컴퓨터 비젼

[HW09]

학번	201203393
분반	00
이름	김헌겸
과제번호	09

제출일 : 2016년 5월 28일 토요일

1. (MATLAB) Huffman Coding을 구현하여 cameraman.tif를 압축하시오.

- 이미지의 히스토그램을 생성하여, 해당 이미지에 대한 각 intensity의 확률을 구하라.

```
function output = huffman_table(im)

[x,y] = imhist(im);
x_size = size(x);
intensity = x./(x_size(1)^2);

huffarray = dlnode();

for i=1 : x_size
    huffarray(i) = dlnode(intensity(i), i-1);
end

huffarray = sort_all_dlnode(huffarray);

huffarray = delete_zero(huffarray);

huffarray = build_huffman_tree(huffarray);

huffman_root = huffarray(1);

%table = zeros(1, x_size(1));
table = cell(1,256);
table = build_huffman_table(huffman_root, table, '');

output = table;
end
```

우선 주어진 이미지에 대해서 허프만 트리를 만들고 허프만 테이블을 작성한다.

이미지의 히스토그램을 imhist를 사용하여 구한 다음, 그에 대한 intensity 값을 구한다. 그리고 dlnode 클래스를 사용하여 모든 intensity 값을 dlnode 객체로 만든다.

```
function output = sort_all_dlnode(huffarray)
    len = length(huffarray);
    for m=1 : len
        index = m;
        for k=m : len-1
            if huffarray(index).Data > huffarray(k+1).Data
                index = k+1;
            end
        end
        temp = huffarray(m);
        huffarray(m) = huffarray(index);
        huffarray(index) = temp;
    end
    output = huffarray;


end
```

그리고 나서는 sort_all_dlnode 함수를 호출하여 huffarray 배열의 값들을 intensity 값의 순서대로 오름차순 정렬을 시킨다. 그리고 intensity 값이 0이라면 해당 이미지에는 존재하지 않는 픽셀 값이므로 이를 제거하기 위해 delete_zero 함수를 사용한다.

```
function output = delete_zero(huffarray)
    len = length(huffarray);
    for i=1 : len
        if huffarray(1).Data == 0
            huffarray(1) = [];
        else
           break;
        end
    end
    output = huffarray;
end
```

이제 dlnode들을 이용하여 huffman tree를 구성한다.

```
function output = build_huffman_tree(huffarray)
    while length(huffarray) ~= 1
        first = huffarray(1);
        second = huffarray(2);
        newnode = dlnode(first.Data + second.Data, -1);
        if(first.Data > second.Data)
            insertLeft(newnode, first);
            insertRight(newnode, second);
        else
            insertLeft(newnode, second);
            insertRight(newnode, first);
        end

        huffarray(1) = [];
        huffarray(1) = [];
        huffarray(length(huffarray)+1) = newnode;

        huffarray = sort_other_node(huffarray);
    end
    output = huffarray;
end
```

오름차순으로 정렬되어 있으므로 첫 번째와 두 번째 노드를 뽑아서 둘의 intensity 값을 합하여 부모 노드를 만든다. 그러고 나서 이 노드를 리스트의 끝에 넣고 끝에서부터 재정렬을 실시한다. 이 작업을 모두 마치고 나면 intensity값이 1인 부모 노드만 남게 된다.

```
function output = sort_other_node(huffarray)
    len = length(huffarray);
    i = len;
    while i ~= 1
        if huffarray(i-1).Data < huffarray(len).Data
            break;
        end
        i = i - 1;
    end

    temp = huffarray(len);
    huffarray(i+1:len) = huffarray(i:len-1);
    huffarray(i) = temp;
    output = huffarray;
end
```

이제 huffman tree가 완성 되었으므로, 이로부터 huffman table을 만든다. 왼쪽 자식 노드는 0, 오른쪽 자식 노드는 1을 갖도록 하여 string형태로 code 값을 갖도록 한다. 이는 트리 구조이므로 재귀적으로 구현한다.

```
function output = build_huffman_table(huffnode, table, code)
    if huffnode.Pixel ~= -1
        table(huffnode.Pixel+1) = {code};
    else
        if ~isempty(huffnode.left)
            table = build_huffman_table(huffnode.left, table, strcat(code, '0'));
        end
        if ~isempty(huffnode.right)
            table = build_huffman_table(huffnode.right, table, strcat(code, '1'));
        end
    end
    output = table;
end
```

이제 huffman table이 완성되었으므로 이로부터 encoding과 decoding을 할 수 있다.

```
function output = huffman_encoding(im, huffman_table)
    encode_data = '';
    im_size = size(im);
    index = 2;
    output = zeros(1, (im_size(1) * im_size(1) / 8));

    output(1) = im_size(1);

    for i=1 : im_size(1)
        for j=1 : im_size(2)
            if im(i,j) == 255
                encode_data = strcat(encode_data, char(huffman_table(256)));
            else
                encode_data = strcat(encode_data, char(huffman_table(im(i,j)+1)));
            end

            while length(encode_data) > 8
                output(index) = uint8(bin2dec(encode_data(1:8)));
                index = index + 1;
                encode_data(1:8) = [];
            end
        end
    end

    if ~isempty(encode_data)
        output(index) = uint8(bin2dec(encode_data(1:length(encode_data))));
        output(index+1) = uint8(length(encode_data));
    end

end
```

encoding을 위해 각 leaf 노드들이 가지고 있는 code값을 하나의 문자열로 이어준다.

우선 이미지의 사이즈 값을 문자열의 처음에 넣어줌으로써 이미지의 크기 값을 헤더의 개념으로 가질 수 있도록 해준다. 그 다음 모든 픽셀들에 대해서 테이블에 존재하는 code값을 넣는다. 픽셀의 값은 0 ~ 255 이지만, 매트랩에서의 배열은 1부터 시작하므로 만약 pixel값이 255라면 배열의 256번째 요소의 코드 값을 받아오고, 아니라면 픽셀+1의 코드 값을 가져와 string에 붙이도록 한다. 그리고 매번 붙일 때마다 반환하기 위한 데이터 스트림을 구성한다. 문자열을 8개씩 잘라서 uint8의 형태로 만들어 주면 해당 이미지를 압축한 비트들로 구성된 데이터 스트림을 얻을 수 있다.

그리고 마지막으로 제일 끝 픽셀의 코드가 얼마일지 알 수 없기 때문에 마지막 값을 넣어주고 난 뒤, 이의 길이를 그 다음에 넣어주어 이 부분에 대한 예외처리도 진행한다.

```
function output = huffman_decoding(huffman_encoded_data, table)
    output = zeros(1, length(huffman_encoded_data));
    decode_data = '';

    length_array = cellfun('length', table);
    length_array = sort(length_array);
    for i=1 : length(length_array)
        if length_array(1) == 0
            length_array(1) = [];
        else
            break;
        end
    end
    min_length = length_array(1);

    im_x = huffman_encoded_data(1);
    huffman_encoded_data(1) = [];

    index = 1;
    while length(huffman_encoded_data) > 2
        decode_data = strcat(decode_data, dec2bin(huffman_encoded_data(1), 8));
        huffman_encoded_data(1) = [];

        bit_place = min_length;
        while length(decode_data) > bit_place
            table_found = 0;
            for j=1 : length(table)
                if strcmp(table(j), decode_data(1 : bit_place))
                    output(index) = uint8(j-1);
                    index = index + 1;
                    decode_data(1:bit_place) = [];
                    bit_place = min_length;
                    table_found = 1;
                    break;
                end
            end
            if ~table_found
                bit_place = bit_place + 1;
            end
```

```
        end

    end

    if ~isempty(huffman_encoded_data)
        decode_data = strcat(decode_data, dec2bin(huffman_encoded_data(1), huffman_encoded_data(2)));
    end

    bit_place = min_length;
    while length(decode_data) >= bit_place
        table_found = 0;
        for j=1 : length(table)
            if strcmp(table(j), decode_data(1 : bit_place))
                output(index) = uint8(j-1);
                index = index + 1;
                decode_data(1:bit_place) = [];
                bit_place = min_length;
                table_found = 1;
                break;
            end
        end
        if ~table_found
            bit_place = bit_place + 1;
        end
    end
    output = reshape(output, [], im_x);
    output = output';
end
```

decoding은 encoding된 값과 table을 이용하여 변환한다. 우선 encode된 데이터의 처음에 있는 이미지 크기 값을 가져온 뒤, 테이블에 있는 각 코드 값들의 길이를 비교하여 최소 코드 길이를 구해낸다. 이제 역시 문자열을 만드는 작업을 한다. 매개변수로 전달 받은 uint8 데이터를 바이너리 문자열로 바꾸는데, 이때, 매번 바꾸는 순간마다 테이블에서 코드 값을 찾아서 픽셀 값을 얻어낸다. 최소 코드 길이를 알고 있기 때문에 초기화 길이를 최소 코드 길이로 하여 한 비트 씩 늘려나가며 모든 코드를 대조해 보아 픽셀 값을 만들고 나면, 제일 마지막 픽셀에 대한 처리를 특별하게 따로 해주고 역시 코드에서 찾아 픽셀 값을 찾아준다. 그리고 마지막으로는 일렬로 구성되어 있는 픽셀들을 reshape를 사용하여 원본 이미지의 크기대로 돌려준다. 이렇게 해서 반환되는 값은 원본 이미지가 된다.

```
function hw_1()

im = imread('cameraman.tif');

table = huffman_table(im);

tic
enc = huffman_encoding(im, table);
toc

enc_elapse_time = toc;

tic
dec = huffman_decoding(enc, table);
toc

dec = uint8(dec);

dec_elapse_time = toc;

whos enc im dec

if im == dec
    disp('two image is totally same');
end

figure,
subplot(1,2,1), imshow(uint8(im))
subplot(1,2,2), imshow(uint8(dec))
```

```
>> hw_1
Elapsed time is 3.655594 seconds.
Elapsed time is 216.707526 seconds.
  Name         Size                Bytes  Class

  dec        256x256               65536  uint8
  enc          1x57714             57714  uint8
  im         256x256               65536  uint8

two image is totally same
>>
```

마지막으로 cameraman.tif와 비교해본다. tictoc을 사용하여 인코딩, 디코딩에 소요된 시간 또한 측정하였다.

결과를 보면 encode 된 데이터의 크기는 57714이고 원본 데이터의 크기가 65536 이므로 88%정도의 데이터로 압축 되었음을 알 수 있다. 그리고 인코딩과 디코딩의 결과로 얻어진 이미지가 완벽하게 원본 이미지와 같음을 통해 huffman coding이 무손실 압축임도 알 수 있다. 그리고 무손실 압축인 만큼 decoding에 시간이 많이 걸리고 있다는 것 또한 알 수 있다.