The Discrete Cosine Transform and JPEG

Alec Koppel, Mark Eisen, Alejandro Ribeiro

March 10, 2021

For image processing applications, it is useful to consider the Discrete Cosine Transform (2D DCT) instead of the 2D DFT due to its superior empirical performance for signal compression and reconstruction tasks. We first introduce the two-dimensional discrete cosine $C_{kl}(m,n)$ of frequencies k,l defined as

$$C_{kl,MN}(m,n) \; = \; \cos\left[\frac{k\pi}{2M}(2m+1)\right]\cos\left[\frac{l\pi}{2N}(2n+1)\right]. \tag{1}$$

Then the two-dimensional DCT of a signal $\mathbf{x}$ is given by substituting $\mathbf{C}_{kl,MN}$ into the expression for the two-dimensional DFT, which, after introducing normalization constants as shown in lecture, yields

$$\begin{aligned} X_C(k,l) &:= \frac{2}{\sqrt{MN}}\sum_{m=0}^{M-1}\sum_{n=0}^{N-1} x(m,n)c_1c_2\cos\left[\frac{k\pi}{2M}(2m+1)\right]\cos\left[\frac{l\pi}{2N}(2n+1)\right] \\ &= \frac{2}{\sqrt{MN}}\langle x, c_1c_2C_{kl,MN}\rangle. \end{aligned} \tag{2}$$

where $c_1 = \frac{1}{\sqrt{2}}$ for $k=0$ and $c_1 = 1$ for $k = 1,\ldots,M-1$ and $c_2 = \frac{1}{\sqrt{2}}$ for $l = 0$ and $c_2 = 1$ for $l = 1,\ldots,N-1$. Note that again this may be computed as an inner product in two dimensions, just like the 2D DFT. Crucial to the theory of image reconstruction and compression is the 2D inverse Discrete Cosine Transform (2D iDCT), which is the signal $\tilde{x}_C$ defined as

$$\tilde{x}_C(m,n) := \frac{2}{\sqrt{MN}}\sum_{k=0}^{M-1}\sum_{l=0}^{N-1} X_C(k,l)c_1c_2\cos\left[\frac{m\pi(2k+1)}{2M}\right]\cos\left[\frac{n\pi(2l+1)}{2N}\right] \tag{3}$$

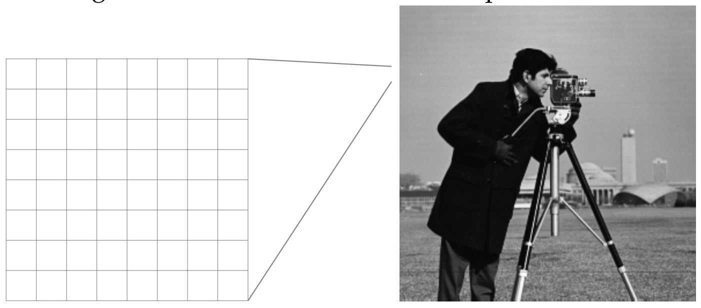
where $c_1 = \frac{1}{\sqrt{2}}$ for $m = 0$ and $c_1 = 1$ for $m = 1,\ldots,M-1$ and $c_2 = \frac{1}{\sqrt{2}}$ for $n = 0$ and $c_2 = 1$ for $n = 1,\ldots,N-1$. Analogous to the 2D DFT, we note that the sum in (3) allows us to represent an arbitrary two-dimensional signal as a sum of cosines, and hence we may ask how many cosines are necessary to represent the signal well in terms of reconstruction error. We explore this question in the first part of this lab. Henceforth you may assume that $M = N$, so that signals are of dimension N^2.

1 Image Compression

1.1 DCT in Two Dimensions. Write down a Python class which takes as input a two-dimensional signal of size N^2 and computes its two-dimensional DCT defined in (2).

1.2 Image Compression. When the signal dimension N^2 is very large, it is difficult to represent it well across its entire domain using the same DFT or DCT coefficients. This is because the computation of the inverse in (3) uses the same DFT or DCT coefficients across the entire domain. In Lab 3 Part 2, we designed an audio compression scheme by partitioning the signal and computing the DFT of each piece so that our DFT coefficients only needed to locally represent the signal over a small domain. We will implement a two-dimensional analogue here.

Hence, write a Python class that takes in a signal (image) of size N^2 and partitions it into patches of size 8×8, and for each patch stores the K largest DFT coefficients and their associated frequencies. We still want the patch to be 8×8 so set the other $64 - K$ coefficients to zero. Your partitioning scheme should resemble the depiction below.



Write another Python class that executes this procedure for the two dimensional DCT. Try both of these functions out on the provided sample image A for $K = 4, 8, 16, 32$. Make sure to keep track of each patch's frequencies associated with the dominant DCT coefficients. (*Hint:* See Python function `matplotlib.pyplot.imshow`.)

1.3 Quantization. A rudimentary version of the JPEG compression scheme for images includes partitioning the image into patches, performing the two-dimensional DCT on each patch, and then rounding (or quantizing) the associated DCT coefficients. We describe this procedure below:

1. Extract a 8×8 block of the image x_{ij} where (i, j) denotes the location of the first pixel of your batch. (*Hint:* It is useful to keep track of these indices.)

2. Perform the DCT of your 8×8 block x_{ij}. Store the DCT in X_{ij}, which is another block of 8×8, $X_{ij}(k,l)$, $k, l \in \{1, \ldots, 8\}$ (or any other consecutive set of 8 integers you decide to use as frequencies).

3. Then quantize the DCT coefficients:

$$\hat{X}_{ij}(k,l) = \text{round}\left[\frac{X_{ij}(k,l)}{Q(k,l)}\right] \tag{4}$$

$Q(k,l)$ is a quantization coefficient used to control how much the (k,l)th frequency is quantized. Since human vision is not sensitive to these "rounding" errors, this is where the *compression* takes place. That is, a smaller set of pixel values requires less bits to represent in a computer.

The standard JPEG quantization matrix that you should apply to each patch is based upon the way your eye observes luminance, and is given as

$$Q_L = \begin{pmatrix} 16 & 11 & 10 & 16 & 24 & 40 & 51 & 61 \\ 12 & 12 & 14 & 19 & 26 & 58 & 60 & 55 \\ 14 & 13 & 16 & 24 & 40 & 57 & 69 & 56 \\ 14 & 17 & 22 & 29 & 51 & 87 & 80 & 62 \\ 18 & 22 & 37 & 56 & 68 & 109 & 103 & 77 \\ 24 & 36 & 55 & 64 & 81 & 104 & 113 & 92 \\ 49 & 64 & 78 & 87 & 103 & 121 & 120 & 101 \\ 72 & 92 & 95 & 98 & 112 & 100 & 103 & 99 \end{pmatrix} \tag{5}$$

Write a function that executes the above procedure. If your code is running too slowly, try using Python's built-in functions.

1.4 Image Reconstruction. Write down a Python class that takes in the compression scheme in question 1.2, computes the iDCT of each patch, and then stitches these reconstructed patches together to form the global reconstructed signal. Write another Python class that executes this procedure for the the quantized DCT coefficients from question 1.3.

Run these classes with the results of questions 1.2 and 1.3 as your inputs. That is, you should have one class that takes the patch-wise iDCT of your compressed image, and one class that ”un-quantizes” your quantized DCT coefficients, then takes the patch-wise iDCT of the result. We define the reconstruction error, ρ_K as follows,

$$\rho_K = ||\mathbf{x} - \hat{\mathbf{x}}_K|| \tag{6}$$

where $\hat{\mathbf{x}}_K$ is a reconstructed signal with K coefficients. Plot the reconstruction error ρ_K versus K for your code from questions 1.2. What do you observe? Are you able to discern what is in the original image?

Play around with the quantization matrix. Do higher or lower values of $Q(k,l)$ yield better reconstruction performance? How much can you alter the entries $Q(k,l)$ and still obtain a compression for which the original image is discernible to your eye?