COMPENG 4TN4 - Project 1

Image Compression by Down and Up Sampling

Instructor: Dr. Xiaolin Wu

Glen Tsui – tsuig – 400201284

As a future member of the engineering profession, the student is responsible for performing the required work in an honest manner, without plagiarism and cheating. Submitting this work with my name and student number is a statement and understanding that this work is my own and adheres to the Academic Integrity Policy of McMaster University and the Code of Conduct of the Professional Engineers of Ontario. Submitted by **[Glen Tsui, tsuig, 400201284]**

Compression Results per step:
Scale: 1

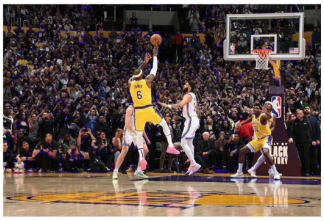
Scale: 1

RGB Original (750 x 500)

YCbCr Transformed (750 x 500)

YCbCr Downsampled (750 x 500)

YCbCr Upsampled (750 x 500)



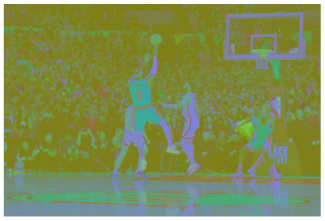
RGB Transformed (750 x 500)

PSNR: 43.33467177732578

Scale: 2

Scale: 2

RGB Original (750 x 500)



YCbCr Transformed (750 x 500)

YCbCr Downsampled (375 x 250)

YCbCr Upsampled (750 x 500)

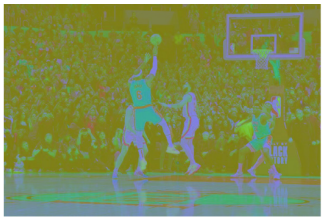
RGB Transformed (750 x 500)

PSNR: 30.340594305066983

Scale: 5

Scale: 5

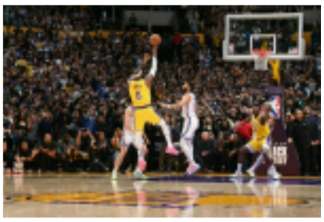
RGB Original (750 x 500)



YCbCr Transformed (750 x 500)

YCbCr Downsampled (150 x 100)

YCbCr Upsampled (750 x 500)



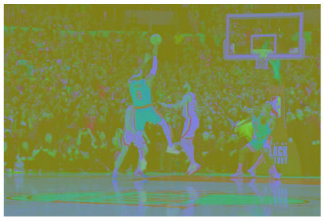
RGB Transformed (750 x 500)

PSNR: 29.019631342148

Scale: 10

Scale: 10

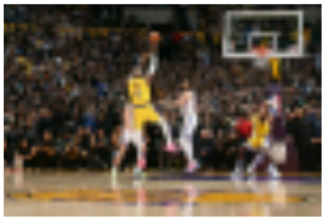
RGB Original (750 x 500)



YCbCr Transformed (750 x 500)

YCbCr Downsampled (75 x 50)

YCbCr Upsampled (750 x 500)



RGB Transformed (750 x 500)

PSNR: 28.54824985648682

Code Review

```python
#function for the colour transform from RGB to YCbCr space
def RGBToYCbCr_transform(image):
    #read the image planes into RGB values
    R, G, B = image[:,:,0], image[:,:,1], image[:,:,2]
    #apply conversion equations from RGB to YCbCr
    Y = 16 + (65.738*R/256)+(129.057*G/256)+(25.064*B/256)
    Cb = 128 - (37.945*R/256) - (74.494*G/256) + (112.439*B/256)
    Cr = 128 + (112.439*R/256) - (94.154*G/256) - (18.285*B/256)
    #stack 8-bit arrays in 3rd dimension (plane-wise)
    return np.stack([Y, Cb, Cr], axis=2).astype(np.uint8)
```

$$\begin{cases} Y = 16 + \frac{65.738R}{256} + \frac{129.057G}{256} + \frac{25.064B}{256} \\ Cb = 128 - \frac{37.945R}{256} - \frac{74.494G}{256} + \frac{112.439B}{256} \\ Cr = 128 + \frac{112.439R}{256} - \frac{94.154G}{256} - \frac{18.285B}{256} \end{cases}$$

Conversion to YCbCr colour space was done for more efficient downsampling. The above equation was used to convert the RGB channels to YCbCr.

```python
#function for the colour transform from YCbCr to RGB
def YCbCrtoRGB_transform(image):
    #read the image planes into YCbCr values
    Y, Cb, Cr = image[:,:,0], image[:,:,1], image[:,:,2]
    #apply conversion equations from YCbCr to RGB
    R = (298.082*Y/256) + (408.583*Cr/256) - 222.921
    G = (298.082*Y/256) - (100.291*Cb/256) - (208.120*Cr/256) + 135.576
    B = (298.082*Y/256) + (516.412* Cb/256) - 276.836
    #stack 8-bit arrays in 3rd dimension (plane-wise)
    return np.stack([R, G, B], axis=2).astype(np.uint8)
```

$$\begin{aligned} R'_D &= \frac{298.082 \cdot Y'}{256} + \frac{408.583 \cdot C_R}{256} - 222.921 \\ G'_D &= \frac{298.082 \cdot Y'}{256} - \frac{100.291 \cdot C_B}{256} - \frac{208.120 \cdot C_R}{256} + 135.576 \\ B'_D &= \frac{298.082 \cdot Y'}{256} + \frac{516.412 \cdot C_B}{256} - 276.836 \end{aligned}$$

Conversion from YCbCr to RGB colour space was achieved using the equation above.

```python
#function to compute the PSNR value
def calcPSNR(before_image, after_image):
    #calculate the mean squared value of the difference (error)
    mse = np.mean((before_image - after_image) ** 2)
    #find the Peak-Signal-to-Noise-Ratio
    psnr = 10 * np.log10(255**2 / mse)
    return psnr
```

$$PSNR = 10\log_{10}\left(\frac{R^2}{MSE}\right) \quad MSE = \frac{\sum_{M,N}[I_1(m,n) - I_2(m,n)]^2}{M * N}$$

The calculation of the PSNR value was achieved with the above equation where R is 255 bits representing the 8-bit unsigned integer data type of the image. In order to compute the PSNR, the MSE value was calculated by finding the mean of the squared difference between each image pixel.

```python
#function to downsample input by a scale factor
def downsample(image, scale_factor):
    #extract height and width values for the image
    height, width = len(image),len(image[0])
    #initialize colour spaces for the image
    Y = image[:,:,0]
    Cb = image[:, :, 1]
    Cr = image[:, :, 2]
    #initialize output colour spaces by calculated height and width after downsample
    downsampled_Y = np.zeros((height//scale_factor, width//scale_factor))
    downsampled_Cb = np.zeros((height//scale_factor, width//scale_factor))
    downsampled_Cr = np.zeros((height//scale_factor, width//scale_factor))
    #loop to end of dimensions, skip by every factor value
    for i in range(0, height, scale_factor):
        for j in range(0, width, scale_factor):
            #sliding window based on the scale factor for each plane
            window_Y = Y[i:i+scale_factor, j:j+scale_factor]
            window_Cb = Cb[i:i+scale_factor, j:j+scale_factor]
            window_Cr = Cr[i:i+scale_factor, j:j+scale_factor]
            #average the values within each area
            avg_Y = np.mean(window_Y)
            avg_Cb = np.mean(window_Cb)
            avg_Cr = np.mean(window_Cr)
            #plot the 8-bit values within a scaled down area
            downsampled_Y[i//scale_factor, j//scale_factor] = avg_Y.astype(np.uint8)
            downsampled_Cb[i//scale_factor, j//scale_factor] = avg_Cb.astype(np.uint8)
            downsampled_Cr[i//scale_factor, j//scale_factor] = avg_Cr.astype(np.uint8)
    #stack 8-bit arrays in 3rd dimension (plane-wise)
    return np.stack([downsampled_Y,downsampled_Cb,downsampled_Cr], axis=2).astype(np.uint8)
```

The downsampling algorithm was achieved by separating the YCbCr channels and utilizing a sliding window with size based on the scale factor to compute the average value at that coordinate. Each coordinate that holds the average value in the final downsampled image is incremented in terms of the scale factor to map out the 8-bit output within the correct dimensions.

```python
#function to perform bilinear interpolation for upsampling
def bilinear_interpolation(image, scale_factor):
    #extract height and width values for the image
    height, width = len(image),len(image[0])
    #initialize output height and width based on factor
    new_height, new_width = int(height * scale_factor), int(width * scale_factor)
    #initialize output image based on upsamplign dimensions
    new_image = np.zeros((new_height, new_width, 3), dtype=np.uint8)
    #map out coordinates of each pixel by dividing by scale factor
    x = np.zeros(new_width)
    for i in range(new_width):
        x[i] = i / scale_factor
    # coordinates of new image pixels
    y = np.zeros(new_height)
    for i in range(new_height):
        y[i] = i / scale_factor
    #closest points to four points
    y_floor = np.floor(y).astype(np.int32)
    x_floor = np.floor(x).astype(np.int32)
    y_ceil = y_floor + 1
    y_ceil[y_ceil > height - 1] = height - 1
    x_ceil = x_floor + 1
    x_ceil[x_ceil > width - 1] = width - 1
    #compute difference values for equation
    dy = y - y_floor
    dx = x - x_floor
    #loop through newly initialized dimensions
    for i in range(new_height):
        for j in range(new_width):
            #define four pixel values to estimate upsample
            q11 = image[y_floor[i], x_floor[j]]
            q12 = image[y_floor[i], x_ceil[j]]
            q21 = image[y_ceil[i], x_floor[j]]
            q22 = image[y_ceil[i], x_ceil[j]]
            #apply bilinear interpolation formula
            new_image[i, j] = (
                (q11*(1 - dx[j])*(1 - dy[i])) + (q12*dx[j]*(1 - dy[i])) +  (q21*(1 - dx[j])*dy[i]) + (q22*dx[j]*dy[i])
            )
    #output uint8 upsampled image
    return new_image.astype(np.uint8)
```

$$f(x,y) \approx f(0,0)(1-x)(1-y) + f(1,0)x(1-y) + f(0,1)(1-x)y + f(1,1)xy$$

The bilinear interpolation method was used for the upsampling process, which focuses on computing the interpolated value at each coordinate based on the weighted average of the attributes of the surrounding four points. This function was achieved similarly to downsampling where it utilizes the scaling factor to initialize the process and output values, and applying the interpolation formula to find the value of the output pixel. These pixels are mapped in 8-bits within the new dimensions with the scaling factor taken into consideration.

```python
#define downsampling and upsampling scale
scale = 10
#load input image
original_image = cv.imread('LebronJames.jpg')
#transform from RGB to YCbCr space
ycbcr_image = RGBToYCbCr_transform(original_image)
cv.imwrite("ycbcrLebronJames.jpg", ycbcr_image)
#downsample image by defined factor
downsample_ycbcr_image = downsample(ycbcr_image, scale)
cv.imwrite("downsample_ycbcrLebronJames.jpg", downsample_ycbcr_image)
#upsample image by defined factor with bilinear interpolation
upsample_ycbcr_image = bilinear_interpolation(downsample_ycbcr_image,scale)
cv.imwrite("upsample_ycbcrLebronJames.jpg", upsample_ycbcr_image)
#transform from YCbCr to RGB space
compressed_image = YCbCrtoRGB_transform(upsample_ycbcr_image)
cv.imwrite("newLebronJames.jpg", compressed_image)
#calculate PSNR value
psnr = calcPSNR(original_image, compressed_image)
print('PSNR:',psnr)
```

The program was designed to output each stage of the process leading up to the final compressed image. Downsampling and upsampling processes were utilized and the pixel values were quantized to 8-bits. The YCbCr colour space was chosen for more effective downsampling, and both forward and reverse colour transform methods were implemented. Built-in functions within Python were not used, and the program outputs the PSNR value of the reconstructed image.