

آرین تشکر - 40023494

تکلیف اول درس سیستم های چند رسانه ای پیشرفته

ساختار فایل تحویلی:

پوشه ی اصلی فایل تکلیف شامل یک فایل Report.pdf (فایلی که هم اکنون در حال خواندن آن هستید) و یک پوشه به نام Deliverables می باشد. در پوشه ی Deliverables چندین فایل سورس متلب قرار گرفته است که هر کدام از آن ها برای قسمتی از تکلیف مورد استفاده قرار می گیرند (به خصوص فایل main.m شامل کد تست بخش های مختلف می باشد). همچنین در این پوشه، دو پوشه ی دیگر به نام های Inputs و Results قرار دارند که به ترتیب فایل های ورودی و نتایج را در خود جای می دهند.

سوال 1) تابعی برای محاسبه PSNR بین دو ماتریس:

سورس کد های مربوط: Deliverables\PSNR.m و Deliverables\MSE.m

کد ها:

MSE.m:

```
function mse = MSE(img1, img2)
    % https://en.wikipedia.org/wiki/Mean\_squared\_error
    mse = mean((img1 - img2).^2, "all");
end
```

برای محاسبه MSE کفایت اختلاف مقادیر درایه های هر یک از دو ماتریس ورودی را نظیر به نظیر بدست بیاوریم، سپس تمامی درایه ها را نظیر به نظیر به توان 2 رسانده و در نهایت روی تمامی ابعاد ماتریس، میانگین بگیریم (معادل این است که آرایه را ابتدا به یک بردار flatten کنیم و سپس از آن بردار میانگین بگیریم).

PSNR.m:

```
function psnr = PSNR(img1, img2)
    % https://en.wikipedia.org/wiki/Peak\_signal-to-noise\_ratio
    psnr = 10 * log10(255^2/MSE(img1, img2));
end
```

تابع محاسبه ی PSNR مستقیماً از تعریف آن در ویکی پدیا استخراج شده است. نکته ی مهم در این تابع آن است که فرض شده که از این تابع برای محاسبه ی PSNR بین دو تصویر استفاده خواهد شد و بنابراین مقدار MAX_I را به صورت ثابت، برابر 255 قرار داده شده است.

سوال 2) تابعی برای تولید بهترین عکس Black&White بر اساس معیار PSNR:

سورس کد مربوط: Deliverables\GetBestBNW.m

نتایج مربوط: Deliverables\Results\BnW*

کد:

GetBestBNW.m

```
function [bnw, psnr] = GetBestBNW(img_dir)
    % Converts an RGB picture to the best possible binary image w.r.t. PSNR.
    img = imread(img_dir);
    % flatten the original image by taking mean of each channel
    flattened_img = mean(img, 3);
    % threshold at 127.5
    bnw = 255 * (uint8(flattened_img / 255));
    % broadcast B&W image to 3 channels
    broadcast_bnw = repmat(bnw, [1, 1, 3]);
    psnr = PSNR(broadcast_bnw, img);
    % save results to file
    [~,filename,extension] = fileparts(img_dir);
    imwrite(bnw, join(["Results/BnW/" filename "_bnw" extension], ""));
    save(join(["Results/BnW/" filename "_psnr.mat"], ""), "psnr");
end
```

برای محاسبه ی بهترین تصویر B&W ممکن بر اساس معیار PSNR لازم است که مقدار MSE کمینه شود. از آنجایی که در MSE هیچ یک از جملات نمی توانند منفی باشند، بنابراین بهینه سازی به صورت حریصانه و برای هر پیکسل نهایتاً منجر به جواب بهینه برای کل تصویر خواهد شد. در این صورت برای یک پیکسل از یک تصویر RGB مجموع مربعات خطا ها خواهد بود (از میانگین صرف نظر می کنیم چرا که در کمینه شدن مقدار MSE تاثیری ندارد و صرفاً هدف ما کمینه کردن مجموع مربعات خطا هاست):

$$\text{SquaredError}(\text{pixel}_{BnW}, \text{pixel}_{original}) \\ = (x - r_{original})^2 + (x - g_{original})^2 + (x - b_{original})^2, x \in \{0, 255\}$$

برای کمینه شدن این حاصل جمع، مقدار مشتق آن را نسبت به x برابر 0 قرار می دهیم و معادله ی بدست آمده را حل می کنیم:



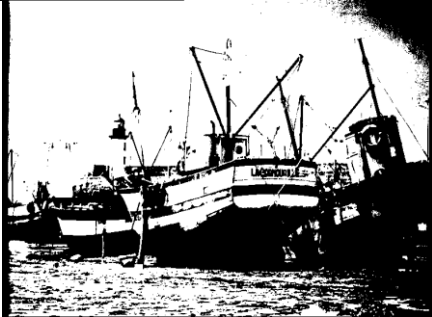



$$\frac{d(\text{SquaredError})}{dx} = 2(x - r_{original}) + 2(x - g_{original}) + 2(x - b_{original}) = 0 \\ \rightarrow x = \frac{r_{original} + g_{original} + b_{original}}{3}, x \in \{0, 255\}$$

از آنجایی که دامنه ی x فقط می تواند مقادیر 0 و 255 را اختیار کند، بهترین مقدار ممکن برای x آن است که آن را نسبت به 0 و 255 گرد کنیم. یعنی:

$$\begin{aligned}
 x &= \frac{r_{original} + g_{original} + b_{original}}{3}, x \in \{0, 255\} \xrightarrow{\div 255} \frac{x}{255} \\
 &= \frac{1}{255} \left(\frac{r_{original} + g_{original} + b_{original}}{3} \right), x \in \{0, 1\} \rightarrow x \\
 &= 255 \times \left[\frac{1}{255} \left(\frac{r_{original} + g_{original} + b_{original}}{3} \right) \right]
 \end{aligned}$$

که در آن علامت براکت نمایانگر عملگر گرد کردن (round) می باشد. این امر معادل آن است که میانگین 3 کانال r، g و b را محاسبه کنیم و سپس اعداد بزرگتر (یا مساوی) 127.5 را برابر 255 و اعداد کوچکتر (یا مساوی) 127.5 را برابر صفر قرار دهیم.

نکته ی مهمی که در مورد این سوال وجود دارد آن است که از آنجایی که برای محاسبه ی PSNR بین دو تصویر ابعاد دو تصویر باید با هم برابر باشند و با توجه به این که تصویر B&W فقط یک کانال دارد (در برابر 3 کانال R، G و B تصویر رنگی) باید تصویر B&W را در طول بعد سومش (کانال Broadcast) کنیم تا به شکل یک تصویر RGB در بیاید. برای این کار کافیست که این عکس را روی بعد سوم، دو بار کپی کنیم.

مقدار PSNR	تصویر خروجی	تصویر اصلی
27.18710377017659		
26.175355954664994		
30.173655364136984		

سوال 3) مختصری در مورد فرمت PNM (Pixel aNyMap format):

خانواده فایل های PNM شامل سه فرمت کلی به نام های Portable GrayMap ، Portable PixMap format (PPM) ، Portable BitMap format (PBM) و format (PGM) می باشند که هر کدام برای منظور خاصی استفاده می شوند. فرمت PNM یکی از اولین فرمت های ذخیره سازی تصویر بود و هدف اصلی از ایجاد آن، ایجاد یک فریم ورک ساده برای ذخیره سازی و خواندن اطلاعات یک تصویر بود که در قالب یک ایمیل (و به صورت ASCII) نیز قابل انتقال باشد. فایل های خانواده PNM می توانند به صورت باینری و یا ASCII نوشته شوند. در ابتدای هر فایل PNM عدد جادویی (Magic Number) این فایل قرار دارد که به فرمت Px است که X می تواند هر عدد طبیعی بین 1 تا 6 باشد و با توجه به این که فایل به صورت ASCII ذخیره شده است و یا Binary هر یک از این 6 حالت به یکی از سه فرمت خانواده ی PNM نگاشت می شوند. در جدول زیر – که از ویکی پدیا استخراج شده است – می توانید مشخصات هر یک از این فرمت ها را مشاهده کنید:

Type	Magic number		Extension	Colors
	ASCII (plain)	Binary (raw)		
Portable BitMap	P1	P4	.pbm	0–1 (white & black)
Portable GrayMap	P2	P5	.pgm	0–255 (gray scale), 0–65535 (gray scale), variable, black-to-white range
Portable PixMap	P3	P6	.ppm	16777216 (0–255 for each RGB channel), some support for 0-65535 per channel

علت وجود فرمت ASCII این است که به راحتی توسط انسان قابل خواندن هستند و همچنین فایلی که به این صورت ذخیره شده باشد مشکل platform dependency ندارد و می توان به راحتی از آن روی هر سیستمی استفاده کرد. در مقابل، استفاده از فرمت binary با وجود این که حجم بسیار کمتری از فرمت ASCII اشغال می کند، ممکن است به علت endian های مختلف در سیستم های مختلف دچار ایراد compatibility شود. به طور کلی هر فایل PNM با یک Magic Number به همراه ابعاد تصویر مورد نظر (اول Width و سپس Height) شروع می شود که این اطلاعات فارغ از روش ذخیره سازی فایل (binary یا ASCII) همواره به صورت ASCII ذخیره می شوند با این حال تفاوت هایی بین فرمت های مختلف وجود دارد دارد:

- PBM: در این فرمت از آنجایی که تصویر باینری است کل Header فایل شامل عدد جادویی نمایانگر فرمت PBM به همراه ابعاد عکس می باشد و در خطوط بعدی داده های شامل مقادیر پیکسل های عکس قرار داده می شوند.
- PGM: در این فرمت علاوه بر ابعاد تصویر مانند فرمت PBM یک پارامتر جدید به عنوان maximum value نیز بعد از ابعاد در تصویر در Header فایل آن نوشته می شود. این عدد مشخص می کند فاصله ی بین سفید مطلق تا سیاه مطلق به چند قسمت تقسیم شده است. مثلاً اگر maximum value برابر 15 باشد یعنی 15 سایه ی خاکستری رنگ مختلف از سیاه مطلق تا سفید مطلق داریم. (با توجه به این که عکس هایی که امروزه مورد استفاده قرار میگیرند معمولاً 8 بیتی هستند بنابراین این سطوح روشنایی را باید به بازه ی [0, 255] برد).
- PPM: فرمت Header این فایل ها دقیقاً مانند فرمت PGM است و صرفاً در قسمت بعد از Header که محتوای فایل را مشخص می کند به ازای هر پیکسل سه عدد متوالی وجود خواهد داشت که به ترتیب مقادیر R، G و B را برای آن پیکسل مشخص می کنند.

سوال 4) نوشتن یک PPM-reader بدون استفاده از تابع imread:

سورس کد مربوط: Deliverables\ReadPPM.m

کد:

ReadPPM.m

```
function imshowable = ReadPPM(ppm_dir)
    % Reads a PPM file and returns an imshow-able array.
    % read file in bytes
    handle = fopen(ppm_dir);
    % the headers are in ASCII. ['P3' width height max_lumin]
    headers = textscan(handle, '%s %d %d %d', 1);
    if ~strcmp(char(headers{1}), 'P6') % incorrect magic number
        fclose(handle);
        error(['Input file is not in PPM format.\n' ...
            'Please refer to https://en.wikipedia.org/wiki/Netpbm for more info.'])
    else
        [C, R, max_lumin] = headers{2:4};
        % cast max_lumin to double to not clip off anything until after
        % multiplication. lumin_ratio used to scale [0, max_lumin] to [0, 255].
        lumin_ratio = 255/double(max_lumin);
        imshowable = zeros([R, C, 3]);
        rest = fread(handle);
        pos = 1;
        for i = 1:R % ROWS
            for j = 1:C % COLUMNS
                % pos = (i-1)*C*3 + (j-1)*3 + 1;
                % (rest[x], rest[x+1], rest[x+2]) = (G, R, B) -> fix mapping
                imshowable(i,j, :) = ...
                    [rest(pos+1), rest(pos+2), rest(pos)]*lumin_ratio;
                pos = pos + 3;
            end
        end
        % cast to uint8 to round doubles (necessary for imshow())
        imshowable = uint8(imshowable);
        fclose(handle);
    end
end
```

عملیات لازم برای خواندن یک فایل PPM در دو مرحله خلاصه می شود:

1. خواندن Header: برای این کار و با توجه به این که فرمت Header طبق استاندارد تعریف شده است و فرض آن که در فایل PPM ذخیره شده کامنت وجود ندارد (خطوطی که با علامت # شروع می شوند)، می توان از تابع textscan برای خواندن یک رشته و 3 عدد بعد از آن به صورت دسیمال استفاده کرد. نکته ی حائز اهمیت در این قسمت آن است فقط به علت آن که می دانیم Header ها به صورت ASCII ذخیره می شوند می توان این اعداد را با استفاده از تابع textscan استخراج کرد.

2. پس از استخراج Header و بدست آوردن ابعاد تصویر به همراه maximum value نوبت به بازسازی تصویر می رسد.

برای بازسازی ابتدا یک ماتریس سه بعدی با ابعاد تصویر و 3 کانال RGB ایجاد می کنیم. سپس با گام 3 بایت از فایل موجود می خوانیم با علم به آن که هر بایت متناظر با یکی از رنگ های R، G و B یک پیکسل است، مقادیر کانال های آن پیکسل را پر می کنیم و در نهایت اعداد آن پیکسل را در lumin_ratio ضرب می کنیم که صرفاً بازه ی [0, maximum value] را به بازه ی [0, 255] می برد.

نکته ای که باید در نظر داشت این است که ماتریس نهایی قبل از این که بتواند توسط تابع imshow نمایش داده شود باید به uint8، cast شود و نباید مقادیر آن از نوع double باشند.

سوال 5) پیاده سازی Unary Encoding و ارائه ی یک روش برای بهبود آن:

سورس کد های مربوط: Deliverables\UnaryCodec.m و Deliverables\NaiveUnaryCodec.m

هر دو Codec به صورت یک class در متلب تعریف شده اند که شی آن ها باید توسط یک Constructor و با آرگومان ورودی مسیر فایل مورد نظر برای encode یا decode ساخته شود. از آنجایی که فایل ها به طور کامل کامنت گذاری شده اند، در اینجا صرفاً نکات مهم هر یک را توضیح می دهیم و برای توضیحات بیشتر می توانید به کامنت های درون فایل مراجعه کنید. ابتدا به توضیح NaiveUnaryCodec (بدون بهینه سازی) می پردازیم.

NaiveUnaryCodec::encode:

```
function [enc, cr, et, cdsiz, tsiz, dsiz] = encode(obj, save_to)
% Encode file in a naive manner.
if nargin > 0
    tic;
    img = imread(obj.input_dir);
    [M, N] = size(img);
    enc = cell([M*N 1]);
    pos = 1;
    for i = 1:M
        for j = 1:N
            enc{pos} = uint8([ones([img(i, j), 1]); 0]);
            pos = pos + 1;
        end
    end
    enc = cell2mat(enc);
    pad = 8 - mod(numel(enc), 8);
    enc = [enc; ones([pad, 1])];
    cr = (M*N*8) / (8 + numel(enc)/8);
    cdsiz = numel(enc)/(M*N);
    tsiz = (numel(enc)+ 8*8)/(M*N);
    dsiz = 0;
    write = fopen(save_to, 'w');
    fwrite(write, [M; N], 'uint');
    fwrite(write, enc, '*ubit1');
    fclose(write);
    et = toc;
end
end
```

برای شروع، ابتدا فایل عکس را با استفاده از تابع imread می خوانیم و ابعاد آن را با استفاده از تابع size استخراج می کنیم. از آنجایی که تست ها روی تصاویر grayscale انجام شده است، در اینجا نیز فرض کردیم که تصاویر دو بعدی هستند. پس از بارگزاری تصویر، به سادگی می توان آن را encode کرد. کافیهست که دقیقاً به تعداد اندازه ی هر پیکسل 1 بگذاریم و سپس با صفر کد را ببندیم. همچنین این نکته نیز حائز اهمیت است که از آنجایی که ممکن است طول فایل encode شده به این روش به بیت مضرب صحیحی از عدد 8 نباشد، مجبوریم انتهای فایل را تا نزدیک ترین مضرب 8، pad کنیم — در غیر این صورت هنگام استفاده

از تابع `fwrite` (از آنجایی که این تابع `byte` به `byte` در فایل می نویسد)، انتهای فایل دور ریخته خواهد شد. در اینجا از عدد 1 برای `padding` استفاده شده است چرا که متناظر با هیچ کد `valid` ای نیست. در Header فایل ابعاد عکس را به صورت دو عدد 4 بایتی ذخیره می کنیم.

NaiveUnaryCodec::decode:

```
function [dec, et] = decode(obj, save_to)
    % Decode a file that has been encoded naively with unary encoding.
    if nargin > 0
        tic;
        read = fopen(obj.input_dir);
        sz = fread(read, [1 2], 'uint');
        M = sz(1); N = sz(2);
        bits = uint8(fread(read, '*ubit1'));
        fclose(read);
        pad = 0;
        for i = size(bits):-1:1
            if bits(i)
                pad = pad + 1;
            else
                break
            end
        end
        bits = bits(1:size(bits)-pad);
        dec = uint8(zeros([M*N 1]));
        code = 0;
        pos = 1;
        for i = 1:size(bits)
            if bits(i)
                code = code+1;
            else
                dec(pos) = code;
                pos = pos + 1;
                code = 0;
            end
        end
        dec = reshape(dec, [N M]);
        imwrite(dec, save_to);
        et = toc;
    end
end
```

برای انجام عملیات `decoding` عملاً همان تابع `encoding` را برعکس اجرا می کنیم. ابتدا ابعاد عکس را از Header استخراج می کنیم و سپس دیتای موجود در باقی فایل را می خوانیم. پیش از پردازش فایل `padding` اضافه شده در فایل را حذف می کنیم. بدین ترتیب که از انتهای فایل و تا رسیدن به اولین 0 (نمایانگر پایان یک کد Unary)، تمام 1 های ملاقات شده را دور می ریزیم. سپس از ابتدای فایل شروع می کنیم و با خواندن هر 1، یک واحد به مجموع فعلی (در متغیر `code`) می افزایشیم به محض رسیدن به اولین صفر متغیر `code` را در یک پیکسل از عکس قرار می دهیم و اشاره گر به آن پیکسل را یک واحد جلو می بریم. این عملیات تا رسیدن به

انتهای فایل تکرار می شود. در نهایت یک بردار $1 \times (M \times N)$ ای داریم که با استفاده از تابع `reshape` می توانیم به راحتی آن را به ابعاد مورد نظر در بیاوریم و با استفاده از تابع `imwrite` به فرمت دلخواه ذخیره کنیم.

اکنون به بررسی روش پیشنهادی خواهیم پرداخت. ایراد اصلی انجام `unary encoding` به صورت `naive` این است که بدون توجه به این که فراوانی یک مقدار `grayscale` در فایل چه اندازه است و صرفاً بر مبنای این که اندازه ی آن پیکسل چقدر است، کدی به آن اختصاص می یابد. بنابراین برای بهبود این روش می توان به جای آن که کد ها را بر مبنای اندازه ی هر پیکسل به آن اختصاص داد، روشی مبتنی بر فراوانی مقادیر مختلف از 0 تا 255 ارائه کنیم بدین ترتیب که پر تکرار ترین مقدار متناظر با کد 0، دومین پر تکرار ترین مقدار متناظر با کد 10 و همچنین سومین متناظر با کد 110 و الی آخر باشند تا کوتاه ترین کد ها به پر تکرار ترین مقادیر اختصاص یابند و بنابراین از حجم کد نهایی کاسته شود. از آنجایی که عمده ی عملیات `UnaryCodec` پیشنهادی دقیقاً مشابه `NaiveUnaryCodec` است، صرفاً قسمت اصلی متفاوت بین این دو `Codec` یعنی تابع `BuildDict` را توضیح خواهیم داد و توضیحات تکمیلی را به کامنت های موجود در فایل ها موکول می کنیم.

`UnaryCodec::BuildDict:`

```
function obj = BuildDict(obj, mode)
    % Builds a dictionary from an unencoded file or reads the it from the
    % header of an encoded file.
    if nargin > 0
        if mode == "enc"
            img = imread(obj.input_dir);
            [M, N] = size(img);
            h = zeros([256, 1]);
            for i = 1:M
                for j = 1:N
                    h(img(i,j) + 1) = h(img(i, j) + 1) + 1;
                end
            end
            [~, imap] = sort(h, 'descend');
            for i = 1:256
                obj.dict(imap(i)) = i-1;
            end
        elseif mode == "dec"
            read = fopen(obj.input_dir);
            fseek(read, 8, 'bof');
            imap = fread(read, [256, 1]);
            fclose(read);
            for i = 1:256
                obj.dict(imap(i) + 1) = i-1;
            end
        else
            error('BuildDict called with unsupported mode.');
```

همان طور که مشخص است، این تابع در دو حالت “enc” و “dec” می تواند فراخوانی شود. ایده ی اصلی در تشکیل این dictionary آن است که بتوان به دو صورت از آن استفاده کرد:

1. $dict(< code word index >) := actual\ grayscale\ value\ (\in [0, 255])$
2. $dict(< actual\ grayscale\ value >) := code\ word\ index\ (number\ of\ 1s\ before\ 0)$

برای تشکیل dictionary در مرحله ی encoding، پس از خواندن فایل یک histogram با 256 سطل (bin) از تصویر خوانده شده ایجاد می کنیم که بسامد هر یک از مقادیر بین 0 تا 255 را در فایل نشان می دهد. دقت کنید که خود این مقادیر اهمیتی ندارند و نکته ی مورد نظر ما صرفاً ترتیب این فراوانی هاست. برای حفظ این ترتیب از خروجی دوم تابع sort متلب استفاده می کنیم که یک آرایه ی imap برای ما بر می گرداند. در این آرایه هر اندیس i متناظر با محل i-امین پر تکرار ترین مقدار ممکن است و از آنجایی که ما در هنگام encoding به عکس این نگاشت احتیاج داریم، آن را برعکس می کنیم (جای اندیس ها و مقادیر را عوض می کنیم) و آن را در dictionary شی ایجاد شده از کلاس ذخیره می کنیم. اکنون در حین انجام encoding هر گاه به پیکسلی با مقدار grayscale برابر با val رسیدیم برای این که بفهمیم برای Unary Code این مقدار باید از چند 1 منتهی به صفر استفاده کنیم صرفاً از $obj.dict(val + 1)$ استفاده می کنیم (به علاوه 1 به علت 1-indexed بودن آرایه ها در متلب است).

بر عکس در قسمت decoding به عکس این dictionary احتیاج داریم یعنی می خواهیم بدانیم که مقدار واقعی متناظر با هر کد unary چقدر است. بنابراین بعد از skip کردن 8 بایت از ابتدای فایل (رزره شده برای نوشتن ابعاد تصویر)، 256 بایت بعدی که شامل dictionary هستند را با علم به این که مقادیر آن مرتب شده هستند می خوانیم و سپس نگاشت از اندیس ها به مقادیر را برعکس می کنیم. اکنون بعد از خواندن یک code word برای آن که بدانیم در تصویر بازسازی شده باید چه مقداری به جای این کد قرار دهیم به راحتی می توانیم از $obj.dict(codeword + 1)$ استفاده کنیم.

ضمناً برای محاسبه ی Entropy و Coding Redundancy از تابع ent در فایل ent.m استفاده شده است که به طور کامل در کلاس توضیح داده شده بود. نتایج نهایی را می توانید در جدول صفحه ی بعد مشاهده کنید.

Image Name		Goldhill	Baboon	Boat
Entropy		7.530255	7.373764	7.220912
Coding Redundancy		0.469745	0.626236	0.779088
Unary Code (Naïve)	Compressed Data Size (BPP)	100.8622	130.6002	122.1033
	Dictionary Size (BPP)	0	0	0
	Total Size (BPP)	100.8624	130.6005	122.1034
	Compression Ratio	0.0793	0.0613	0.0655
	Encoding Time	2.9447	2.1860	3.3798
	Decoding Time	7.0793	6.3457	9.5519
	PSNR	Inf	Inf	Inf
Proposed Method	Compressed Data Size (BPP)	75.7300	67.0645	57.0442
	Dictionary Size (BPP)	0.0049	0.0078	0.0045
	Total Size (BPP)	75.7351	67.0726	57.0488
	Compression Ratio	0.1056	0.1193	0.1402
	Encoding Time	2.7781	1.7227	2.8604
	Decoding Time	5.2974	3.0762	4.4099
	PSNR	Inf	Inf	Inf

همانطور که به وضوح از نتایج مشخص است روش پیشنهادی می تواند در حدود 42.3٪ (بر اساس معیار Total Size – عدد بدست آمده حاصل میانگین بدون میزان Improvement روی هر 3 فایل کد شده است) از روش معمولی بهتر عمل کند و این در حالیست که به دلیل کاهش حجم کد، زمان encoding و decoding نیز کاهش پیدا می کنند. (حدود 14.1٪ صرفه جویی در زمان encoding و حدود 43.5٪ صرفه جویی در زمان decoding)