MAT4110 Compulsory assignment 2

Jan Ole Åkerholm

October 29, 2019

Introduction

The goal of this assignment is to use a singular value decomposition in order to reduce the amount of storage space required in order to store image data.

A black and white image of height m and width n can be represented as an $m \times n$ matrix A. Typically the values are integers between 0 to 255, with 255 being completely white and 0 being completely black.

For this project, three different images will be used. These are originally color images, but are converted to black and white, and the integers are scaled to be between 0 and 1. This means the image matrix A is an $m \times n$ matrix with all the values in the matrix being between 0 and 1. The original images can be seen below:

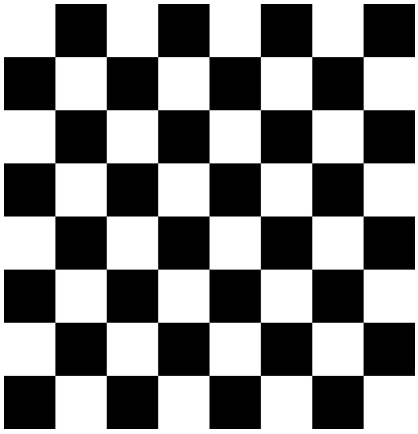

Figure 1: chessboard.png

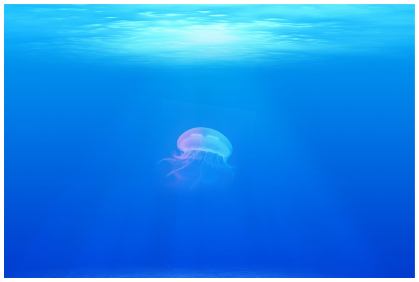

Figure 2: jellyfish.jpg

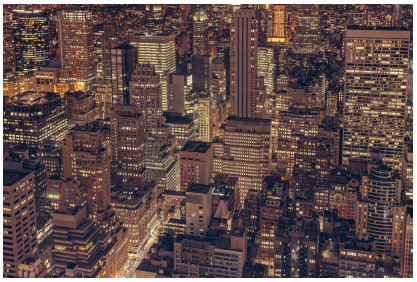

Figure 3: new-york.jpg

Compression using singular value decomposition (SVD)

Singular value decomposition allows decomposing an $m \times n$ matrix A in three separate matrices: U, S and V, so that $A = USV$.

Here U is $m \times m$, S is $m \times n$ and V is $n \times n$.

S is a rectangular diagonal matrix, and the values on the diagonal σ_i are known as the *singular values*. The singular values are commonly ordered in descending order in the matrix, i.e. $\sigma_1 > \sigma_2 > ... > \sigma_r$ (where $r = \min(m, n)$).

The compression can be done by retaining only the k largest singular values, where k is some chosen integer which is smaller than $\min(m, n)$. The U and V matrices must be similarly reduced, so that their shapes are now:
$U : m \times k$
$S : k \times k$ containing the k largest singular values on the diagonal.

$V : k \times n$
i.e. we keep only the first k columns of U, and the first k rows of V.

From the shapes it's clear that the result of the product USV is still an $m \times n$ matrix, even though the matrices are much smaller.

Compression ratio

The number of values requried to store the original matrix A is

$$\text{size}_{\text{uncompressed}} = mn$$

The number of values required to store the compressed singular value decomposition is

$$\text{size}_{\text{compressed}} = mk + k + kn = k(1 + m + n)$$

This gives the compression ratio:

$$C_R = \frac{\text{size}_{\text{uncompressed}}}{\text{size}_{\text{compressed}}} = \frac{mn}{k(1 + m + n)}$$

Note that if k is large, the compression ratio may be lower than 1. i.e. it requires more values to store the SVD than the original matrix. The compression is therefore only worth doing if k is such that $C_R > 1$, i.e.

$$\frac{mn}{k(1 + m + n)} > 1$$

$$\implies k < \frac{mn}{1 + m + n}$$

For the special case of a square image, i.e. $n = m$, this becomes:

$$k < \frac{m^2}{1 + 2m}$$

Implementation

The implementation is done using Python with the "Pillow"-package for loading images and converting them to black and white images. The uncompressed black and white images look as follows:

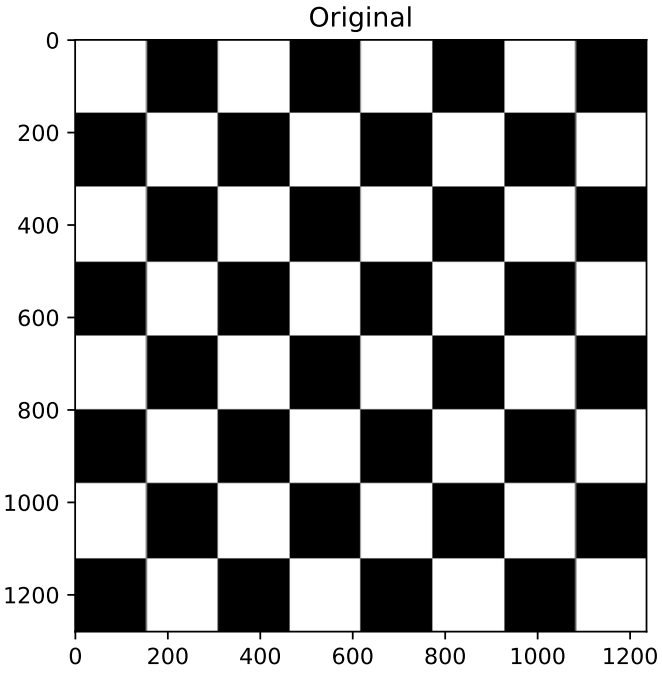

Figure 4: Chessboard image converted to black and white

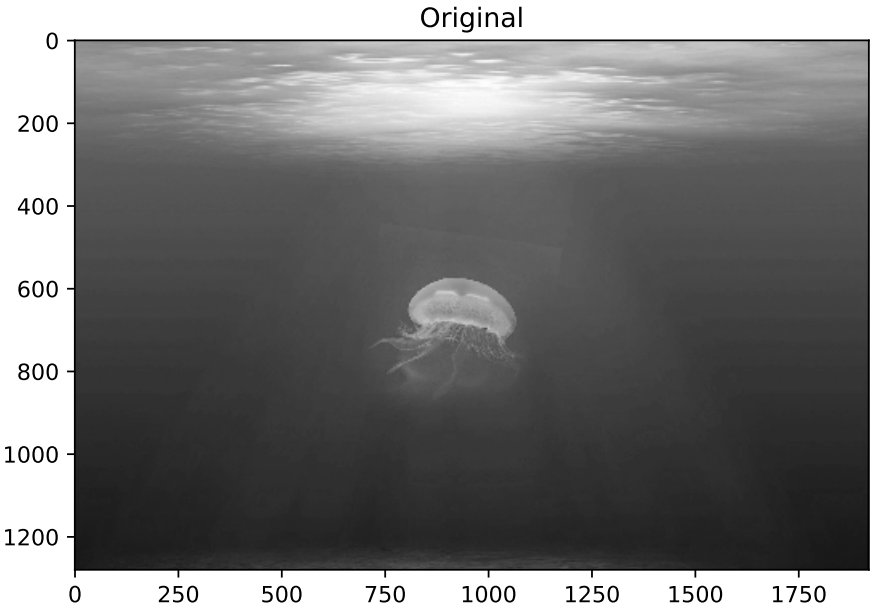

Figure 5: Jellyfish image converted to black and white

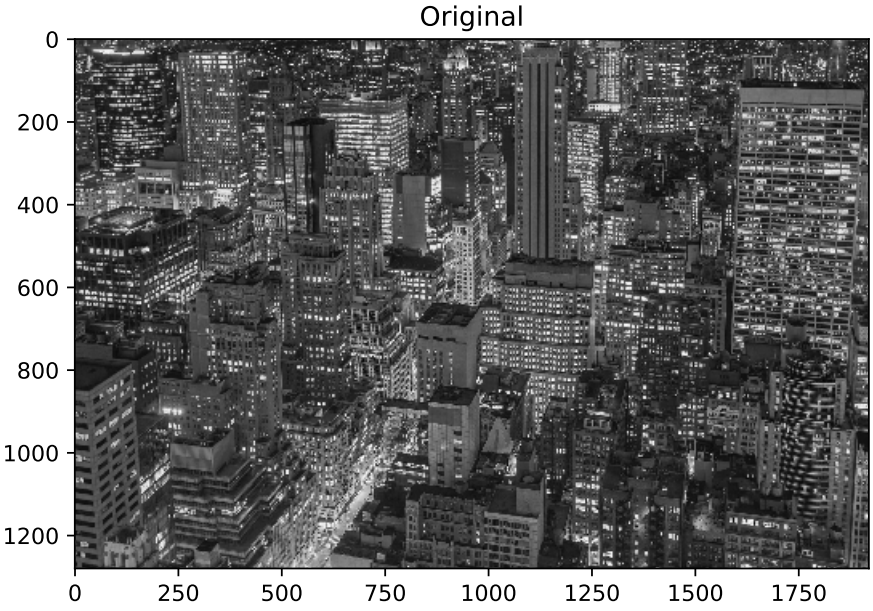

Figure 6: New York image converted to black and white

The compression is done using a function "SVD_compress(im, k)":

```
def SVD_compress(im, k):
    max_k = np.min(im.shape)
    if k > max_k:
        print("Chosen k is larger than smallest dimension, reducing to k =",
              max_k)
        k = max_k

    U, S, V = np.linalg.svd(im)
    U = U[:,:k]
    S = S[:k]
    V = V[:k,:]

    return U, S, V, k
```

The function takes the image matrix and the k values to retain as input, and returns U, S, V and k. The reason for returning k is in case the function chooses a smaller value than the input (since k must be smaller than $\min(m, n)$).

Note that S is stored as a vector containing the singular values, and not a diagonal matrix, as there is no reason for storing the extra zeroes that are not on the diagonal.

The decompression is simply done by matrix multiplying the U, S and V vectors together:

```
def SVD_decompress(U, S, V):
    US = U @ np.diag(S)
    im = US @ V
    return im
```

Three other functions are made for the program, one for plotting the compression ratio as a function of k for various image sizes, one for returning the number of values required to store the SVD, and one for converting an image to black and white and scaling the matrix, those are as follows:

```
def plot_compression_ratio():
    """ Calculates and plots the compression ratio for an m x n image as a
    function of k
    """
    for m in [512, 1024, 2048]:
        for n in [1024]:
            k_max = min(m, n)
            k = np.linspace(25, k_max, 1000)
            cr = m*n/(k*(1 + m + n))
            plt.plot(k, cr, label = "m = {:d}, n = {:d}".format(m, n))

    plt.plot([25,k_max], [1,1], "r--", alpha = 0.5, label = "1")
    plt.legend()
    plt.title("Compression ratios")
    plt.xlabel("k")
    plt.ylabel("Compression ratio")
    plt.savefig("images/CRs.pdf")
    plt.show()

def SVD_size(U, S, V):
    """ Returns the number of values required to store a singular value
    decomposition
    """
    return U.size + S.size + V.size


def rgb2gray(im):
    """ Takes a color Pillow image and returns a 2d grayscale Numpy matrix
    with values scaled between 0 and 1
    """
    return np.matrix(im.convert("L"))/255
```

The code is then used as follows:

```
plot_compression_ratio()


im = Image.open("chessboard.png")
im = rgb2gray(im)

im2 = Image.open("jellyfish.jpg")
im2 = rgb2gray(im2)

im3 = Image.open("new-york.jpg")
im3 = rgb2gray(im3)


image_list = [im, im2, im3]
image_names = ["chessboard", "jellyfish", "new-york"]
k_list = [int(1280*2**(-i)) for i in range(11)]
print(k_list)

```

```
for image, image_name in zip(image_list, image_names):
    plt.figure()
    plt.imshow(image, cmap="gray")
    plt.title("Original")
    fn = "images/{:s}-original.pdf".format(image_name)
    plt.savefig(fn)

    for k in k_list:
        U, S, V, k = SVD_compress(image, k)
        CR = image.size/SVD_size(U, S, V)
        im_compressed = SVD_decompress(U, S, V)
        plt.figure()
        plt.imshow(im_compressed, cmap="gray")
        plt.title("Compressed, k = {:d}, CR = {:f}".format(k, CR))
        fn = "images/{:s}-{:04d}.pdf".format(image_name, k)
        plt.savefig(fn, dpi = 600)

    U, S, V, k = SVD_compress(image, np.min(image.shape))
    plt.figure()
    cross_1 = np.argmin(np.abs(S - 1))
    print("First 5 singular values:", S[:5])
    print("Singular values cross 1 at k={:d}".format(cross_1))
    print("Singular values at cross:", S[cross_1-3:cross_1+2])
    plt.semilogy(S)
    plt.title("Singular values")
    plt.xlabel("index")
    fn = "images/{:s}-svs.pdf".format(image_name, k)
    plt.savefig(fn)

    plt.show()
```

Results

Compression ratio

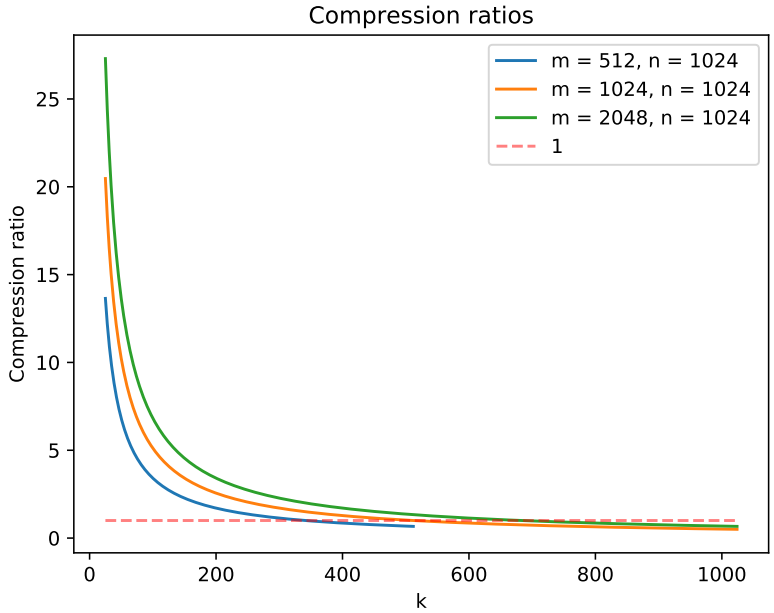

Figure 7: Compression ratio of an image using SVD compression for various values of k and various different image sizes

For all cases it's clear that if $k = \min(m, n)$ the compression ratio is below 1, meaning the required number of values to store the SVD is larger than the number of values required to store the original matrix A.

Chessboard

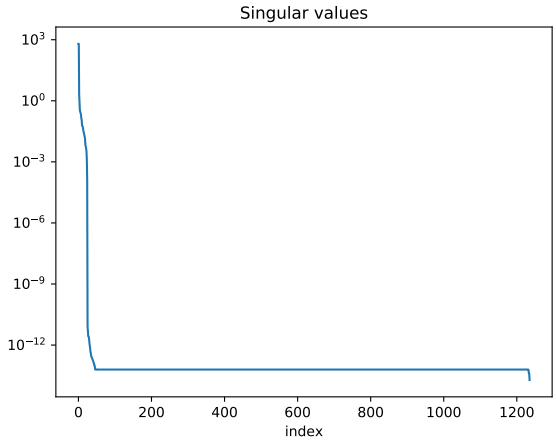

Figure 8: Singular values of the chessboard matrix

Figure 8 shows the singular values for the image matrix corresponding to the chessboard image. The singular values drop off very fast, and become 0 (to numerical precision) almost immediately. In fact the values become smaller than 1 already at σ_4. This indicates that not much information is lost by removing most of the singular values, and a decent compression can be found even with $k = 2$:

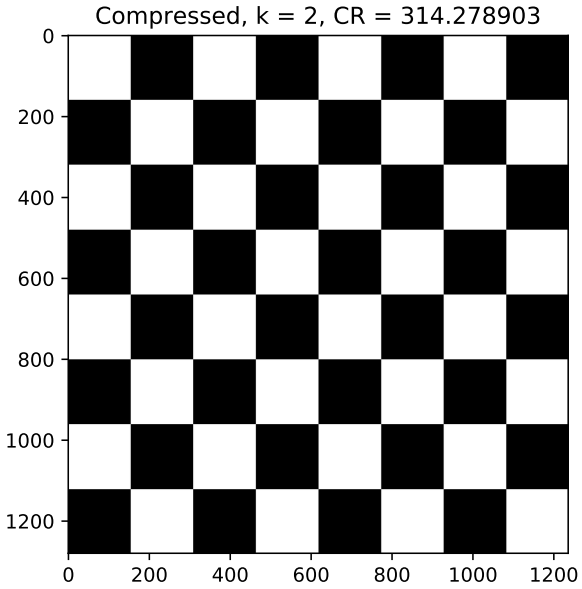

Figure 9: Chessboard image compressed with $k = 2$. In this case the compression ratio is more than 314.

It's not unsurprising that an image with such a regular pattern is easy to compress, as very little information is technically needed in order to represent a pattern like this.

Jellyfish

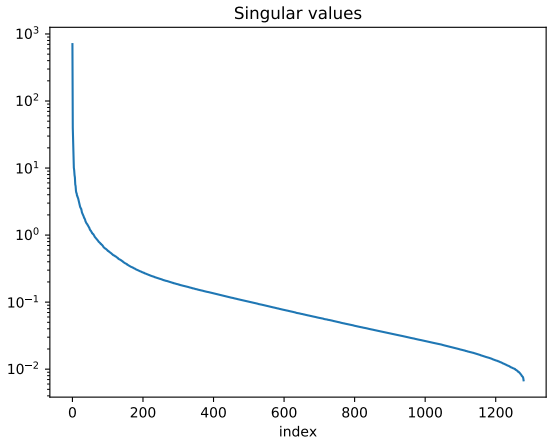

Figure 10: Singular values of the jellyfish matrix

Figure 10 shows the singular values for the image matrix corresponding to the jellyfish image. In this case the singular values drop off much more slowly, and doesn't reach 0 until very late. This shows that even with a large value k there will be some loss of information. In this case the singular values drop below 1 at σ_{62}. A decent image can be found with k as small as $k = 80$, giving a compression ratio of approximately 9.6:

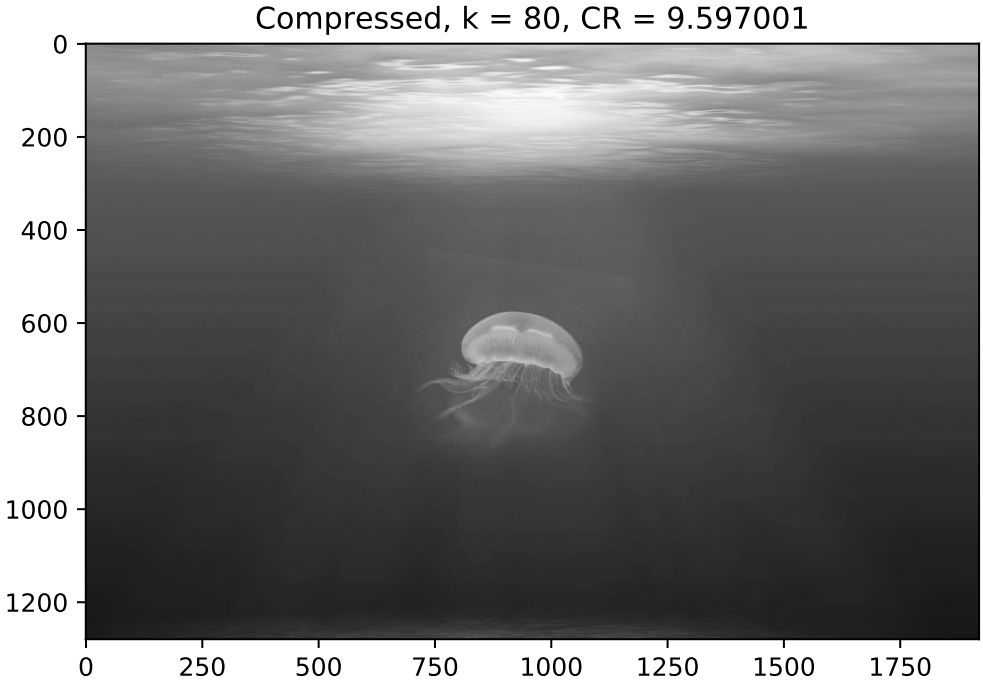

Figure 11: Jellyfish image compressed with $k = 80$.

With $k = 40$ it's clear that significant information is getting lost and compression artifacts appear in the image:

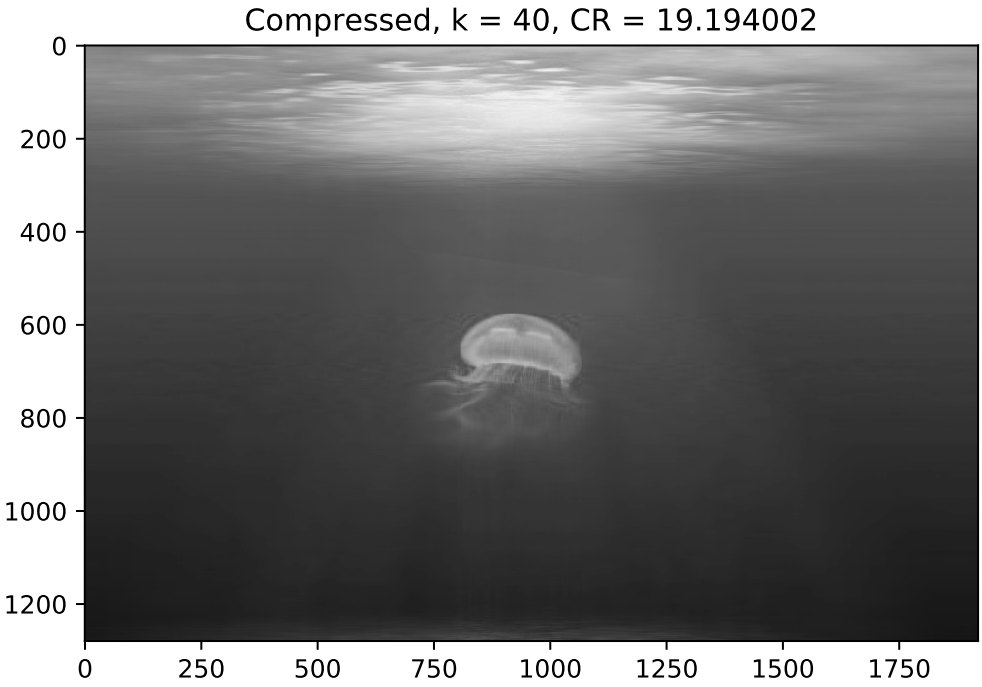

Figure 12: Jellyfish image compressed with $k = 40$.

New York

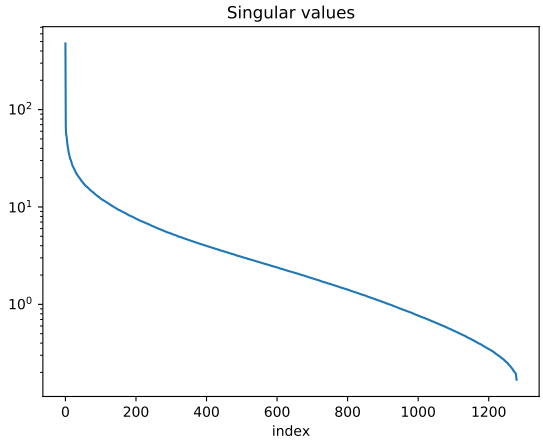

Figure 13: Singular values of the New York matrix

Figure 13 shows the singular values for the image matrix corresponding to the New York image. In this case the singular values drop off much more slowly than even the jellyfish image, indicating more singular values are required to store the information in the image. By looking at the original image we can see that there is a lot of detail which does not appear to follow any common pattern. This is in contrast to the chess board image which has a very clear pattern, and the jellyfish image which has large fairly uniform areas and few sharp edges.
In this case the singular values drop below 1 at σ_{919}. With $k = 640$ there is visible loss of information, but the image still looks decent. In this case the compression ratio is approximately 1.2:

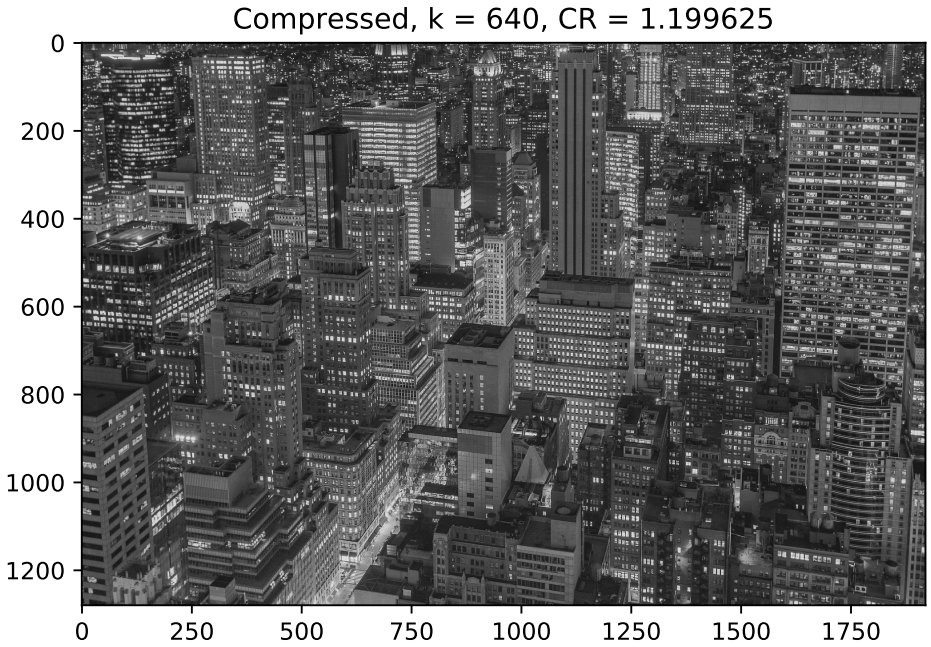

Figure 14: New York image compressed with $k = 640$.

At $k = 320$ the image starts clearly showing compression artifacts. There is noise, and the contrast is reduced:

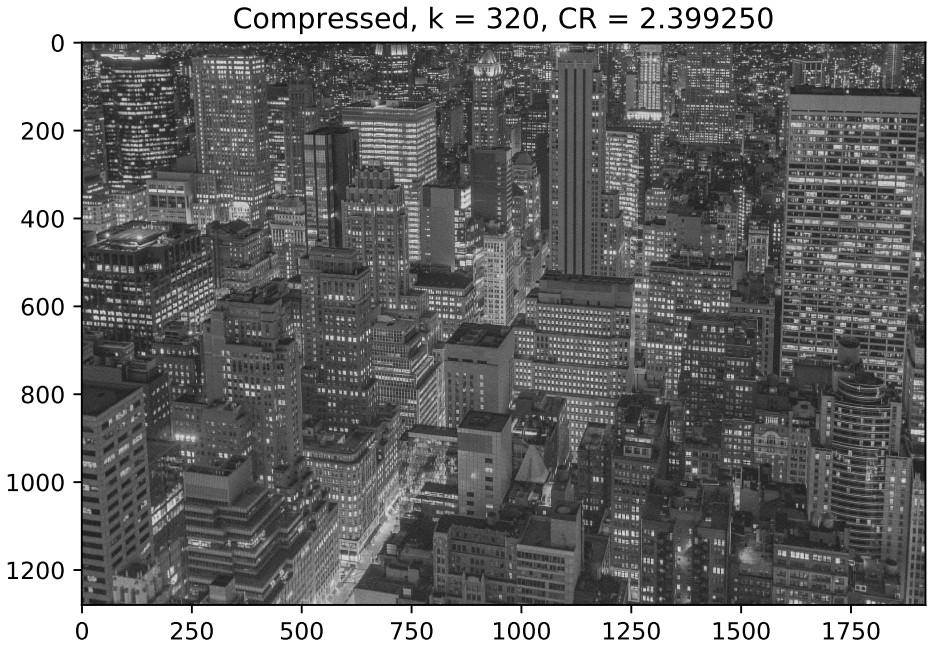

Figure 15: New York image compressed with $k = 320$.

At $k = 80$, which was fine for the jellyfish, the image is clearly much worse off than the original:

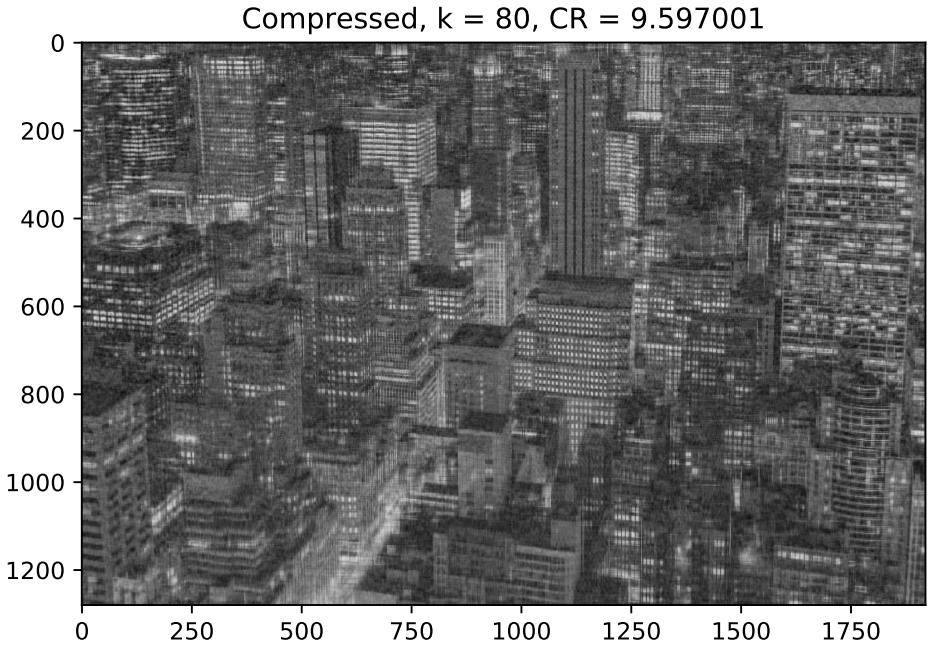

Figure 16: New York image compressed with $k = 80$.

At $k = 2$, which was fine for the chessboard, the image is completely unrecognizable:

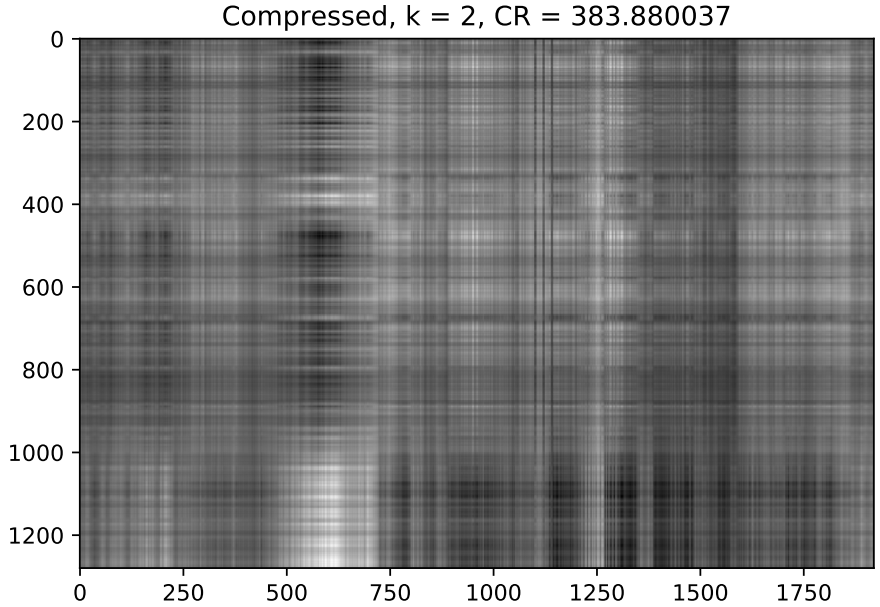

Figure 17: New York image compressed with $k = 2$.