CS33713 – IMAGE PROCESSING

JPEG COMPRESSION

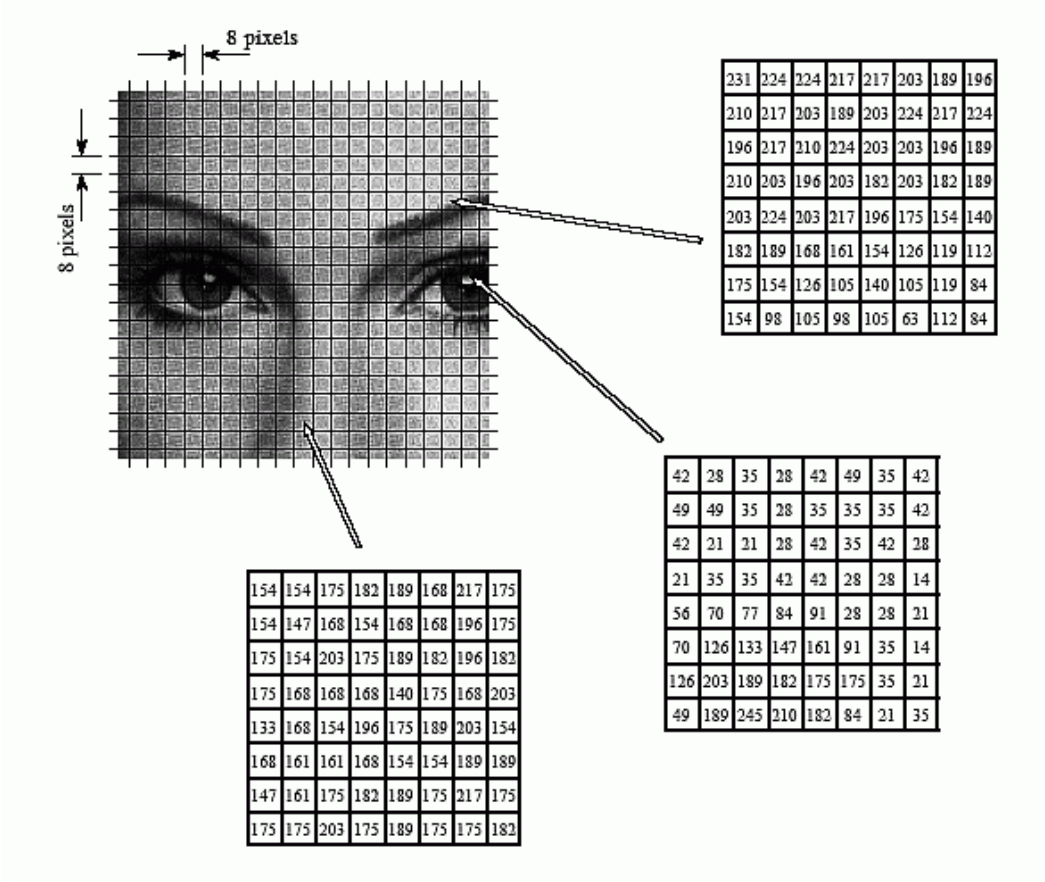
Thanushanth Kanagarajah

200636H

Contents

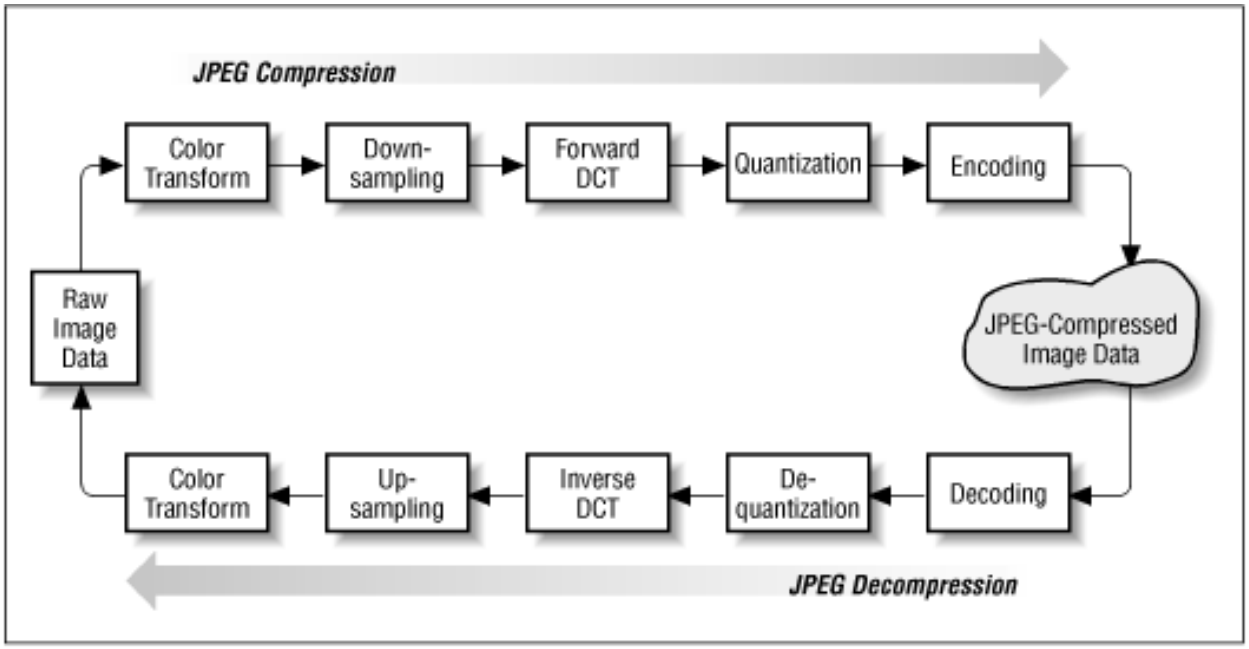
Introduction

The primary reason JPEG is a widely used standard image format is that it effectively reduces the amount of storage and transmission space required for images.
Uncompressed and in their raw format, photos can get pretty large, which can lead to proble ms like running out of storage for additional applications or data.
A lot of modern applications require this compression in order to function.
Although there are other picture compression techniques, JPEG is the most commonly used one.
It makes use of a lossy compression method that shrinks the size of the image by throwing a way some of the original image data.
There are tradeoffs between file size and image quality; more data is discarded to create a s maller image. In JPEG compression, the algorithm aims to remove information from the original image that is less noticeable to the human eye.



JPEG compression steps

JPEG compression, short for Joint Photographic Experts Group compression, is a widely-used image compression technique that significantly reduces the size of digital images while striving to maintain acceptable image quality. It accomplishes this through a series of well-defined steps, each contributing to the overall compression process. These steps involve transforming the image data, quantization, run-length encoding, and Huffman encoding, all working together to produce smaller image files that are still visually pleasing. This process is essential for efficient storage and transmission of images in various applications, from digital photography to web graphics and beyond.



Color space conversion

Images can be depicted using various color space formats, with RGB and YCrCb being two notable ones. In the initial step of the JPEG image compression process, the original image's color space is transformed into the YCrCb format. This choice is since the human visual system is more responsive to changes in brightness than to changes in color. In the YCrCb format, the Y component represents the image's luminance, while the other two components denote its chrominance. This conversion is achieved through a linear operation that essentially involves matrix multiplication.

Here I attach python code of the color space conversion.

```
imgOriginal = cv2.imread('pic.jpg', cv2.IMREAD_COLOR)
# convert BGR to YCrCb
img = cv2.cvtColor(imgOriginal, cv2.COLOR_BGR2YCR_CB)
width = len(img[0])
height = len(img)
```

```
# initially y will be the Blue component intensity values, cr will be Green and cb
will be Red component values
y = np.zeros((height, width), np.float32) + img[:, :, 0]
cr = np.zeros((height, width), np.float32) + img[:, :, 1]
cb = np.zeros((height, width), np.float32) + img[:, :, 2]



# size of the image in bits before compression
totalNumberOfBitsWithoutCompression = len(y) * len(y[0]) * 8 + len(cb) * len(cb[0])
* 8 + len(cr) * len(cr[0]) * 8

# channel values should be normalized, hence subtract 128
y = y - 128
cr = cr - 128
cb = cb – 128
```

Channel value normalization and chroma subsampling

The subsequent stage of the JPEG compression process, the algorithm focuses on normalizing channel values and downsampling the chroma components. Each pixel's value in the image is adjusted by subtracting 128 to normalize. However, the downsampling operation is specifically applied to the chroma components (Cr and Cb), leaving the luminance component (Y) untouched. This strategic choice aligns with our earlier discussion, recognizing that the human visual system places greater sensitivity on luminance than chrominance. By downsampling only the chroma components, the algorithm achieves a reduction in image size without significantly compromising the information essential for image quality, as much of the critical data resides in the

luminance component (Y). This tactic is known as chroma subsampling and is widely employed in various image and video processing pipelines.

There are four formats of commonly used chroma subsampling.

4:4:4 — uncompressed image with no chroma subsampling, transports both luminance and color data entirely.
4:2:2 — has half of the chroma of 4:4:4 and reduces the size of an uncompressed image signal by one-third with little to no visual difference.
4:2:0 — has one-quarter of the chroma of 4:4:4 and reduces the bandwidth of an uncompressed video signal by half compared to no chroma subsampling [3].

```
# 4: 2: 2 subsampling is used
SSH, SSV = 2, 2
# filter the chrominance channels using a 2x2 averaging filter # another type of
filter can be used
crf = cv2.boxFilter(cr, ddepth=-1, ksize=(2, 2))
cbf = cv2.boxFilter(cb, ddepth=-1, ksize=(2, 2))
# slicing step=SSV and SSH (Reduced to by SSV factor from rows and columns)
crSub = crf[::SSV, ::SSH]
cbSub = cbf[::SSV, ::SSH]
```

Discrete Cosine Transform

Another intriguing attribute of the human visual system that can be harnessed to effectively reduce image size without significant quality loss is its sensitivity to "frequency-dependent contrast." In this context, spatial frequency pertains to the patterns of light and dark elements in an image. High spatial frequencies encompass intricate details and sharp edges, while low spatial frequencies represent broader, global shapes within the image. The characteristic of frequency-dependent contrast sensitivity in our visual system implies that we are more likely to overlook smaller objects or finer details in an image when compared to larger, more prominent elements. This feature offers valuable insights into how image compression algorithms can prioritize the retention of essential information while economizing on storage space and transmission bandwidth.

```
# number of iteration on x axis and y axis to calculate the luminance cosine transform values
hBlocksForY = int(len(yDct[0]) / windowSize)  # number of blocks in the horizontal direction for luminance
vBlocksForY = int(len(yDct) / windowSize)  # number of blocks in the vertical direction for luminance
# number of iteration on x axis and y axis to calculate the chrominance channels cosine transforms values
hBlocksForC = int(len(crDct[0]) / windowSize)  # number of blocks in the horizontal direction for chrominance
vBlocksForC = int(len(crDct) / windowSize)  # number of blocks in the vertical direction for chrominance
```

Quantization

Quantization is a pivotal stage within the JPEG compression process, where the algorithm effectively trims down the data within an image. This operation involves the approximation and rounding of values in the Discrete Cosine Transform (DCT) domain, a domain that shifts the image's spatial information into frequency data. The outcome of this rounding process is a loss of precision, which is why JPEG is classified as a lossy compression format. The level of quantization is adjustable and serves as the lever that governs the delicate balance between image quality and file size; opting for a higher quantization level leads to more compact files but potentially compromises image quality. By applying quantization to the coefficients, JPEG intelligently discards less perceptible image details, achieving significant compression while safeguarding the overall image structure. This step is instrumental in striking the right equilibrium between compression efficiency and the preservation of acceptable image quality in JPEG images.

```
# define quantization tables
QTY = np.array([[16, 11, 10, 16, 24, 40, 51, 61],  # luminance quantization table
                [12, 12, 14, 19, 26, 48, 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]])
```

```
QTC = np.array([[17, 18, 24, 47, 99, 99, 99, 99],  # chrominance quantization table
                [18, 21, 26, 66, 99, 99, 99, 99],
                [24, 26, 56, 99, 99, 99, 99, 99],
                [47, 66, 99, 99, 99, 99, 99, 99],
                [99, 99, 99, 99, 99, 99, 99, 99],
                [99, 99, 99, 99, 99, 99, 99, 99],
                [99, 99, 99, 99, 99, 99, 99, 99],
                [99, 99, 99, 99, 99, 99, 99, 99]])
# define window size
windowSize = len(QTY)
```

Run length encoding

The next is Run-length encoding (RLE). That constitutes a vital phase in the JPEG compression process, with a primary objective of streamlining the encoded image data. In this step, the algorithm scans the quantized data to pinpoint consecutive runs of identical values, and it encodes these sequences succinctly by specifying the value and the length of each run. This clever technique effectively trims down data redundancy, yielding a more concise representation, which proves especially advantageous in regions of the image with uniform or near-uniform colors. The beauty of RLE lies in its significant contribution to reducing file size without sacrificing image quality. This makes it a valuable tool in JPEG compression, ensuring the efficient storage and transmission of images while maintaining their visual integrity.

```
# find the run length encoding for each channel
# then get the frequency of each component in order to form a Huffman dictionary
yEncoded = run_length_encoding(yZigzag)
yFrequencyTable = get_freq_dict(yEncoded)
yHuffman = find_huffman(yFrequencyTable)

crEncoded = run_length_encoding(crZigzag)
crFrequencyTable = get_freq_dict(crEncoded)
crHuffman = find_huffman(crFrequencyTable)

cbEncoded = run_length_encoding(cbZigzag)
cbFrequencyTable = get_freq_dict(cbEncoded)
cbHuffman = find_huffman(cbFrequencyTable)
```

Huffman(entropy) encoding

The final step is Huffman encoding. This technique, categorized as an entropy encoding algorithm, is employed to further condense the data slated for storage. During this step, commonly occurring values are encoded with fewer bits, while less frequent values are assigned a greater number of bits. This smart approach effectively trims down the average number of bits required per symbol, ensuring a more efficient representation of the image data. Huffman encoding is an essential final touch in the JPEG compression process, enhancing data compression while maintaining the integrity of the image.

```
def find_huffman(p: dict) -> dict:
    """
    returns a Huffman code for an ensemble with distribution p
    :param dict p: frequency table
    :returns: huffman code for each symbol
    """
    # Base case of only two symbols, assign 0 or 1 arbitrarily; frequency does
not matter
    if len(p) == 2:
        return dict(zip(p.keys(), ['0', '1']))

    # Create a new distribution by merging lowest probable pair
    p_prime = p.copy()
    a1, a2 = lowest_prob_pair(p)
    p1, p2 = p_prime.pop(a1), p_prime.pop(a2)
    p_prime[a1 + a2] = p1 + p2

    # Recurse and construct code on new distribution
    c = find_huffman(p_prime)
    ca1a2 = c.pop(a1 + a2)
    c[a1], c[a2] = ca1a2 + '0', ca1a2 + '1'

    return c
```

Conclusion

JPEG compression is a crucial process for efficiently reducing the storage and transmission requirements of digital images while striving to maintain acceptable image quality. This is achieved through a series of well-defined steps that work in tandem to compress image data effectively. Starting with color space conversion, the transformation into the YCrCb format optimizes the representation of brightness and color components. Channel value normalization and chroma subsampling ensure that the image retains its essential information while reducing redundancy. The Discrete Cosine Transform helps convert spatial information into frequency data, and quantization sacrifices some precision to achieve compression.

Run-length encoding comes into play to streamline encoded data, and Huffman encoding fine-tunes the data representation, using fewer bits for common values and more bits for less frequent ones. All these steps are strategically designed to reduce the average number of bits needed per symbol while still maintaining image quality. The process is a careful balance of compression efficiency and image preservation, making JPEG compression a widely used standard for various applications, from digital photography to web graphics, where efficient storage and transmission are essential.

• • •