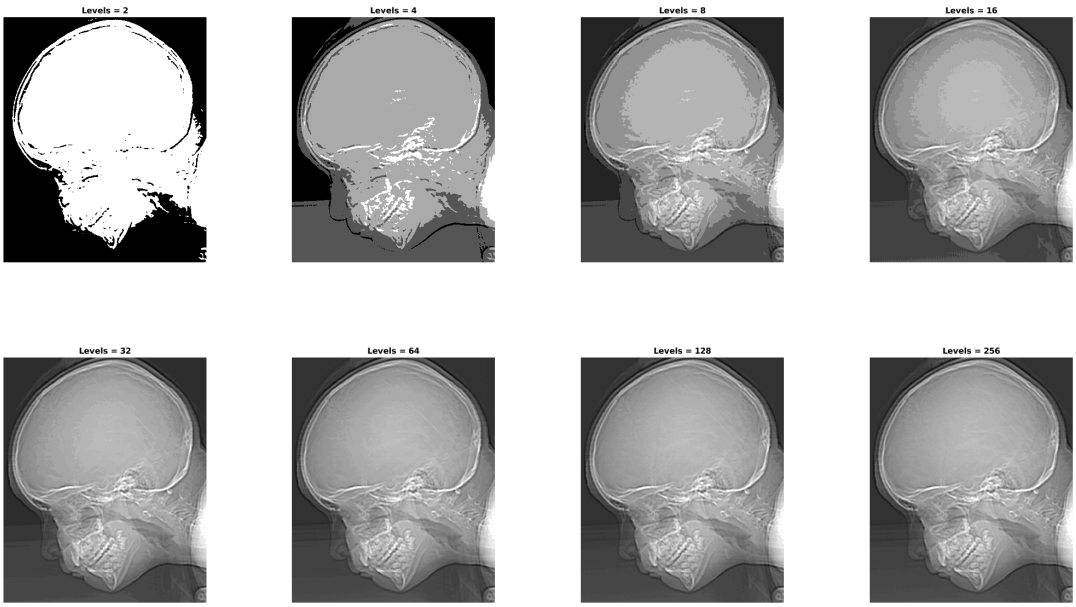
Image Processing Lab1

Name: 徐竣霆

Student ID: 110060012

Proj02-02: Reducing the Number of Intensity Levels in an Image

Result

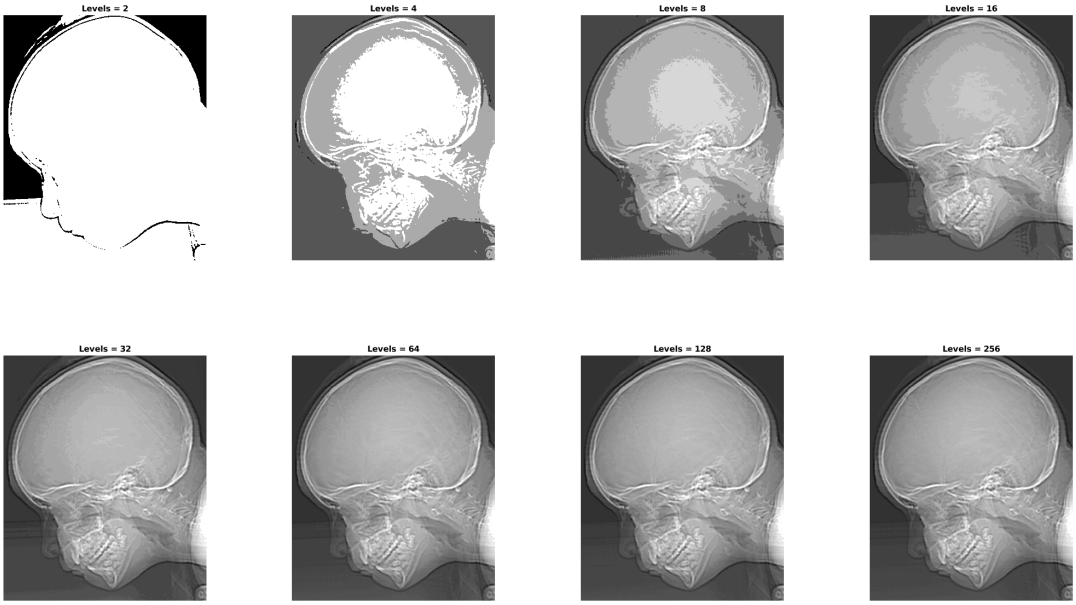


Implementation

```
function quantizedImage = reduceIntensityLevel(originalImage, intensityLevel)
    % Check if the intensity level is within the range [2, 256]
    if intensityLevel < 2 || intensityLevel > 256
        error('Intensity level must be within the range [2, 256]');
    end

    % Convert the original image to double precision
    originalImage = double(originalImage);

    % Compute the scaling factor
    scalingFactor = 255 / (intensityLevel - 1);

    % Compute the step size for quantization
    stepSize = floor(256 / intensityLevel);

    quantizedImage = uint8(floor(originalImage / stepSize) * scalingFactor);
    end
```

For example, if $intensityLevel = 4$, which means we have to set $stepSize = \lfloor 256/intensityLevel \rfloor = 64$, and set $scalingFactor = 255/(intensityLevel - 1) = 85$ to make the biggest contrast between colors(i.e. $[0, 85, 170, 255]$ represents quotients $[0, 1, 2, 3]$).

There is something interesting, if we delete 'originalImage = double(originalImage);', the result will be quite differnt.



It's due to the precision scales(and it takes me hours to fix it).

Discussion

None

Proj02-03: Zooming and Shrinking Images by Pixel Replication

Result

Original

Original Image

Shrink by 10 and then Zoom by 10

Zoom by 10 Image

Implementation

```
function resizedImage = resizeImage_replication(originalImage, scalingFactor)
    % Get the dimensions of the original image
    [height, width, channels] = size(originalImage);

    % Calculate the dimensions of the new image
    newHeight = round(height * scalingFactor);
    newWidth = round(width * scalingFactor);

    % Create a new image of the calculated dimensions
    resizedImage = zeros(newHeight, newWidth, channels, 'uint8');

    % Iterate over each pixel in the new image, doing nearest neighbor interpolat
    for x = 1:newHeight
        for y = 1:newWidth
            original_x = min(round((x - 1) * (double(height - 1) / (newHeight - 1
            original_y = min(round((y - 1) * (double(width - 1) / (newWidth - 1)))
            resizedImage(x, y, :) = originalImage(original_x, original_y, :);
        end
    end
```

As taught in slide, enumerate every pixels, use the value of nearest neighbors, which is achieved by round(), use min() to deal with the boundary cases.

Discussion

Pixel Replication(aka. Nearest-Neighbor Interpolation) doing downscaling will make some pixels(information) lost. While doing upscaling back to the original size, these pixels(information) cannot be restored, we just use nearest pixels to simulate them. This causes the phenomenon of "resolution degradation".

Proj02-04: Zooming and Shrinking Images by Bilinear Interpolation

Result

Original

Original Image

Shrink by 12.5 and then Zoom by 12.5

Zoom by 12.5 Image

Implementation

```
function resizedImage = resizeImage_bilinear(originalImage, scalingFactor)
    % Get the size of the original image
    [height, width, channels] = size(originalImage);

    % Calculate the size of the new image
    newHeight = round(height * scalingFactor);
    newWidth = round(width * scalingFactor);

    % Initialize the resized image
    resizedImage = zeros(newHeight, newWidth, channels, 'like', originalImage);

    % Loop over each color channel (R, G, B) or single channel for grayscale
    for c = 1:channels
        for x = 1:newHeight
            for y = 1:newWidth
                % Get the new pixel coordinates back to the original image
                X = (x - 1) * (double(height - 1) / (newHeight - 1)) + 1;
                Y = (y - 1) * (double(width - 1) / (newWidth - 1)) + 1;

                % Get the surrounding pixel coordinates in the original image
                x1 = min(floor(X), height);
                x2 = min(ceil(X), height);
                y1 = min(floor(Y), width);
                y2 = min(ceil(Y), width);

                % Calculate the distances between the original and new pixel
                dx = X - x1;
                dy = Y - y1;

                % Perform bilinear interpolation
                pixelValue = (1 - dy) * (1 - dx) * double(originalImage(x1, y1, c
                             dy * (1 - dx) * double(originalImage(x1, y2, c)) +
                             (1 - dy) * dx * double(originalImage(x2, y1, c)) +
                             dy * dx * double(originalImage(x2, y2, c));

                % Assign the interpolated value to the new image
                resizedImage(x, y, c) = uint8(pixelValue);
            end
        end
    end
end
```

Based on slides(p26), derive the formula, enumerate for each pixel. Notice that double() was used to deal with the precision. The $(x1, y1), (x2, y2)$ correspond to the 4 neighboring grids, use min() to deal with boundaries.

Discussion

Again, downscaling leads to information lost, some pixels will be replaced by weighted average of 4 neiboring pixels. Upscaling try to simulate the original information, while it's not perfect, inaccuracies remain.

(bonus)Proj02-04: Zooming and Shrinking Images by Bicubic Interpolation

---

Result

Original

Original Image

Shrink by 12.5 and then Zoom by 12.5

Zoom by 12.5 Image

Implementation

```
function resizedImage = resizeImage_bicubic(originalImage, scalingFactor)
    % Get the size of the original image
    [height, width, channels] = size(originalImage);

    % Calculate the size of the new image
    newHeight = round(height * scalingFactor);
    newWidth = round(width * scalingFactor);

    % Initialize the resized image
    resizedImage = zeros(newHeight, newWidth, channels, 'like', originalImage);

    % Bicubic interpolation function
    function w = cubicWeight(t)
        % Based on Wikipedia: https://en.wikipedia.org/wiki/Bicubic_interpolatior
        a = -0.5;  % Common choice
        absT = abs(t);

        if absT <= 1
            w = (a + 2) * (absT^3) - (a + 3) * (absT^2) + 1;
        elseif absT <= 2
            w = a * (absT^3) - 5 * a * (absT^2) + 8 * a * absT - 4 * a;
        else
            w = 0;
        end
    end

    % Loop over each color channel (R, G, B) or single channel for grayscale
    for c = 1:channels
        for x = 1:newHeight
            for y = 1:newWidth
                % Get the new pixel coordinates back to the original image
                X = (x - 1) * (double(height - 1) / (newHeight - 1)) + 1;
                Y = (y - 1) * (double(width - 1) / (newWidth - 1)) + 1;

                % Get the surrounding pixel coordinates in the original image
                x1 = min(floor(X), height);
                y1 = min(floor(Y), width);

                % Calculate the distances between the original and new pixel
                dx = X - x1;
                dy = Y - y1;

                % Initialize the interpolated value
                pixelValue = 0;

                % Perform bicubic interpolation on 16 surrounding pixels
                for m = -1:2  % Loop over a 4x4 grid of pixels
                    for n = -1:2
                        % Get the pixel position in the original image
                        xm = min(max(x1 + m, 1), height);
                        yn = min(max(y1 + n, 1), width);

                        % Calculate the cubic weights
                        wx = cubicWeight(m - dx);
```

```
                    wy = cubicWeight(n - dy);

                    % Aggregate the weighted sum
                    pixelValue = pixelValue + double(originalImage(xm, yn, c)
                end
            end

            % Assign the computed value to the resized image
            resizedImage(x, y, c) = uint8(min(max(pixelValue, 0), 255));
        end
    end
end
end
```

The Weight Function is looked up from wikipedia.

Bicubic convolution algorithm [edit]

Bicubic spline interpolation requires the solution of the linear system described above for each grid cell. An interpolator with similar properties can be obtained by applying a convolution with the following kernel in both dimensions:

$$W(x) = \begin{cases} (a+2)|x|^3 - (a+3)|x|^2 + 1 & \text{for } |x| \le 1, \\ a|x|^3 - 5a|x|^2 + 8a|x| - 4a & \text{for } 1 < |x| < 2, \\ 0 & \text{otherwise,} \end{cases}$$

where a is usually set to −0.5 or −0.75. Note that $W(0) = 1$ and $W(n) = 0$ for all nonzero integers n

Based on slides(p28), derive the formula, enumerate for each pixel. Notice that double() was used to deal with the precision. It's a bit more complicated than bilinear, since we have 16 grids to concern, use nested for and min() to deal with boundaries.

Discussion

Since the information alreast lost, I don't see trying to involve more neighboring pixels(16 in this case) can do any better. Different methods of interpolations are just different ways of simulation, the performances strongly depend on the properties and characterists of the original image itself.