Image coding and compression

Dr. Su Su Maung
CEIT Department
Yangon Technological University

Outline of Lecture9

- Introduction
- Lossless compression
- Lossy Compression

Introduction

- There are two different classes of compression methods:
  - lossless compression: where all the information is retained, and
  - lossy compression :where some information is lost.
- Lossless compression is used for images of legal, scientific or political significance, where loss of data, even of apparent insignificance, could have considerable consequences.
- However, lossless compression is used as part of many standard image formats, it can not have high compression ratios.

Lossless and lossy Compression

- lossless compression
  - Human coding
  - Run length encoding
- lossy compression
  - The JPEG algorithm

Human coding

- Rather than using a fixed length code (8 bits) to represent the grey values in an image, Human coding uses a variable length code, with smaller length codes corresponding to more probable grey values.

Human coding (cont.)

- Suppose we have a 2-bit greyscale image with only four grey levels: 0, 1, 2, 3, with the probabilities 0.2, 0.4, 0.3 and 0.1 respectively.

Grey value	Probability	Fixed code	Variable code
0	0.2	00	000
1	0.4	01	1
2	0.3	10	01
3	0.1	11	001

- The average number of bits per pixel can be easily calculated as the expected value.
- Notice that the longest codewords are associated with the lowest probabilities. This average is indeed smaller than 2.

$$(0.2 \times 3) + (0.4 \times 1) + (0.3 \times 2) + (0.1 \times 3) = 1.9$$

Human coding (cont.)

- This can be made more precise by the notion of entropy, which is a measure of the amount of information.
- Specifically, the entropy H of an image is the theoretical minimum number of bits per pixel required to encode the image with no loss of information.

$$H = -\sum_{i=0}^{L-1} p_i \log_2(p_i)$$

$$H = -\big(0.2\log_2(0.2) + 0.4\log_2(0.4) + 0.3\log_2(0.3) + 0.1\log_2(0.1)\big) = \boxed{1.8464.}$$

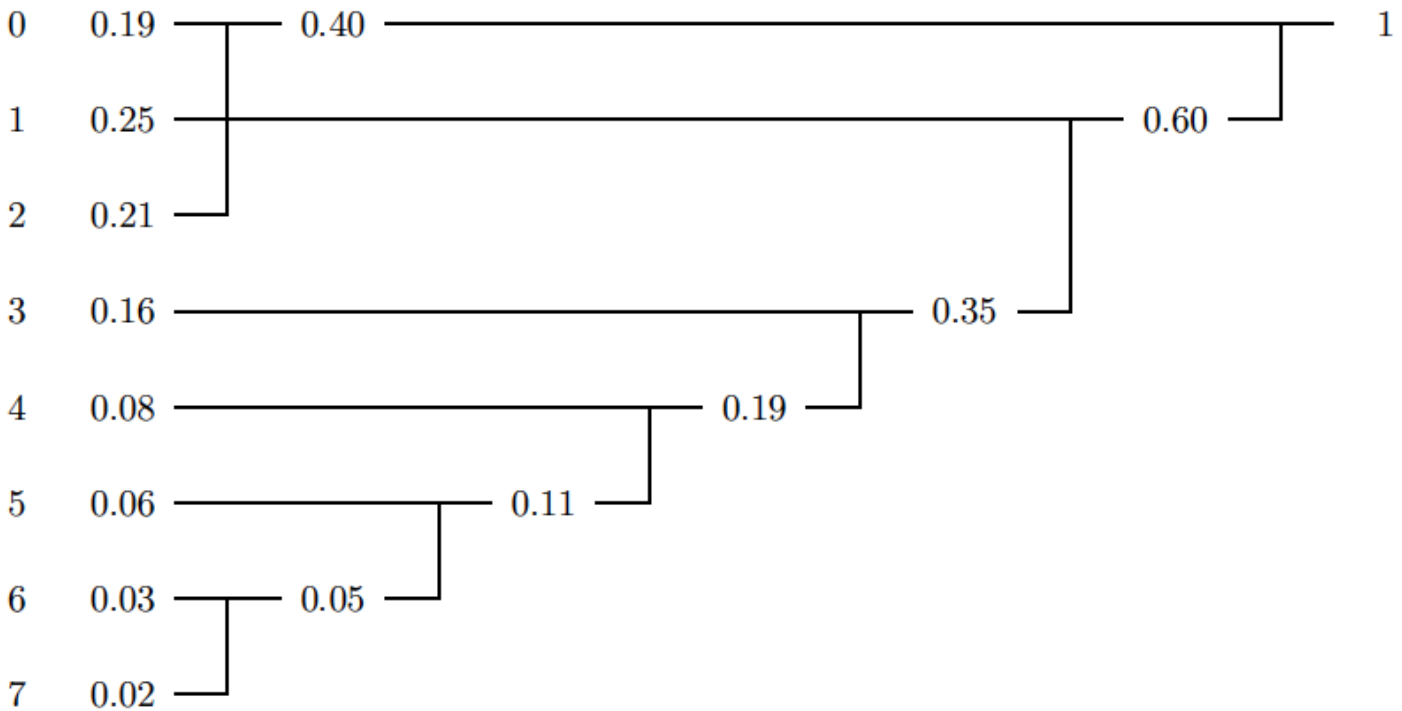
- This means that no matter what coding scheme is used, it will never use less than 1.8464 bits per pixel.
- On this basis, the Human coding scheme given above, giving an average number of bits per pixel much closer to this theoretical minimum than 2, provides a very good result.

Human coding (cont.)

- To obtain the Human code for a given image we proceed as follows:
  - Determine the probabilities of each grey value in the image.
  - Form a binary tree by adding probabilities two at a time, always taking the two lowest available values.
  - Now assign 0 and 1 arbitrarily to each branch of the tree from its apex.
  - Read the codes from the top down.
- To see how this works, consider the example of a 3-bit greyscale image (so the grey values are 0-7) with the following probabilities:

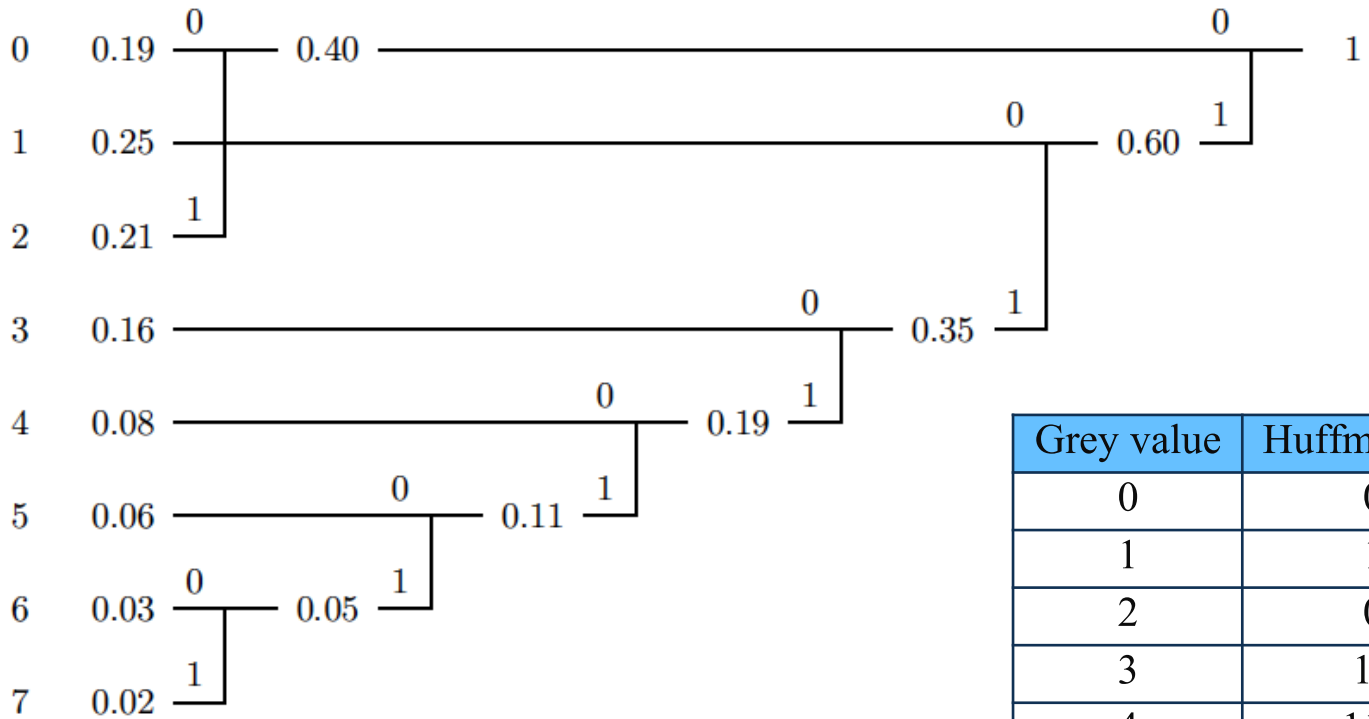
Grey value	0	1	2	3	4	5	6	7
Probability	0.19	0.25	0.21	0.16	0.08	0.06	0.03	0.02

Human coding (cont.)



The Huffman code tree

Human coding (cont.)



Assigning 0's and 1's to the branches

Grey value	Huffman code
0	00
1	10
2	01
3	110
4	1110
5	11110
6	111110
7	111111

Human coding (cont.)

- we can evaluate the average number of bits per pixel as an expected value:

$$(0.19 \times 2) + (0.25 \times 2) + (0.21 \times 2) + (0.16 \times 3) + (0.08 \times 4) + (0.06 \times 5) + (0.03 \times 6) + (0.02 \times) = 2.7$$

- which is a significant improvement over 3 bits per pixel, and very close to the theoretical minimum of 2.6508 given by the entropy.

- Human codes are uniquely decodable, in that a string can be decoded in only one way. For example, consider the string

1 1 0 1 1 1 0 0 0 0 0 1 0 0 1 1 1 1 1 0

1 1 0 | 1 1 1 0 | 0 0 | 0 0 | 1 0 | 0 1 | 1 1 1 1 0

3 4 0 0 1 2 5

Run length encoding

- Run length encoding (RLE) is based on a simple idea: to encode strings of zeros and ones by the number of repetitions in each string.
- For a binary image, there are many different implementations of RLE; one method is to encode each line separately, starting with the number of o's. So the following binary image:

0	1	1	0	0	0
0	0	1	1	1	0
1	1	1	0	0	1
0	1	1	1	1	0
0	0	0	1	1	1
1	0	0	0	1	1

Would be encoded as:

(123) (231) (0321) (141) (33) (0132)

Run length encoding (cont.)

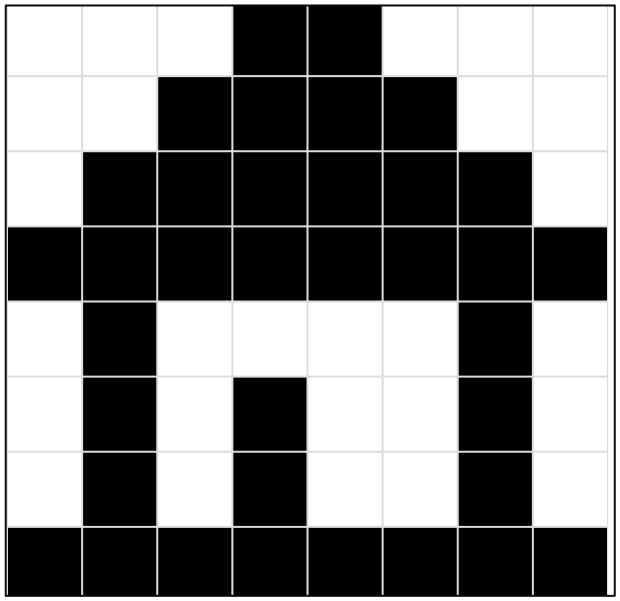
- Another method is to encode each row as a list of pairs of numbers; the rest number in each pair given the starting position of a run of 1's, and the second number its length.
- So the above binary image would have the encoding.

0	1	1	0	0	0
0	0	1	1	1	0
1	1	1	0	0	1
0	1	1	1	1	0
0	0	0	1	1	1
1	0	0	0	1	1

(22) (33) (1361) (24) (43) (1152)

Run length encoding (cont.)

Example of RLE encoding



(0323)

(0242)

(0161)

(8)

(011411)

(01111211)

(01111211)

(8)

Run length encoding (cont.)

- Greyscale images can be encoded by breaking them up into their bit planes.
- To give a simple example, consider the following 4-bit image and its binary representation:

10	7	8	9
11	8	7	6
9	7	5	4
10	11	2	1

1010	0111	1000	1001
1011	1000	0111	0110
1001	0111	0101	0100
1010	1011	0010	0001

0	1	0	1
1	0	1	0
1	1	1	0
0	1	0	1

0th plane

1	1	0	0
1	0	1	1
0	1	0	0
1	1	1	0

1th plane

0	1	0	0
0	0	1	1
0	1	1	1
0	0	0	0

2nd plane

1	0	1	1
1	1	0	0
1	0	0	0
1	1	0	0

3rd plane

Run length encoding (cont.)

- And then each plane can be encoded separately using our chosen implementation of RLE.
- However, there is a problem with bit planes, and that is that small changes of grey value may cause significant changes in bits. For example, the change from value 7 to 8 causes the change of all four bits, since we are changing the binary strings 0111 to 1000. The problem is of course exacerbated for 8-bit images.
- For RLE to be effective, we should hope that long runs of very similar grey values would result in very good compression rates for the code. But this may not be the case.

Run length encoding (cont.)

- A 4-bit image consisting of randomly distributed 7's and 8's would thus result in uncorrelated bit planes, and little effectieve compression.
- To overcome this difficulty, we may encode the grey values with their binary Gray codes.

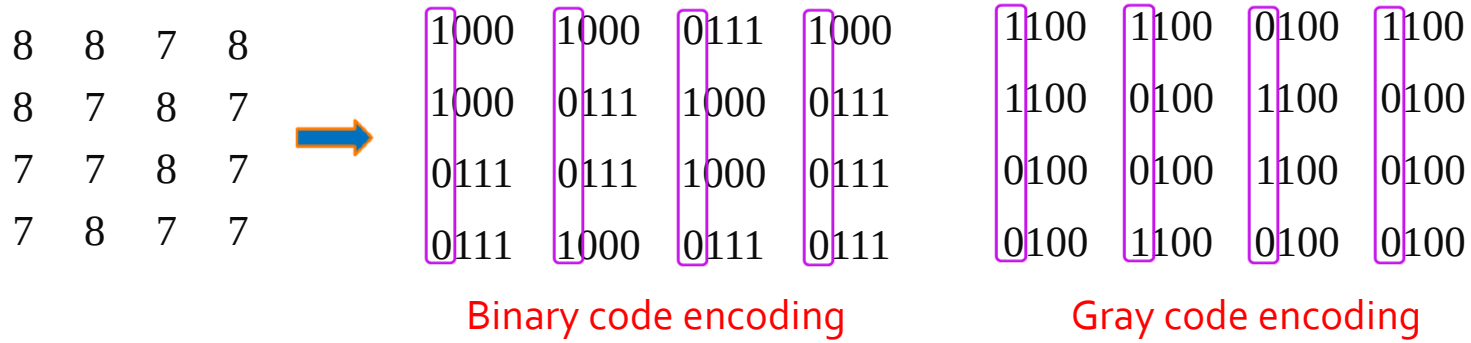
4-bit Gray code

Decimal no:	Gray Code			
15	1	0	0	0
14	1	0	0	1
13	1	0	1	1
12	1	0	1	0
11	1	1	1	0
10	1	1	1	1
9	1	1	0	1
8	1	1	0	0
7	0	1	0	0
6	0	1	0	1
5	0	1	1	1
4	0	1	1	0
3	0	0	1	0
2	0	0	1	1
1	0	0	0	1
0	0	0	0	0

- A Gray code is an ordering of all binary strings of a given length so that there is only one bit change between a string and the next. So a 4-bit Gray code is:

Run length encoding (cont.)

- To see the advantages, consider the following 4-bit image with its binary and Gray code encodings:



Run length encoding (cont.)

0	0	1	0
0	1	0	1
1	1	0	1
1	0	1	1

0^{th} plane

0	0	1	0
0	1	0	1
1	1	0	1
1	0	1	1

1^{th} plane

0	0	1	0
0	1	0	1
1	1	0	1
1	0	1	1

2^{nd} plane

1	1	0	1
1	0	1	0
0	0	1	0
0	1	0	0

3^{rd} plane

0	0	0	0
0	0	0	0
0	0	0	0
0	0	0	0

0^{th} plane

0	0	0	0
0	0	0	0
0	0	0	0
0	0	0	0

1^{th} plane

1	1	1	1
1	1	1	1
1	1	1	1
1	1	1	1

2^{nd} plane

1	1	0	1
1	0	1	0
0	0	1	0
0	1	0	0

3^{rd} plane

The binary bit planes and gray code planes

- Notice that the Gray code planes are highly correlated except for one bit plane, whereas all the binary bit planes are uncorrelated.

RLE code in Python

- Break 4-bit image into bit planes.
- Encode each plane separately using RLE

```
import numpy as np
# 4-bit image
c = np.array([[8,8,7,8],[8,7,8,7],[7,7,8,7],[7,8,7,7]])
# break into bit-planes
bps=[(c>>i)%2 for i in range(8)]
rle = np.zeros(8).tolist()
for i in range(8): # generate RLE code
   data = bps[i]
   data = data.flatten()
   changes = np.hstack(((data!= np.roll(data,1))*1,[1]))
   changes[0] = 1
   diffs = np.nonzero(changes)
   rle[i] = (diffs-np.roll(diffs,1)).tolist()[0]
   if data[0]==1:
      rle[i][0]=0
   else:
      rle[i] = rle[i][1:]
```

RLE code in Python (cont.)

8	8	7	8
8	7	8	7
7	7	8	7
7	8	7	7

Break 4-bit image into bit planes

0^{th} plane

0	0	1	0
0	1	0	1
1	1	0	1
1	0	1	1

1^{th} plane

0	0	1	0
0	1	0	1
1	1	0	1
1	0	1	1

2^{nd} plane

0	0	1	0
0	1	0	1
1	1	0	1
1	0	1	1

3^{rd} plane

1	1	0	1
1	0	1	0
0	0	1	0
0	1	0	0

Output RLE code

0	list	10	[2, 1, 2, 1, 1, 3, 1, 2, 1, 2]
1	list	10	[2, 1, 2, 1, 1, 3, 1, 2, 1, 2]
2	list	10	[2, 1, 2, 1, 1, 3, 1, 2, 1, 2]
3	list	11	[0, 2, 1, 2, 1, 1, 3, 1, 2, 1, ...]

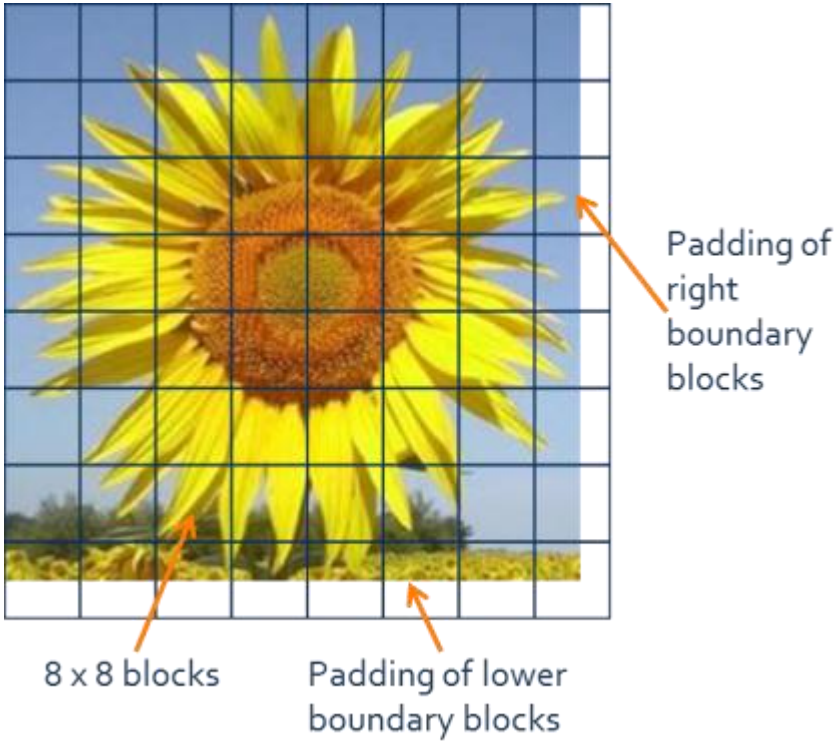
The lossy JPEG

- JPEG stands for "Joint Photographic Experts Group"
- JPEG compression is mainly discards much of the original information so the exact original image cannot be reconstructed from a JPEG file. This is called lossy compression.
- While this sounds bad, a photograph actually contains considerable information that the human eye cannot detect so this can be safely discarded.

The lossy JPEG (cont.)

The JPEG baseline compression scheme is applied as follow:

- The image values are shifted by subtracting 128 from each value.
- The image is partitioned into blocks of size 8×8. Each block is then independently transformed using the 8×8 DCT.
- If the image dimensions are not exact multiples of 8, the blocks on the lower and right hand boundaries may be only partially occupied.
- These boundary blocks must be padded to the full 8×8 block size and processed in an identical fashion to every other block.



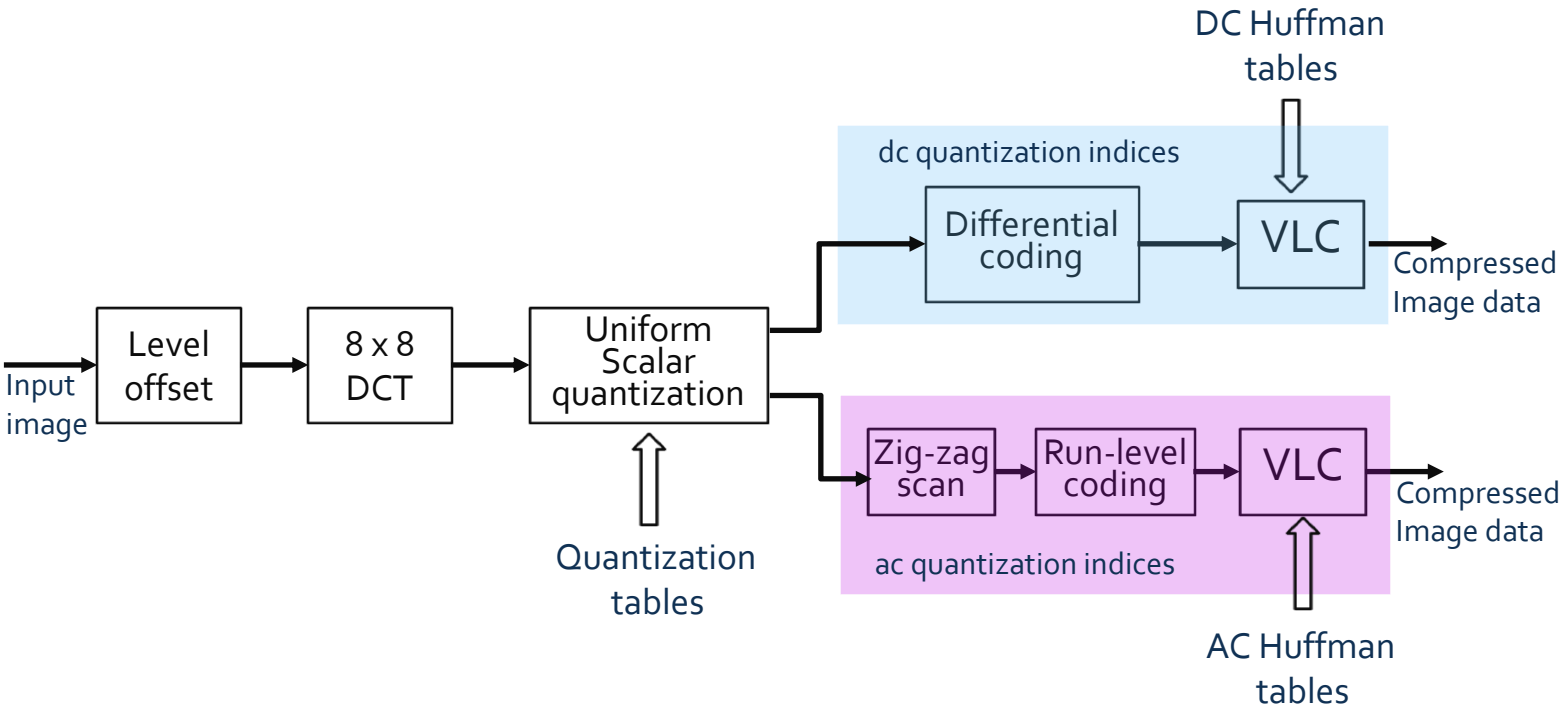
The lossy JPEG (cont.)

- The DCT coefficients are then quantized by dividing quantization values defined in the quantization tables.
- Most of the AC coefficients are reduced to zero and leave a very small number of nonzeroes are concentrated at the low spatial frequencies (the neighborhood of the DC coefficient).
- To improve the compression ratio, the quantized block is rearranged into the zig-zag order, then applied the runlength coding method to convert the sequence into the intermediate symbols.

The lossy JPEG (cont.)

- The DC coefficients are encoded by listing the difference between each value and the values from the previous block. This helps keep all values (except for the first) small.
- These values are then compressed using variable-length coding.
- All other values (known as the AC coefficients) are coded using a variable-length code that defines the coefficients values and number of preceding zeros.

Baseline JPEG coder



The lossy JPEG (cont.)

- To decompress, the steps above are applied in reverse:
  - The variable-length coding can be decoded with no loss of information.
  - Then the vector is read back into an 8 x 8 matrix.
  - The matrix is multiplied by the quantization table.
  - The inverse DCT is applied to the result.
  - The result is shifted back by 128 to obtain the original image block.

The lossy JPEG (cont.)

The first coefficients of each block which will be the largest elements and which are known as the DC coefficients. All other values are know as AC coefficients.

139	144	149	153	155	155	155	155
144	151	153	156	159	156	156	156
150	155	160	163	158	156	156	156
159	161	162	160	160	159	159	159
159	160	161	162	162	155	155	155
161	161	161	161	160	157	157	157
162	162	161	163	162	157	157	157
162	162	161	161	163	158	158	158

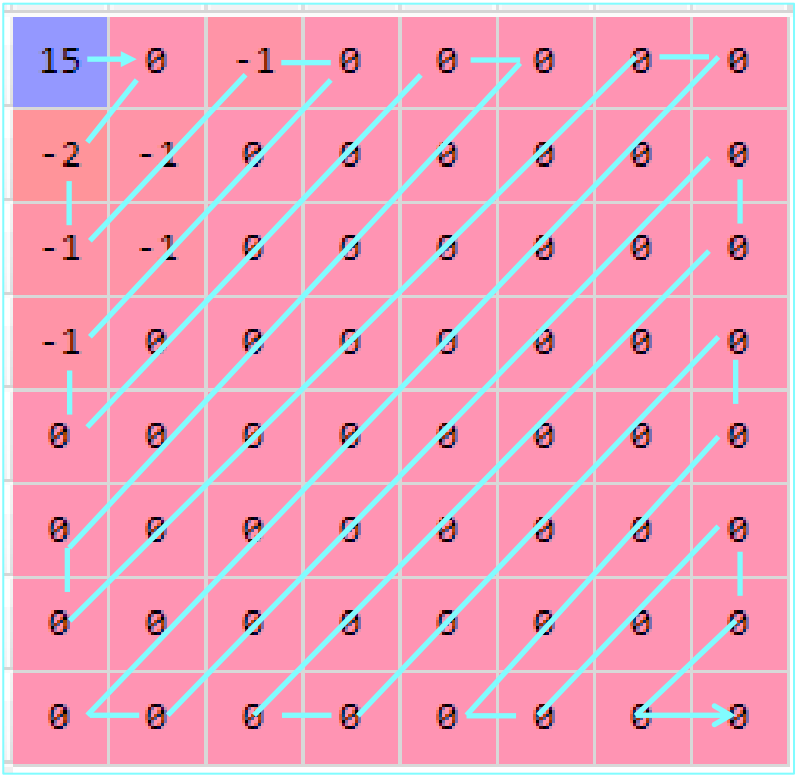
11	16	21	25	27	27	27	27
16	23	25	28	31	28	28	28
22	27	32	35	30	28	28	28
31	33	34	32	32	31	31	31
31	32	33	34	34	27	27	27
33	33	33	33	32	29	29	29
34	34	33	35	34	29	29	29
34	34	33	33	35	30	30	30

235.6	-1	-12.1	-5.2	2.1	-1.7	-2.7	1.3
-22.6	-17.5	-6.2	-3.2	-2.9	-0.1	0.4	-1.2
-10.9	-9.3	-1.6	1.5	0.2	-0.9	-0.6	-0.1
-7.1	-1.9	0.2	1.5	0.9	-0.1	-0	0.3
-0.6	-0.8	1.5	1.6	-0.1	-0.7	0.6	1.3
1.8	-0.2	1.6	-0.3	-0.8	1.5	1	-1
-1.3	-0.4	-0.3	-1.5	-0.5	1.7	1.1	-0.8
-2.6	1.6	-3.8	-1.8	1.9	1.2	-0.6	-0.4

The 8×8 original data block and its equivalent block of DCT coefficients

The lossy JPEG (cont.)

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



quantization table (left) and the Quantized DCT coefficients (right)

15 0 -2 -1 -1 -1 0 0 -1 EOB

The lossy JPEG (cont.)

142	144	147	150	152	153	154	154
149	150	153	155	156	157	156	156
157	158	159	161	161	160	159	158
162	162	163	163	162	160	158	157
162	162	162	162	161	158	156	155
160	161	161	161	160	158	156	154
160	160	161	162	161	160	158	157
160	161	163	164	164	163	161	160

-3	0	2	3	3	2	1	1
-5	1	0	1	3	-1	0	0
-7	-3	1	2	-3	-4	-3	-2
-3	-1	-1	-3	-2	-1	1	2
-3	-2	-1	0	1	-3	-1	0
1	0	0	0	0	-1	1	3
2	2	0	1	1	-3	-1	0
2	1	-2	-3	-1	-5	-3	-2

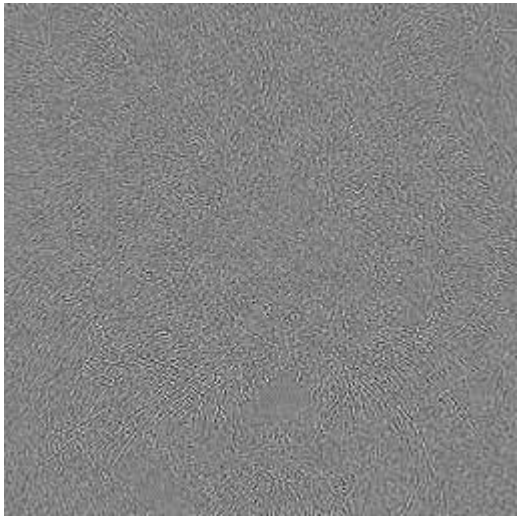
The reconstructed image and the differences between original and reconstructed values

The lossy JPEG (cont.)

There is no apparent difference between the original image and JPEG decompression. They look identical. But they are not identical.

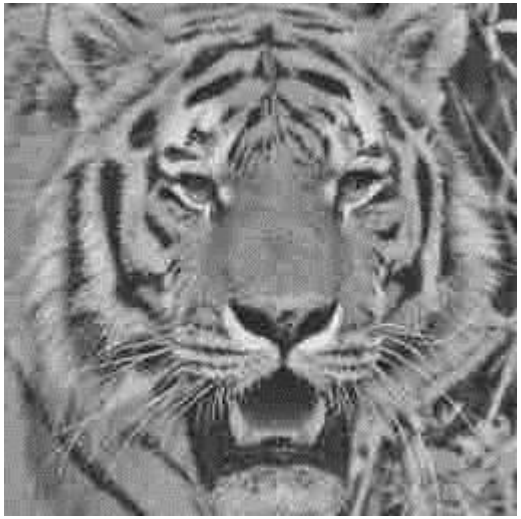
An image before and after JPEG compression and decompression and the difference between two images

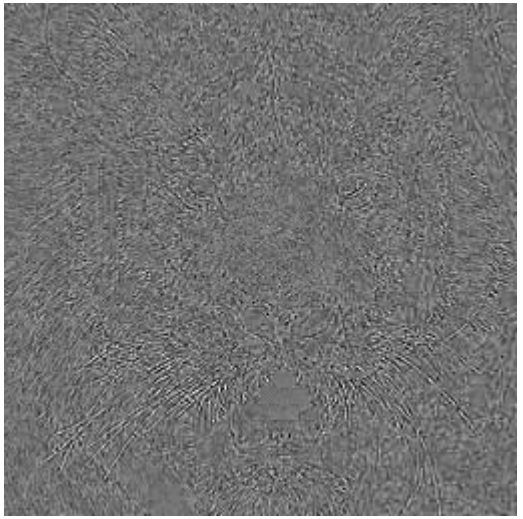
The lossy JPEG (cont.)



JPEG compression with a scale factor of 2 and difference between the original image and JPEG decompression

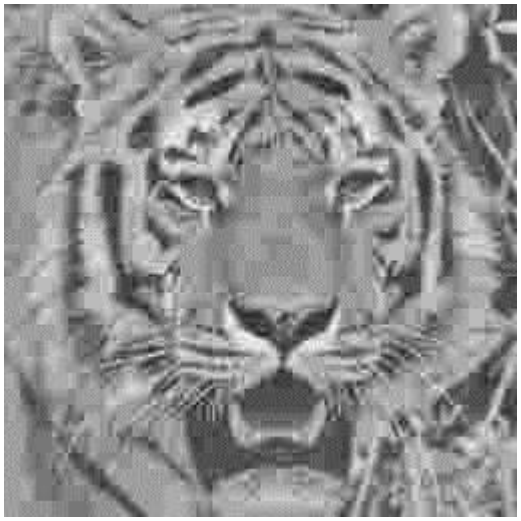
The lossy JPEG (cont.)

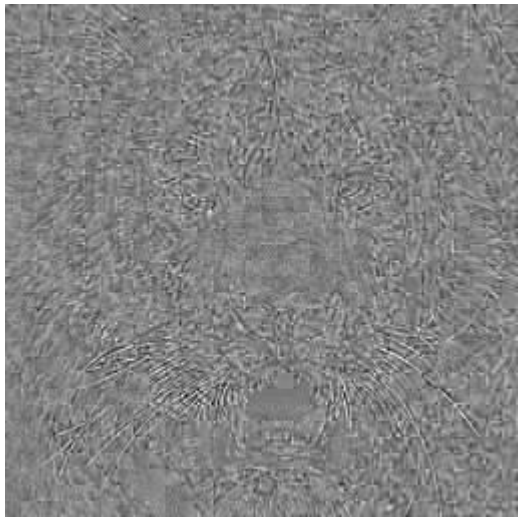




JPEG compression with a scale factor of 5 and difference between the original image and JPEG decompression

The lossy JPEG (cont.)





JPEG compression with a scale factor of 10 and difference between the original image and JPEG decompression

JPEG compresseion in Python

```
import numpy as np
from scipy.fftpack import dct
from scipy.fftpack import idct
import skimage.color as co
import skimage.io as io
def dct2(x):
   return dct(dct(x,norm = 'ortho').T,norm='ortho').T
def idct2(x):
   return idct(idct(x,norm='ortho').T,norm='ortho').T
def jpg_in(x,n):
   bd = dct2(np.float64(x)-128)
   return np.round(bd/(q*n))
def jpg_out(x,n):
   return np.round(idct2(x*q*n)+128)
q=np.array([[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]])
```

```
c = io.imread('tiger1.jpg').astype('float64')
c = co.rgb2gray(c)
rs,cs = c.shape
cj1 = np.zeros_like(c)
for i in range(0,rs,8):
   for j in range(0,cs,8):
      cj1[i:i+8,j:j+8] = jpg_in(c[i:i+8,j:j+8],2)
L = len(np.where(abs(cj1)<1)[0])
c1 = np.zeros_like(c)
for i in range(0,rs,8):
   for j in range(0,cs,8):
      c1[i:i+8,j:j+8] = jpg_out(cj1[i:i+8,j:j+8],2)
```

The lossy JPEG (cont.)

Image	Size (KB)	Items of information	Maxima Value	Minima Value
original	22	65536	255	0
Scale Factor 1	18	19225	58	-62
Scale Factor 2	13	12304	29	-31
Scale Factor 5	9	6641	12	-12
Scale Factor 10	7	3766	6	-6

Next Week Lecture (Week10)

- Lecture10:Wavelets Transform

Thank You