Image compression using Discrete Cosine Transform

13.04.2020

—

Hansin Ahuja
2018csb1094

Overview

One of the transformations used most frequently for image compression is the discrete cosine transform (DCT).

It is obtained by using the following equations (symbols carry their standard meanings):

$$r(x, y, u, v) = s(x, y, u, v)$$

$$= \alpha(u)\alpha(v) \cos\left[\frac{(2x+1)u\pi}{2n}\right] \cos\left[\frac{(2y+1)v\pi}{2n}\right]$$

where

$$\alpha(u) = \begin{cases} \sqrt{\dfrac{1}{n}} & \text{for } u = 0 \\ \sqrt{\dfrac{2}{n}} & \text{for } u = 1, 2, \dots, n-1 \end{cases}$$

Goals

1. To understand JPEG image compression by implementing the DCT transform.
2. To appreciate the data saved during this compression, and the detail with which we can decompress to get the original image back, keeping in mind that this is a form of lossy image compression.

Approach

Compression:

1) Divide the image into subimages of size n x n. Apply zero padding if required.
2) Apply level shifting by subtracting 128 from each pixel of the subimage.
3) Apply DCT on each subimage.
4) Apply element wise division of the pixels by a standard normalization matrix.
5) Round the pixel values.

Decompression:

1) Divide the image into subimages of size n x n.
2) Apply element wise multiplication of the pixels with the aforementioned normalization matrix.
3) Apply inverse DCT on the subimages.
4) Round off the pixel values.
5) Apply level shifting by adding 128 to each pixel value.

Dataset:

Kodak Lossless True Color Image Suite http://r0k.us/graphics/kodak/index.html

We use the grayscale versions of these images. A couple of images in the dataset:

Observations

Let's perform compression on one input image first (Fig 1).

Dimensions = 512 x 768

Memory = 512 x 768 x 8 = 3,145,728 bits

Subimage size = 8 x 8

Normalization matrix: Luminance quantization matrix for JPEG images (https://www.researchgate.net/figure/Luminance-quantization-matrix-for-JPEG_fig3_258382989)

Let's take the second subimage of the second row (Fig 2). The corresponding matrix is:

144	140	141	148	137	138	144	136
138	145	138	140	148	140	141	136
142	142	142	141	144	151	148	151
121	138	140	142	142	148	147	142
106	120	140	133	134	136	138	125
114	111	128	136	133	133	121	119
115	108	120	128	130	141	129	118
156	167	108	115	114	119	129	140

Applying the compression algorithm detailed above, we get:

3	-1	-1	0	0	0	0	0
4	0	-1	0	0	0	0	0
0	2	2	0	0	0	0	0
-2	0	-2	0	0	0	0	0
0	0	1	0	0	0	0	0
0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0

We can see that the relevant information is concentrated in the upper left-hand corner of the resulting matrix, indicating that most of the information is packed in the lower frequency coefficients of the DCT. The number of zero values in the entire compressed image was found to be 356,473.

So all the relevant information was packed in $(512 \cdot 768) - 356,473 = 36,743\ pixels$, and

In terms of number of bits, $memory\ used = 36,743 \cdot 8 = 293,944\ bits$

Giving us a compression ratio $C = 3,145,728 \div 293,944 = 10.7$

And relative data redundancy $R = 1 - (1 \div C) = 0.90$

Note: Counting the number of zeros underestimates the data required, as these zeros may be present somewhere in the upper left hand corner, but it does give us an idea of the matrix being sparse. A more accurate way would be to encode the information using the zigzag algorithm and then calculating the number of bits used.

Fig. 1: Input image

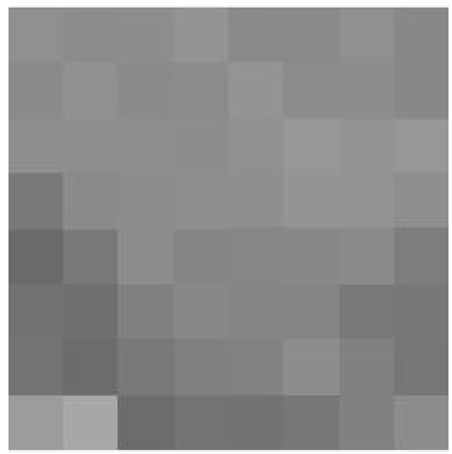

Fig. 2: One subimage of the input image

Let's perform the decompression algorithm, the results of which have been shown in Fig. 3. We can see that the image was restored with very little loss of information. The RMSE between the two images is 3.96.

Let's say that we decompress the image using only the first value of each subimage. The results of this decompression is shown in Fig. 4. The RMSE between Fig. 2 and Fig. 4 is equal to 12.76.

We can conclude from this observation that the larger aspects of the image, such as spatial relation between different objects, are stored in the lower frequency coefficients The finer details, like precise edge locations, which lend the image it's fine quality are stored in the larger coefficients. The largest coefficients store almost no information.

Fig 3: Decompression result

Fig. 4: Decompression using 1x1 portion of subimage

Some results:

Changing the portion of subimage used:

The outputs of these experiments in original quality can be found here: https://drive.google.com/open?id=1xBKIJmXxmTA0hMODOnjB7jy4-nm8w5Uf

Image	**Portion of subimage used**	**RMSE**	**Bits used**	**Compression ratio**	**Relative redundancy**
Input image: Fig. 5	-	-	512 x 768 x 8	-	-
FIg. 6	1 x 1	29.29	6109 x 8	64.37	0.98
Fig. 7	2 x 2	21.71	22291 x 8	17.64	0.94
Fig. 8	3 x 3	16.58	45308 x 8	8.67	0.88
Fig. 9	4 x 4	12.65	68405 x 8	5.75	0.82
Fig. 10	5 x 5	9.58	83494 x 8	4.71	0.78
Fig. 11	6 x 6	8.00	89581 x 8	4.39	0.77
Fig. 12	7 x 7	7.56	90974 x 8	4.32	0.76
Fig. 13	8 x 8	7.49	91320 x 8	4.31	0.76


Fig. 5


Fig. 6


Fig. 7


Fig. 8


Fig. 9


Fig. 10


Fig. 11


Fig. 12


Fig. 13

RMSE table for entire dataset:

“n x n” indicates the portion of the subimage used.

	1x1	2x2	3x3	4x4	5x5	6x6	7x7	8x8
Image ↓								
kodim01.png	25.12	20.80	16.85	13.45	10.43	8.72	7.91	7.76
kodim02.png	11.32	9.06	7.63	6.42	5.47	4.90	4.68	4.65
kodim03.png	12.76	8.96	7.18	5.80	4.77	4.19	3.99	3.96
kodim04.png	14.03	9.86	7.59	5.89	5.00	4.66	4.56	4.55
kodim05.png	29.29	21.71	16.58	12.65	9.58	8.00	7.56	7.49
kodim06.png	21.24	16.64	13.83	11.35	9.07	7.46	6.78	6.55
kodim07.png	18.31	12.19	8.19	5.99	4.78	4.29	4.16	4.15
kodim08.png	36.28	27.51	21.94	17.51	13.48	10.52	8.81	7.94
kodim09.png	17.15	12.33	8.91	6.59	5.19	4.47	4.22	4.16
kodim10.png	16.07	11.72	8.81	6.79	5.32	4.61	4.37	4.31

kodim11.png	19.16	14.49	11.72	9.41	7.56	6.46	6.06	5.98
kodim12.png	16.03	10.75	8.03	6.32	5.08	4.50	4.27	4.14
kodim13.png	29.62	24.35	20.27	16.60	13.28	11.24	10.37	10.12
kodim14.png	20.68	15.36	12.21	9.69	7.83	6.83	6.52	6.47
kodim15.png	18.71	11.47	9.41	7.33	6.30	5.30	5.00	4.66
kodim16.png	13.14	10.71	9.29	7.88	6.58	5.58	5.12	5.02
kodim17.png	16.32	11.06	8.49	6.72	5.61	5.04	4.81	4.77
kodim18.png	20.81	16.43	13.19	10.54	8.55	7.48	7.11	6.93
kodim19.png	21.41	17.02	13.54	10.41	8.34	6.74	5.89	5.60
kodim20.png	18.09	12.90	10.04	7.76	6.11	5.13	4.76	4.70
kodim21.png	20.97	15.96	12.74	10.06	8.07	6.82	6.39	6.24
kodim22.png	15.72	12.54	10.18	8.05	6.68	5.92	5.66	5.57
kodim23.png	13.01	9.37	7.01	5.17	4.02	3.46	3.32	3.31
kodim24.png	22.39	17.91	14.82	11.94	9.49	8.01	7.37	7.11

Conclusions

1) A lot of memory can be saved using image compression techniques.
2) In the case of DCT, that comes at the cost of loss of some information.
3) Most of the larger aspects of the image are stored in the smaller frequency coefficients.
4) The finer details are stored in the slightly larger frequency coefficients.
5) Little to no information is stored in the largest frequency coefficients.
6) Increasing the block size of the image, theoretically would lead to loss of more information but memory used would be lower.
7) Decreasing the block size of the image, theoretically would lead to loss of lesser information but memory used would be greater.

Appendix

The codes for the assignment:

1) **Function compress:**

   Arguments:

   img: image to be compressed

   n: block size

   Q: normalization matrix

   Returns:

   img_comp: the compressed image

```
function img_comp = compress(img, n, Q)

    padX = mod(size(img, 1), n);
    padY = mod(size(img, 2), n);
    img = padarray(img, [padX, padY], 'post');
    img_comp = zeros(size(img));
    M = size(img, 1);
    N = size(img, 2);
    numX = M/n;
    numY = N/n;
    img = double(img);

    for i = 1:numX
        for j = 1:numY
            subimg = img(i*n - n + 1: i*n, j*n - n + 1: j*n);
            subimg = subimg - 128;
            subimg_comp = zeros(size(subimg));
            for u=0:n-1
                for v=0:n-1
                    for x=0:n-1
                        for y=0:n-1
                            pix = subimg(x+1, y+1);
                            c1 = cos(((2*x+1)*u*pi)/(2*n));
                            c2 = cos(((2*y+1)*v*pi)/(2*n));
                            subimg_comp(u+1, v+1) = subimg_comp(u+1, v+1) +
(pix*c1*c2);
```

```
                    end
                end
                if(u==0), subimg_comp(u+1, v+1) = subimg_comp(u+1, v+1) *
sqrt(1/n);
                else, subimg_comp(u+1, v+1) = subimg_comp(u+1, v+1) * sqrt(2/n);
end
                if(v==0), subimg_comp(u+1, v+1) = subimg_comp(u+1, v+1) *
sqrt(1/n);
                else, subimg_comp(u+1, v+1) = subimg_comp(u+1, v+1) * sqrt(2/n);
end
            end
        end
        subimg_comp = subimg_comp ./ Q;
        subimg_comp = round(subimg_comp);
        img_comp(i*n - n + 1: i*n, j*n - n + 1: j*n) = subimg_comp;
    end
end

end
```

2) Function decompress:

Arguments:

img_comp: image to be decompressed

n: block size

m: Portion of the block to be used (range = 1:n)

Q: normalization matrix

Returns:

img_dec: the decompressed image

```
function img_dec = decompress(img_comp, n, m, Q)

    img_dec = zeros(size(img_comp));
    M = size(img_comp, 1);
    N = size(img_comp, 2);
    numX = M/n;
    numY = N/n;
    for i = 1:numX
        for j = 1:numY
            subimg_comp = img_comp(i*n - n + 1: i*n, j*n - n + 1: j*n);
            subimg_comp = subimg_comp .* Q;
```

```
            subimg_dec = zeros(size(subimg_comp));
            for x=0:n-1
                for y=0:n-1
                    for u=0:m-1
                        for v=0:m-1
                            pix = subimg_comp(u+1, v+1);
                            c1 = cos(((2*x+1)*u*pi)/(2*n));
                            c2 = cos(((2*y+1)*v*pi)/(2*n));
                            val = pix*c1*c2;
                            if(u==0), val = val * sqrt(1/2); end
                            if(v==0), val = val * sqrt(1/2); end
                            subimg_dec(x+1, y+1) = subimg_dec(x+1, y+1) + val;
                        end
                    end
                    subimg_dec(x+1, y+1) = subimg_dec(x+1, y+1)*(2/n);
                end
            end
            subimg_dec = round(subimg_dec);
            subimg_dec = subimg_dec + 128;
            img_dec(i*n - n + 1: i*n, j*n - n + 1: j*n) = subimg_dec;
        end
    end
end
```

3) Function rmse:

Arguments:

A: image 1

B: image 2

Returns:

err: RMSE between the two images

```
function err = rmse(A, B)
    minx = min(size(A, 1), size(B, 1)); A=A(1:minx, :); B=B(1:minx, :);
    miny = min(size(A, 2), size(B, 2)); A=A(:, 1:miny); B=B(:, 1:miny);
    A = double(A);
    B = double(B);
    err = sqrt(mean((A(:)-B(:)).^2));
end
```

4) Function main:

Arguments:

img_path: path of image

Returns:

Performs compression using block size = 8, and the standard luminance matrix, and displays the original image and the decompressed images using only the top-left 1 x 1, 4 x 4 and 8 x 8 portion of the subimages, along with their RMSEs.

It returns err: RMSE between the original image and 8 x 8 decompressed Image

```
function err = main(img_path)
    img = rgb2gray(imread(img_path));
    Q = [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      35    55    64    81    104   113   92;
    49      64    78    87    103   121   120   101;
    72      92    95    98    112   100   103   99];
    n = 8;
    img_comp = compress(img, n, Q);
    imgs = [];
    for m=1:n
        imgs(:, :, m) = decompress(img_comp, n, m, Q);
    end
    figure;
    subplot(2,2,1); imshow(uint8(img)); title(sprintf('Original image'));
    subplot(2,2,2); imshow(uint8(imgs(:, :, 1))); title(sprintf('1x1
decompression\nRMSE = %.2f', rmse(img, imgs(:, :, 1))));
    subplot(2,2,3); imshow(uint8(imgs(:, :, 4))); title(sprintf('4x4
decompression\nRMSE = %.2f', rmse(img, imgs(:, :, 2))));
    subplot(2,2,4); imshow(uint8(imgs(:, :, 8))); title(sprintf('8x8
decompression\nRMSE = %.2f', rmse(img, imgs(:, :, 3))));
    err = rmse(img, imgs(:, :, 8));
end
```