Image Compression by Down and Up Sampling

Abd Elelah Arafah
McMaster University
1280 Main St W, Hamilton, ON L8S 4L8
arafaha@mcmaster.ca

Abstract

Image compression is a very important technique which is widely used in computer vision and other applications. In this paper, we propose the use of a simple average pooling kernel for down sampling an image, and then we use bilinear interpolation for up sampling the compressed image. In this paper, we will also show the results and explain why such a method is useful for certain image compression applications.

1. Introduction

Images are a widely used technology and it is an important part of modern life as it has a wide range of applications, such as: social media and medical imaging. With that being said, we need the use of different image processing techniques to allow us to perform many necessary functions. One of which is image compression which will be explored in this paper.

1.1. Overview of Image Compression

There are two different types of image compression methods: lossy and lossless. Lossless techniques result in a very accurate representation of the input image, while lossy compression techniques produce a degraded version of the input image.

1.2. Proposed Method

In this paper, we will discuss a simple lossy method which means, the result of the compression will be a degraded image of the input image. The specific techniques which we will explore in this paper are using average pooling for down-sampling and bilinear interpolation as well as n nearest neighbors for up-sampling. In this case we can consider the input image and the down sampling to be performed at the transmitter side of the system, while the up sampling and the output image at the receiver side of the system.

2. Image Preprocessing

Before performing down-sampling to the image, we will first convert the RGB color space of the input image into the YUV color space.

2.1. Conversion into YUV color space

The following formulas are used to convert from RGB to YUV:

$$R = Y + 1.13983V \quad (1)$$

$$G = Y - 0.39465U - 0.5060V \quad (2)$$

$$B = Y - 2.03211U \quad (3)$$

Using the above we can implement a function in python to convert each of the R, G and B channels into YUV channels [1].

```
# Define the conversion matrices for RGB to YUV and YUV to RGB
RGB_to_YUV_conversion_matrix = np.array([[0.299, 0.587, 0.114], [0.5, -0.41869, -0.08131], [-0.16874, -0.33126, 0.5]])
YUV_to_RGB_conversion_matrix = np.array([[1.000, 0.000, 1.13983], [1.000, -0.39465, -0.58060], [1.000, 2.03211, 0.000]])
# Function which converts from RGB to YUV color space
def RGB2YUV_fxn(input_img):
    yuv_img = np.zeros_like(input_img)
    yuv_img[:,:,2] = np.dot(input_img, RGB_to_YUV_conversion_matrix[2])
    yuv_img[:,:,1] = np.dot(input_img, RGB_to_YUV_conversion_matrix[1])
    yuv_img[:,:,0] = np.dot(input_img, RGB_to_YUV_conversion_matrix[0])
    temp_var = yuv_img[:,:,1:] + 128.0
    # 8-bit representation
    yuv_img[:,:,1:] = temp_var.astype(np.uint8)
    return yuv_img
```

Figure 1. The sample python code which implements the RGB to YUV conversion.

(a) Y channel (b) U channel (c) V channel

Figure 2. The results of the RGB to YUV conversion

Note that images in Figure 2 are the same size as the input images, however they are resized in this report. Please refer to the submission package for the correct output sizing.

(a) Y channel (b) U channel (c) V channel

Figure 3: The results of the average pooling down-sample

2.2. Benefits of YUV

The reason for utilizing the YUV color space is that most information is stored in the Y channel as seen in Figure 2. In that way, when down-sampling, we can apply different factors for the Y, U and V channels so that we retain much more information on the Y channel, while compressing U and V with higher factors [2].

3. Down Sampling using Average Pooling

For down sampling the input image we will use a simple average pooling kernel method.

3.1. Average pooling

The way average pooling works in down sampling an image is that the method uses a kernel sliding window which averages out each window into a single pixel value.

3.2. Implementation details of Average pooling

To achieve average pooling for an image. We can develop a sample python code which executes this technique.

```python
def downsample_YUV_image_fxn(image):
    # Get the YUV channels
    Y, U, V = split_channels_manual_fxn(image)

    # Downsample the Y channel by a factor of 2
    Y_downsampled_channel = downsample_avg_fxn(Y, 2)

    # Downsample the U and V channels by a factor of 4
    U_downsampled_channel = downsample_avg_fxn(U, 4)
    V_downsampled_channel = downsample_avg_fxn(V, 4)

    return Y_downsampled_channel, U_downsampled_channel, V_downsampled_channel

# implementation of a simple average pooling kernel method
def downsample_avg_fxn(color_channel, downsampling_factor):
    # Define the downsampling kernel
    kernel = np.ones((downsampling_factor, downsampling_factor)) / (downsampling_factor * downsampling_factor)

    # We can Pad the channel
    height, width = color_channel.shape
    pad_H = height + downsampling_factor - (height % downsampling_factor)
    pad_W = width + downsampling_factor - (width % downsampling_factor)
    pad_ch = np.zeros((pad_H, pad_W), dtype=color_channel.dtype)
    pad_ch[:height, :width] = color_channel

    # Apply the downsampling kernel to the padded channel
    downsampled_ch = np.zeros((pad_H // downsampling_factor, pad_W // downsampling_factor), dtype=color_channel.dtype)
    for i in range(0, pad_H, downsampling_factor):
        for j in range(0, pad_W, downsampling_factor):
            kernel_window = pad_ch[i:i+downsampling_factor, j:j+downsampling_factor]
            downsampled_ch[i//downsampling_factor, j//downsampling_factor] = manual_sum_fxn(kernel * kernel_window)

    # Remove the padding
    downsampled_ch = downsampled_ch[:height//downsampling_factor, :width//downsampling_factor]

    return downsampled_ch
```

Figure 4. The sample python code which implements the average pooling.

The way the implementation is executed is that we first define a kernel window using the factor of which the down sampling is applied. In this case a factor of 4 for U and V channels, and factor of 2 for Y channel. After that the algorithm simply loops through the entire image, averaging out each window to one pixel. As seen in Figure 3, the image for Y is 2 times smaller than the input image for the Y channel, while U and V channels are 4 times smaller than the input image for these channels.

4. Up Sampling using Bilinear Interpolation

For up sampling the image at the decoder we will use a simple Bilinear Interpolation method.

4.1. Bilinear Interpolation

The way bilinear interpolation works in up sampling is that it upscales a give image by taking a weighted average of the four closest pixels in the original image, bilinear interpolation estimates the value of a new pixel. The weights are determined by the distance between the new pixel and each of its neighbors.

4.2. Implementation details Bilinear Interpolation

For the implementation of bilinear interpolation we can develop the sample python code which takes an image channel (For example, Y channel) as well as the size of the original input image. Then it outputs the upscaled version of the input channel.

$$f(x,y) = \left(\frac{y2-y}{y2-y1}\right) f(x,y1) + \left(\frac{y-y2}{y2-y1}\right) f(x,y2) \quad (4)$$

Using equation (4) we can develop a sample python algorithm which loops through the image and performs calculations for pixels using 4 neighboring pixels [3].

```python
# Implementing Bilinear interpolation to upsize an image channel
def bilin_upsample_fxn(image_ch, size):
    # Determine the size of the starting image channel
    H, W = image_ch.shape

    # Determine the size of the output image from the input image size
    new_H, new_W = size

    # Create an array to store the output image channel
    output_ch = np.zeros((new_H, new_W))

    # Computation of the coordinates as image pixels
    x_px_coord = float(W - 1) / (new_W - 1)
    y_px_coord = float(H - 1) / (new_H - 1)

    # Perform bilinear interpolation to compute the output image
    for i in range(new_H):
        for j in range(new_W):
            compute_x = x_px_coord * j
            compute_y = y_px_coord * i
            x_1 = int(compute_x)
            y_1 = int(compute_y)
            x_2 = min(x_1 + 1, W - 1)
            y_2 = min(y_1 + 1, H - 1)
            dx = compute_x - x_1
            dy = compute_y - y_1
            output_ch[i, j] = (1 - dx) * (1 - dy) * image_ch[y_1, x_1]
            + dx * (1 - dy) * image_ch[y_1, x_2] + (1 - dx) * dy * image_ch[y_2, x_1] + dx * dy * image_ch[y_2, x_2]
    return output_ch
```

Figure 5. The sample python code which implements the bilinear interpolation for an image channel.

The output images resulting from running the code from Figure 5, appear to be a degraded version of the YUV channel images from Figure 2 and are also the same size dimensions.

(a) Y channel (b) U channel (c) V channel

Figure 7. The results of the bilinear interpolation

Note that images in Figure 8 are the same size as the input images, however they are resized in this report. Please refer to the submission package for the correct output sizing.

5. Up Sampling using N Nearest Neighbors

We will also explore another method of up sampling using n nearest neighbors.

5.1. N Nearest Neighbors

The n nearest neighbors algorithm works in a similar manner to bilinear interpolation. This method is regarded under machine learning algorithms and the advantage is has over bilinear interpolation is that we can specify the number of neighbors. However, by the nature of the algorithm we expect that edges might be more blurry and pixelated as compared to the previous method. With that being said, considering both algorithms gives us a wider range of applications and images of which we can process as each of the considered algorithms is beneficial for different applications.

5.2. Implementation details of N Nearest Neighbors

This algorithm is also fairly simple. The way it calculates pixels is by considering N near pixels which are located closest to the pixel. This works by converting the image into its pixels representation and then by looping through the image and computing the Euclidian distance between the neighboring pixels. We can implement this algorithm in python as seen by the following figure:

```python
def n_nearest_neighbors_fxn(image_ch, up_sample_factor, size, n_neighbors=4):
    # Determine the size of the starting image channel
    H, W = image_ch.shape
    new_H, new_W = size

    # Create an empty array to hold the upscaled image
    upscale_array = np.zeros((new_H, new_W), dtype=image_ch.dtype)

    # Iterate over each pixel in the upscaled image
    for y in range(new_H):
        for x in range(new_W):
            # Compute pixel
            y_px = int(y / up_sample_factor)
            x_px = int(x / up_sample_factor)

            # Check if the original pixel location is within the bounds of the image
            if y_px >= 0 and y_px < H and x_px >= 0 and x_px < W:
                # Create a subarray of the (n_neighbors) array
                neighborhood = image_ch[max(0, y_px - n_neighbors//2) : min(H, y_px + n_neighbors//2 + 1),
                                        max(0, x_px - n_neighbors//2) : min(W, x_px + n_neighbors//2 + 1)]
                # Calculate the Euclidean distance between the pixel and its neighbors
                euclidean_distances = np.sqrt(np.sum((neighborhood - image_ch[y_px, x_px])**2, axis=None if neighborhood.ndim == 2 else (1, 2)))
                # Choose the pixel which is closest
                nearest_neigh = np.argmin(euclidean_distances)
                nearest_neighbor = neighborhood.flatten()[nearest_neigh]

                # value of the nearest neighbor = the current pixel
                upscale_array[y, x] = nearest_neighbor
    return upscale_array
```

Figure 7. The sample python code which implements the 4 nearest neighbors for an image channel.

The resulting images from this algorithm are also very similar to the YUV channels resulted from the bilinear interpolation algorithm:

(a) Y channel (b) U channel (c) V channel

Figure 8. The results of the n nearest neighbors

Note that images in Figure 8 are the same size as the input images, however they are resized in this report. Please refer to the submission package for the correct output sizing.

6. Reverse color space conversion (YUV to RGB)

After performing up-sampling to the compressed image, we will first convert the YUV color space back into RGB.

6.1. Implementation of YUV to RGB

The following formulas are used to convert from YUV to RGB:

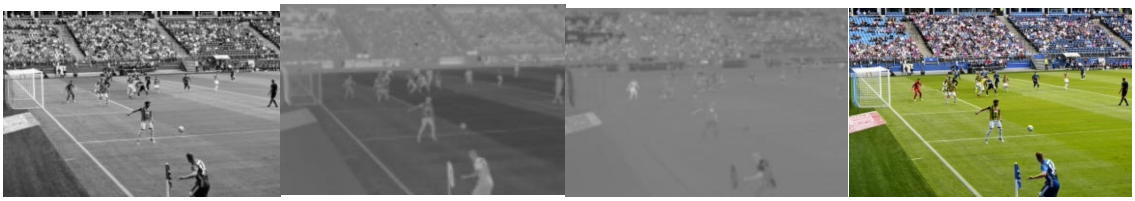
$$Y = 0.299R + 0.587G + 0.144B \quad (5)$$

$$U = -0.14713R - 0.28886G + 0.436B \quad (6)$$

$$V = 0.615R - 0.51499G - 0.10001B \quad (7)$$

Using the above we can implement a function in python to convert each of YUV channels into RGB channels [1].

.



(a) Y channel (b) U channel (c) V channel (d) RGB image

Figure 9: YUV to RGB conversion

This conversion can be implemented into python using the following sample code:

```python
# Define the conversion matrices for YUV to BGR and BGR to YUV
YUV_to_BGR_conversion_matrix = np.array([[1.000, 0.000, 1.403], [1.000, -0.34413, -0.71414], [1.000, 1.773, 0.000]])
# Function to convert from YUV to BGR color space
RGB_to_BGR_conversion_matrix = np.array([[0, 0, 1], [0, 1, 0], [1, 0, 0]])
# Function which converts from YUV to BGR color space
def YUV2RGB_fxn(input_img):
    # First convert from YUV to RGB.
    yuv_image = input_img.astype(np.float32)
    yuv_image[:,:,1:] = yuv_image[:,:,1:] - 128.0
    rgb_image = np.dot(yuv_image, YUV_to_RGB_conversion_matrix.T)

    # Convert from RGB to BGR color space This is because we need it in the BGR order to show the image correctly with op
    bgr_image = np.dot(rgb_image, RGB_to_BGR_conversion_matrix.T)
    # 8-bit representation
    bgr_image = manual_clip_fxn(bgr_image, 0, 255).astype(np.uint8)
    return bgr_image
```

Figure 10. The sample python code which implements the YUV to RGB then BGR conversion.

In this code after converting to RGB we convert the color space into BGR to be able to visualize it using the OpenCV im.read() function. The result of this conversion is seen in figure 9.

7. Results and Discussion

After the implementation of each part of the system. We can see that we've constructed a system for which takes in an image, converts it into YUV color space then down samples it at the transmitter side, this compressed image is then received at the decoder and then it is up sampled, converted back into RGB and finally displayed at the receiver side. This method of encoding and decoding an image could be very necessary in image processing.

7.1. Results Obtained

Firstly, before discussing the results. We can implement a function for calculating the Peak-Signal-Noise Ratio PSNR of each images using the following formulas:

$$PSNR = 10 \log\left(\frac{255}{MSE}\right) \tag{8}$$

$$MSE = \frac{1}{W * H * 3} \sum_{I,J,C} (I_{I,J,C,REF} - I_{I,J,C})^2 \tag{9}$$

This can be implemented in python with the following sample code:

```python
def compute_PSNR_fxn(image1, image2):
    # Ensure both images have the same shape
    if image1.shape != image2.shape:
        raise ValueError("Error images don't have the same size dimensions")

    # compute mean squared error
    MSE_compute = np.mean((image1 - image2) ** 2)

    # compute PSNR
    if MSE_compute == 0:
        return float('inf')
    else:
        psnr = 20 * np.log10(255 / np.sqrt(MSE_compute))
        return psnr


def compute_MSE_fxn(image1, image2):
    # compare sizes
    assert image1.shape == image2.shape
    # Calculate the mean of the squared differences
    return np.mean( (image1 - image2) ** 2  )
```

Figure 11. The sample python code which implements the PSNR and MSE functions.

Using the formulas implemented we can test the accuracy and the amount of error we get from the down sampling and up sampling process. When we run the entire code combined. All images are shown on the screen, including PSNR and MSE values.

```
Input Image Pixel Size: 640x427
Bilinear_interpolation Output Image Pixel Size: 640x427
n_near_neighbors Output Image Pixel Size: 640x427
Bilinear_interpolation PSNR Error: 28.559973271347374
n_near_neighbors PSNR Error: 28.330496271939573
Bilinear_interpolation MSE Error: 90.59057864949258
n_near_neighbors MSE Error: 95.50601702771273
```

Figure 12. The results from running the entire code.

From Figure 12, we can see that we get the same image size after performing down and up sampling to the input image. We also see that we get a PSNR error of approximately 28.5599db and an MSE value of 90.591 for the bilinear interpolation up sampling method. As for the Machine learning algorithm for n nearest neighbors, we do see a slight improvement in the PSNR error value at approximately 28.3305db and a MSE value of 95.506. Keep in mind that these values differ depending on the image however, the range seems to be in between 25-30db PSNR error range for the tested images.

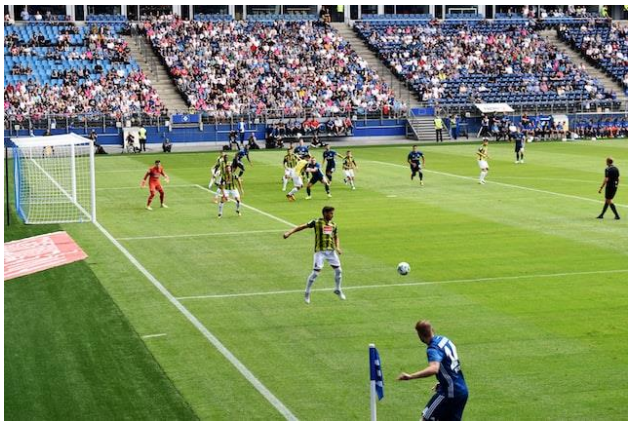


Figure 13. Input image.

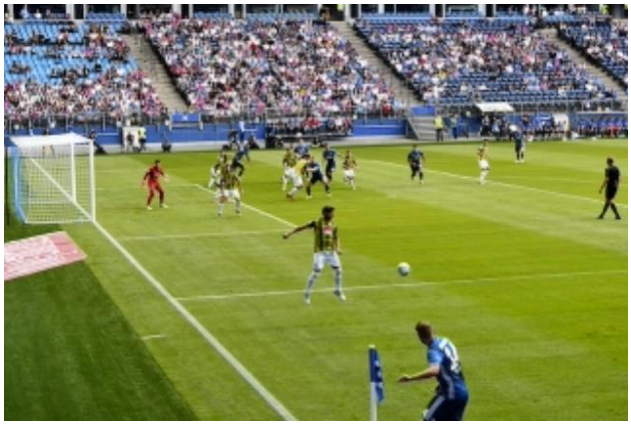


Figure 14. Output image at the receiver when using bilinear interpolation for up sampling.

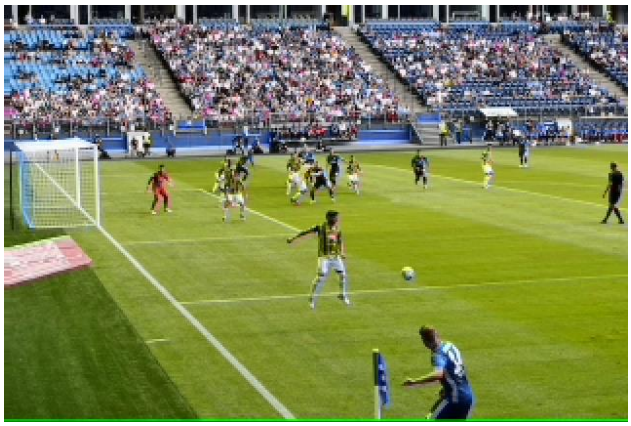


Figure 15. Output image at the receiver when using n nearest neighbors for up sampling.

7.2. Discussion and conclusion

We notice from Figures 13-15 that all are similar versions of the same image. The input image is of a better quality and is less pixelated than the output images. This was expected as we opted to use a lossy method for our image compression method. We can see that such a system is very useful and can be utilized for many transmitter-to-receiver image processing applications.

As we can see from the results, the PSNR value of this system is fairly reasonable, and the images reconstructed are a really good lossy representation of the input image at the transmitter.

To further improve the PSNR value of the system, we can consider choosing different values for factors at which we down scale YUV color spaces. Also, we can consider another color space system which is more curated for such application of image compression. Also, we can consider other techniques to up sample or down sample the image. An example of this is that we can use a trained model to utilize a deep learning neural network to perform the up scaling of the compressed image.

In conclusion, in this paper we discussed a very important technology in the image processing field. Image compression is a very widely used method in current day devices and we explored in this paper how an implementation of such a simple lossy image compression system could be done.

References

[1] Wikimedia Foundation. (2023, February 12). YUV. Wikipedia. Retrieved February 19, 2023, from https://en.wikipedia.org/wiki/YUV

[2] Lin, T.-L., Liu, B.-H., & Jiang, K.-H. (2021). An efficient algorithm for luminance optimization in chroma downsampling. IEEE Transactions on Circuits and Systems for Video Technology, 31(9), 3719–3724. https://doi.org/10.1109/tcsvt.2020.3039007

[3] *Resampling*. Chapter 6. resampling. (n.d.). Retrieved February 19, 2023, from http://pippin.gimp.org/image_processing/chap_resampling.html