University of Burgundy

(Lab Report) :: Image Compression

Prof. Christophe Stolz

Student-1: VAMSHI KODIPAKA

Student-2: PARMAR HARDIKSINH

---

1. Arithmetic Coding:

Explanation: Arithmetic coding is a common algorithm used in both lossless and lossy data compression algorithms. It is an entropy encoding technique, in which the frequently seen symbols are encoded with fewer bits than rarely seen symbols. It has some advantages over well-known techniques like Huffman coding.

How does Arithmetic coding work:

It converts the entire input data into a single floating point number n where (0.0 <= n < 1.0). The interval is divided into sub-intervals in the ratio of the probability of occurrence frequencies. For a startpoint and endpoint of an entire range the lower-limit of a character range is the upper-limit of the previous character given by start point + cumulative frequency X (endpoint –start point). Therefore, each interval corresponds to one symbol. The first symbol restricts the tag position to be in one of the intervals. The reduced interval is partitioned recursively as more symbols are processed.

Observation: once the tag falls into an interval, it never gets out of it.

NewHigh:=OldLow+Range*HighRange(X);
NewLow:=OldLow+Range*LowRange(X);
Range=OldHigh−OldLow

OldLow and ldHigh are initialized to 0 and 1.

******* ENCODER ********

Reading inputs:

```
%~~~~~~~~~~~~~~~~~~~ Arithmetic Encoder :: Test1~~~~~~~~~~~~~~~~~~%

Symb = ['A','C','T','G'];
Prob = [0.5 , 0.3 ,0.15 , 0.05];
Seq = [ 'A' ,'C' , 'T', 'A', 'G' , 'C','G','C'];

Encode = Arith_Encoder (Symb , Prob , Seq)

%~~~~~~~~~~~~~~~~~~~ Arithmetic Coding :: Test2~~~~~~~~~~~~~~~~~~%

Symb = ['B','E','_','A'];
Prob = [0.5 , 0.3 ,0.15 , 0.05];
Seq = [ 'B','E','_','A','_','B','E','E'];

Encode = Arith_Encoder (Symb , Prob , Seq)
```

Function Implementation:

```
%~~~~~~~~~~~~~~~~~~~ Arithmetic Encoder ~~~~~~~~~~~~~~~~~~~~%
function Low  = Arith_Encoder(Symb , Prob , Seq)

newHigh = zeros(length(Symb));
newLow = zeros(length(Symb));

tempH = 0;
tempL = 1;




for iter = 1:length(Symb)

    newHigh(iter) = 1 - tempH;
    tempH = tempH + Prob(iter);

    tempL = tempL - Prob(iter);
    newLow(iter) = tempL;
end

Low = 0;
High = 1;

iter = 1;
while iter < length(Seq)+1

    INDEX = 0;

    for iter2 = 1: length(Symb)
        if Seq(iter) == Symb(iter2)
            INDEX = iter2;
        end
    end

    range = High - Low ;
    High  = Low + ( range * newHigh(INDEX));
    Low = Low + (range * newLow (INDEX));
    iter = iter + 1;
end

end
```

Encoder Outputs:

```
Encode =    0.6189    %For Test input-1
Encode =    0.6077    %For Test input-2
```

******* DECODER ********

Reading inputs:

```
%~~~~~~~~~~~~~~~~~~~ Arithmetic Decoding ~~~~~~~~~~~~~~~~~~~%

Arith_Decoder(Symb , Prob , Encode)
```

Function Implementation:

```
%~~~~~~~~~~~~~~~~~~ Arithmetic Decoder ~~~~~~~~~~~~~~~~~~~~%

function seq = Arith_Decoder(Symb , Prob , code)

newHigh = zeros(length(Symb));
newLow = zeros(length(Symb));

tempH = 0;
tempL = 1;

for iter = 1:length(Symb)

    newHigh(iter) = 1 - tempH;
    tempH = tempH + Prob(iter);

    tempL = tempL - Prob(iter);
    newLow(iter) = tempL;
end

i = code;
THRESHOLD = 0.01;
seq = [];
while i >= THRESHOLD
    INDEX = 1;
    for iter2 = 1: length(newHigh)
        if i < newHigh(iter2) && i > newLow(iter2)
            INDEX=iter2;
            seq = [seq, Symb(INDEX)];

            break
        end

    end
    range = newHigh(INDEX) - newLow(INDEX);
    i =  vpa(i - newLow(INDEX))./range;


end

end
```

Decoder Outputs:

```
ans =    'ACTAGC'             %For Test input-1
ans =    'BE_'                %For Test input-2
```

2. Huffman Coding:

Explanation: Huffman coding is a lossless data compression algorithm. The idea is to assign variable-length codes to input characters, lengths of the assigned codes are based on the frequencies of corresponding characters. The most frequent character gets the smallest code and the least frequent character gets the largest code.

The variable-length codes assigned to input characters are Prefix Codes, means the codes (bit sequences) are assigned in such a way that the code assigned to one character is not prefix of code assigned to any other character. This is how Huffman Coding makes sure that there is no ambiguity when decoding the generated bit stream.

Reading inputs:

```
%% ~~~~~~~~~~~~~~~~~~~~~ Huffman Coding ~~~~~~~~~~~~~~~~~~~~~

% Read an image
I = imread('lena.bmp');

% Convert image into probability list (in descending order)
I_Prob = Prob_Vector(I);

Huff_Result = Huff_Encoder(I_Prob);
```

Function Implementation:

```
%~~~~~~~~~~~~~~~~~~ Huffman Encoder ~~~~~~~~~~~~~~~~~~~~%

function code = Huff_Encoder( Prob )

code = cell(1,length(Prob));
tempP = Prob;
queue = [];

mark = length(Prob);

    for i= length(tempP) : -1 : 1
        if i == 1
            if isempty(queue)
            break;
            else
                code{i} = [ {1},code{i}];
                for j1 = i : length(Prob)
                    code{j1} = [ {0},code{j1}];
                end
            end

            elseif ( tempP(i) <= tempP(i-1))
            if ~isempty(queue)

                for q =1:length(queue)
                    if queue{q}(1)< tempP(i)+ tempP(i-1)
                        tempP(i) = tempP(i)+ queue{q}(1);
                        code{i-1} = [ {1},code{i-1}];
                        for j = i : mark
```

```
                            code{j} = [ {0},code{j}];
                        end
                        tempP(i-1) = tempP(i)+ tempP(i-1);

                        for j = i-1 : queue{q}(2)-1
                           code{j} = [ {1},code{j}];
                        end

                        if q == length(queue)
                            destn = length(Prob);
                        else
                             destn = queue{q+1}(2);
                        end

                        for j = queue{q}(2) : destn
                           code{j} = [ {0},code{j}];
                        end

                        queue = queue(1:length(queue)-1);
                    end
                    break;

                end

            else
             code{i-1} = [ {1},code{i-1}];
            for j = i : mark
                code{j} = [ {0},code{j}];
            end
            tempP(i-1) = tempP(i)+ tempP(i-1);
            end

            else
            queue = [{[tempP(i), i]} ,queue];
            mark = i-1;
        end
    end

end
```

Output:

img_prob =

[0.010387420654296875, 0.010261535644453125, 0.010196685791015625, 0.009960174560546875, 0.009510040283203125, 0.009471893310546875, 0.009407043457031 25, 0.009281158447265625, 0.009189605712890625, 0.009120941162109375, 0.009117126464843 75, 0.009098052978515625, 0.009056091308593 75, 0.008930206298828125,

………………….

0.000030517578125, 0.000030517578125, 0.000026702880859375, 0.000026702880859375, 0.00002288818359375, 0.00000762939453125, 0.000003814697265625, 0.000003814697265625, 0.000003814697265625, 0.000003814697265625, 0.000003814697265625]

******Probability Vector in Huffman coding*******

```
function temp1 = Prob_Vector( img )

[M , N]= size(img);
temp1 = zeros(1,256);
len = length(temp1);
for i = 1:M
    for j = 1:N
        temp1(img(i,j)) = temp1(img(i,j)) + 1;
    end
end

temp1 = sort (temp1 ,'descend');
temp1 = vpa(temp1./ (M*N));
marker = 0;
for i=1:len
    if temp1(i) > 0
        continue;
    else
        marker = i;
        break;
    end
end

temp1 = temp1(1:marker-1);

end
```

3. JPEG Coding:

Explanation: JPEG uses a lossy form of compression based on the discrete cosine transform (DCT). This mathematical operation converts each frame/field of the video source from the spatial (2D) domain into the frequency . A perceptual model based loosely on the human psychovisual system discards high-frequency information, i.e. sharp transitions in intensity, and color hue. In the transform domain, the process of reducing information is called quantization.

In simpler terms, quantization is a method for optimally reducing a large number scale (with different occurrences of each number) into a smaller one, and the transform-domain is a convenient representation of the image because the high-frequency coefficients, which contribute less to the overall picture than other coefficients, are characteristically small-values with high compressibility. The quantized coefficients are then sequenced and losslessly packed into the output bitstream.

Nearly all software implementations of JPEG permit user control over the compression-ratio (as well as other optional parameters), allowing the user to trade off picture-quality for smaller file size. In embedded applications, the parameters are pre-selected and fixed for the application.

The compression method is usually lossy, meaning that some original image information is lost and cannot be restored, possibly affecting image quality. There is an optional lossless mode defined in the JPEG standard. However, this mode is not widely supported in products.

Reading inputs:

```
%% ~~~~~~~~~~~~~~~~~~~~~~ JPEG Coding ~~~~~~~~~~~~~~~~~~~~~~

Jpeg_Result = JPEG(I);
```

Function Implementation:

```
%~~~~~~~~~~~~~~~~~~ JPEG Encoder ~~~~~~~~~~~~~~~~~~~~%
function T = JPEG (img)

% Level shift image by 2^(m-1)
img = double(img) - 128;
[M , N ] = size(img);
imshow(img);
imwrite(img,'Written_Input.bmp','bmp');

% Default JPEG normalizing array
Z = [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];


%Zigzag reordering Pattern

order = [1 9 2 3 10 17 25 18 11 4 5 12 19 26 33  ...
    41 34 27 20 13 6 7 14 21 28 35 42 49 57 50 ...
    43 36 29 22 15 8 16 23 30 37 44 51 58 59 52 ...
    45 38 31 24 32 39 46 53 60 61 54 47 40 48 55 ...
    62 63 56 64];
H = dctmtx(8);

%2D DCT cosine Transform
B = blkproc(img, [8 8], 'P1 * x * P2', H, H');

%Quantization and Normalization
T = blkproc(B, [8 8], 'round(x ./ P1)', Z);
figure(), imshow(T);
imwrite(T,'Written_Output.bmp','bmp');

% break 8 x 8 blocks into columns

T = im2col(T, [8 8], 'distinct');

% get number of blocks
dim = size(T, 2);
T = T(order, :);

% create end-of-block symbol
EOB = max(img(:)) + 1;
r = zeros(numel(T) + size(T, 2), 1);
count = 0;
```

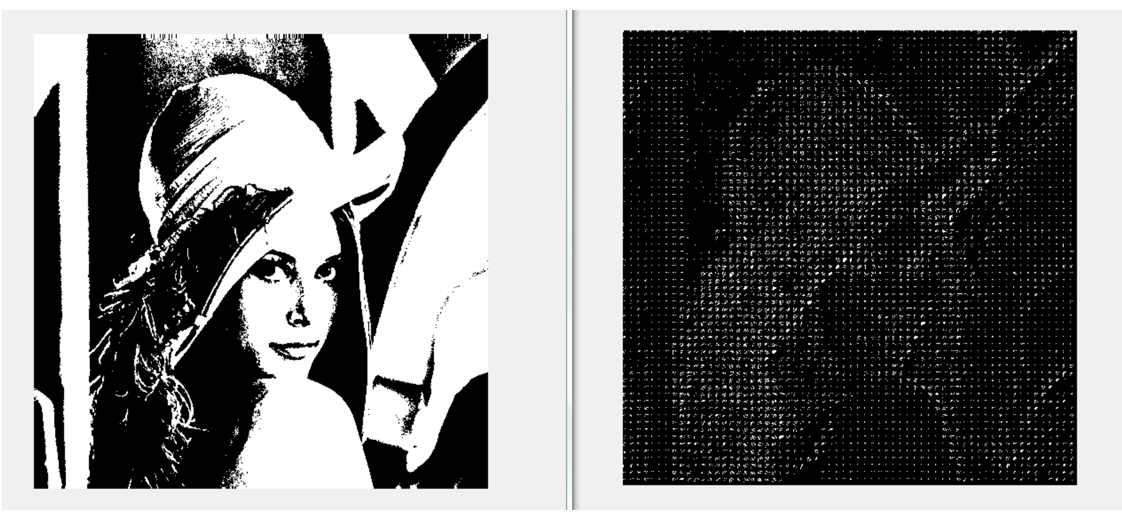
```
% process one block(one column) at a time
for j = 1:dim
    i = find(T(:, j), 1, 'last');    % find last non-zero element
    if isempty(i)                    % check if there are no non-zero values
        i = 0;
    end
    p = count + 1;
    q = p + i;
    r(p:q)  = [T(1:i, j); EOB];      % truncate trailing zeros, add eob
    count = count + i + 1;           % and add to output vector
end

r((count + 1):end) = [];             % delete unused portion of r
T            = struct;
T.size       = uint16([M N]);
T.blockNo = uint16(dim);
T.huffman    = Huff_Encoder(r);
end
```

Output:

result =
struct with fields:

size:	**[512 512]**
blockNo:	**4096**
huffman:	**{1×51457 cell}**



Lena.bmp image observations for Huffman Coding