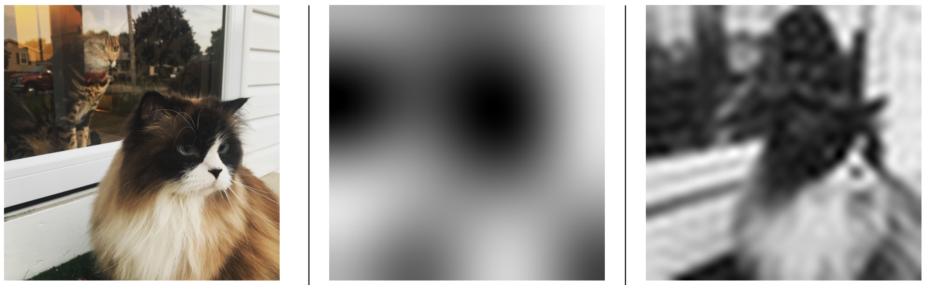
2.2 Results

Running this code yielded these results. I am unable to demonstrate the GIF in LaTeX however they are available in the jupyter notebook.



(a) Original Image (b) More Components (c) Less Components

Figure 3: This figure demonstrated how highpass filtering changes with respect to the components.



(a) Original Image (b) Few Components (c) More Components

Figure 4: This figure demonstrated how lowpass filtering changes with respect to the components.

3 Exercise 3

Show that convolving a 2D convolution kernel with an image is (approximately) equivalent to multiplying the transforms of the kernel and the image and then applying the inverse transform. You will need to center and pad the kernel so that the signals are the same size.

In this example I used the Scipy's signal library in order to demonstrate 2d convolution. I simply convolved an image and then multiplied the transforms of the kernel and the image and applied the inverse transform.

Once each are performed am able to compare them to show that both methods are equivalent.

$$\frac{1}{256}\begin{bmatrix}1 & 4 & 6 & 4 & 1\\ 4 & 16 & 24 & 16 & 4\\ 6 & 24 & 36 & 24 & 6\\ 4 & 16 & 24 & 16 & 4\\ 1 & 4 & 6 & 4 & 1\end{bmatrix} \tag{2}$$

Using the convolution matrix and signal we are able to generate a convolved image. As seen below in figure 5.

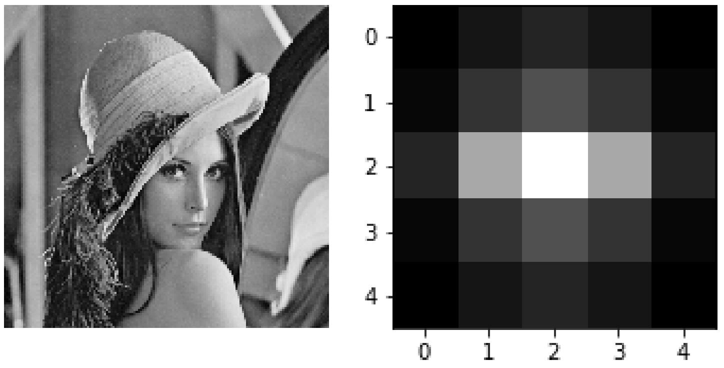


Figure 5: Uploaded image along with graphical representation of convolution matrix

In my program then performed a similar task, but by use of matrix multiplication. This also generated a convolved image, I then determined the mean square error between them to produce a quantifiable difference.

3.1 Code Description

In my program I loaded the image along with a convolution matrix and used signal to produce a convolved image.

```
original_image=mpimg.imread('E3_image.png')
convolution_matrix = 1.0/256 * np.array([[1, 4, 6, 4, 1],
                                         [2, 8, 12, 8, 2],
                                         [6, 24, 36, 24, 6],
                                         [2, 8, 12, 8, 2],
                                         [1, 4, 6, 4, 1]])

convolved_image = signal.convolve2d(original_image,
 ↪  convolution_matrix, mode='full')
convolved_image_fft2 = np.fft.fft2(convolved_image)
```

I then took found the pad of the image and convolution matrix and performed matrix multiplication.

```
    pad_original_image = np.pad(original_image, ((0, 4), (0,4)),
    ↪  'constant');
    pad_convolution_matrix = np.pad(convolution_matrix, ((0,
    ↪  511), (0, 511)), 'constant');

    pad_original_image_fft2 = np.fft.fft2(pad_original_image)
    pad_convolution_matrix_fft2 =
    ↪  np.fft.fft2(pad_convolution_matrix)

    multiplied_image = np.multiply(pad_original_image_fft2,
    ↪  pad_convolution_matrix_fft2)
    inversed_image = np.fft.ifft2(multiplied_image);
```

```
    mse1=(np.abs(convolved_image_fft2-multiplied_image) **
    ↪  2).mean()
    mse2=(np.abs(convolved_image-inversed_image) ** 2 ).mean()
```

3.2 Results

Performing both of these were able to generate strikingly similar results. The mean square error between the convolved_image_fft2 and multiplied_image was 7.91481301239e-25. Which is impressively small, and the difference between convolved_image and inversed_image was even smaller, 8.50963543153e-31.

4 Exercise 4

Select an image dataset, e.g. the MNIST handwritten digits, and compute the principal components. Show that the individual images can be approximated by the sum of the first k principal components.

For this exercise I imported the MNIST dataset in order to perform principle component analysis (PCA) on it. My intent was that once I performed PCA I would be able to determine the most important components and then combine them to generate an image not too different from the original.

PCA is a statistical procedure that uses an orthogonal transformation to convert a set of observations of possibly correlated variables into a set of values of linearly uncorrelated variables called principal components.

In this exercise I utilized sklearn's decomposition library which includes PCA functionality.

4.1 Code Description

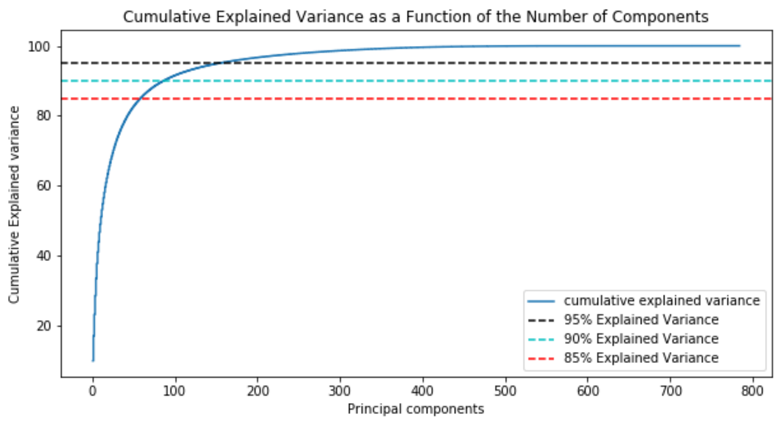
In order to generate an image from the sum of principal components, I fist needed to get the MNIST dataset and fit it with PCA.

```
mnist = fetch_mldata('MNIST original')
pca = PCA()
pca.fit(mnist.data)
```

Then Using cumulative explained variance it is possible to determine the number of components required to attain a certain accuracy in the image.

This was done by sorting the explained variance in the fit and calculating the sum over the entire set. Once performed this yielded a function relating the number of components to the variance.

```
var_exp = [(i/sum(pca.explained_variance_))*100 for i in
↪  sorted(pca.explained_variance_, reverse=True)]
cum_var_exp = np.cumsum(var_exp)
```



This graph shows us that it is possible to isolate a specific percentage and determine the components required to yield that image.

```
[784, np.argmax(cum_var_exp > 99) + 1, np.argmax(cum_var_exp
↪  > 95) + 1, np.argmax(cum_var_exp > 90) + 1,
↪  np.argmax(cum_var_exp >= 85) + 1]
```

$$[784, 331, 154, 87, 59] \tag{3}$$

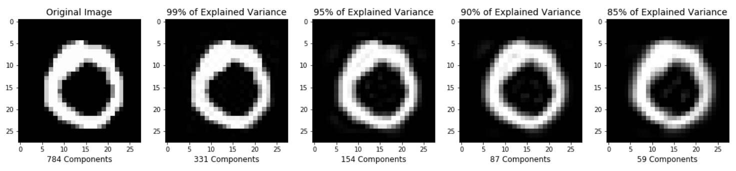
Yielding us with a set of components corresponding to specific percentages. As shown in equation 3.

In order to utilize this I defined a function that takes a percentage and an image set and performs a PCA fit, isolates the required components, and performs a PCA inverse transform to generate the image back.

```
def explained_variance(percentage, images):
pca = PCA(percentage)
pca.fit(images)
components = pca.transform(images)
approx_original = pca.inverse_transform(components)
return approx_original
```

4.2 Results

My program was able to successfully generate a set of images corresponding to how similar that are to the original and output how many components were required to do so.


Original Image
99% of Explained Variance
95% of Explained Variance
90% of Explained Variance
85% of Explained Variance
784 Components
331 Components
154 Components
87 Components
59 Components