Towards Finding an Optimal Image Compression Algorithm for Matlab: ScaleCompress

David Itten, Martin Lanter, Daniel Schweizer
Computational Intelligence Lab Project, ETH Zurich, Switzerland

***Abstract*—We present ScaleCompress, a three-step image compression algorithm. It first scales the input image to a smaller size, then reduces the number of bits representing a pixel, and additionally uses run-length encoding to further reduce the size of the compressed image. The evaluation shows that ScaleCompress performs clearly better than our PCA and SVD-based solutions from the assignments. ScaleCompress is very efficient for images with patches of similar colors or color gradients (e.g. photographs of landscapes or people), and performs sufficient on images with lots of fine-grained details (e.g. texture images).**

I. Introduction

Matlab offers a wide range of image manipulation tools. All techniques applied by Matlab operate on matrices with a minimum of 8 bits per cell. With this representation it is not possible to improve the image compression rate when using fewer than 8 bits to represent a pixel. Our techniques try to leverage this problem by applying bit level techniques in order to achieve a denser representation of the images in Matlab. Our novel approach to find an optimal image compression algorithm for Matlab is combining three different algorithms described in section **??**.

II. Models and Methods

ScaleCompress is a chain of three successive compress-algorithms. It consists of the following steps:

- Scaling an image to a smaller size.
- Reducing the number of bits representing a pixel, thus cutting off the least significant bits
- Merging sequences of equal values (runs).

The decompression algorithm simply goes through the corresponding decompress-steps in reverse order. The following sections describe the steps of ScaleCompress in more detail and explain how they are put together to form a powerful and efficient compression algorithm.

A. Image Scaling

The Image Scaling algorithm uses Matlab's image resize function (with bicubic interpolation) to reduce the image to a smaller size. To decompress the image again, just scale the compressed image back to its original size. Scaling compresses the image with a deterministic, predictable compression rate. Thus, the algorithm's compression rate can directly and accurately be adjusted by modifying the scaling factor. Scaling an image down blurs very capillary details away. It however leaves the overall structure almost untouched. In particular, color gradients remain almost perfectly retained.

B. n-bit Compression

Matlab represents images as W×H×Z matrices, where W and H are the width and height of the image and Z is the number of color channels. The entries of such a matrix are unsigned 8-bit integers. For compression purposes, however, these values can be reduced to far less bits (e.g. 4), which will only result in a marginal loss of precision. BitCompress is a part of ScaleCompress that takes a matrix M and returns a compressed form of it. The parameter n defines the number of bits to be used to represent one value. In order to achieve an optimal usage of the n-bit representation, the values from M are shifted and scaled in such a way that the smallest value from M will be represented as 0, the largest value as 2^{n-1}, and the other values are mapped linearly to their corresponding representation in the interval [0 ... 2^{n-1}]. Due to implementation reasons (Matlab doesn't offer a 1-bit data type), n-bit-compression returns the compressed form of M as a vector. Additionally, it also returns the size of the original matrix, such that the decompressing algorithm is later able to recreate a matrix of the correct size.

C. Run-length Encoding

When doing the bit compression, we additionally apply run-length encoding [1] in order to get an even better compression rate. Notice that this step is lossless. As the bit compression narrows the number of different values in the image representation (i.e. 16 for 4-bit instead of 256 for 8-bit), entries next to each other often have equal values. At this point, run-length encoding can become very effective. Looking at the sequence of n-bit integers returned by the bit compression algorithm from above, we can differentiate two cases:

- If there are L $\geq$ 2 equal values v in a sequence, we call this a run. A run can be encoded in a space-efficient way: BitCompress first writes a 0 bit to state that the encoding of a run follows, then it uses r bits (where r is an input parameter of the algorithm) to encode the length of the run, and then it uses n bits to encode v.
- If we have a single value (i.e. its neighbors are different from it), BitCompress first writes a 1 bit to state that the encoding of a single value follows. Then, this value is encoded using n bits.

Figure **??** shows how BitCompress compresses some small matrix (including run-length encoding). Notice that the resulting output vector is not only shown as

a bitstring, but also as three 8-bit integers, as it has to be stored in this form in Matlab. Also notice that the decompressed matrix differs only slightly from the original one.

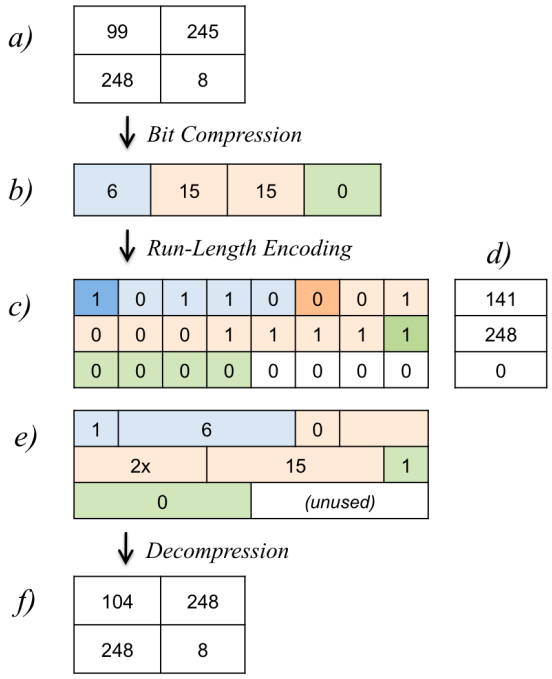

Figure 1. Bit Compression using Run-Length Encoding of a 2x2 8-bit matrix: a) The original matrix. b) The resulting vector when 4-bit compression is applied to the original matrix. c) The resulting bitstring when Run-Length Encoding is used to store the vector. The last four bits are not used and will not be read in the decompression phase. d) Interpretation of the bitstring as three 8-bit integers. This is how the compressed vector is stored. e) Schematic representation of how the blocks in the bitstring in c) have to be interpreted: The blue and green blocks - initiated by a 1 bit - denote single values. The orange block - initiated by a 0 bit - denotes a run of length 2. f) Restored matrix.

We also tried to use some Huffman encoding [2] for lossless compression of images. The idea of Huffman encoding is to use a smaller bit representation for pixel values that occur more often within an image without increasing the error of the compressed image. Unfortunately the runtime of this algorithm was quite high because of the large amount of linear-time lookups in the encoding dictionary. Therefore, we didn't further follow this approach.

D. Algorithm Benchmark

There are three obvious criterions when benchmarking an image compression algorithm: Runtime, compression rate and compression error. Thus, any image compression algorithm applied to a set of test images can be represented as a point in 3-dimensional space, where one axis stands for one criterion, respectively. The closer a point is to the origin, the better the algorithm is. For the optimization of ScaleCompress, we focus mainly on compression rate $r \in [0, 1]$ and compression error $e \in [0, 1]$. Therefore, we define the benchmark $b = e^2 + r^2$ which is the squared Euclidean distance from the point (e, r) to the origin. Both values e and r have the same weight in this benchmark. We treat the runtime only as additional information and expect it to be sufficiently low but we don't include it in the benchmark.

We used this benchmark with a set of test images to evaluate different compression algorithms. In particular, we decided against the use of a PCA-based algorithm. PCA's runtime is somewhat lower than the runtime of ScaleCompress. However, ScaleCompress yields results with a significantly better compression rate and error compared to PCA. A combination of both does not work out well. Instead of benefitting from the strengths of the two algorithms, rather their weaknesses dominate the combination of the algorithms.

Furthermore, we found that algorithms using SVD don't lead to better results either. An implementation that reshapes the image matrix to an almost quadratic size before applying SVD turned out to be better than just applying SVD but still didn't reach the benchmark values of ScaleCompress and moreover was significantly slower.

III. Results

We evaluated ScaleCompress with three different sets of images from the USC-SIPI image database[1].

- misc: Various images with gradients like photos.
- aerials: Aerial photographs with structures like streets.
- textures: Highly detailed structures like walls or bark.

The three different sets give us a rough grasp of how ScaleCompress performs for different types of images.

A. Finding the optimal scaling factor

The scaling factor directly affects the compression rate and error of ScaleCompress. While the compression rate only depends on the scaling factor, the error also depends heavily on the image itself. With a scaling factor s the algorithm scales both width and height of an image down by s, thus reducing the amount of pixels by $1/s^2$ in total. The higher s is, the lower is the compression rate and the higher is the error. In the worst case, scaling averages the color of pixels and leads to an error of 0.5 per pixel. We have measured the impact of scaling factors between 1 (no scaling at all) and 32 and computed the benchmark values. Figure **??** shows the average quadratic error and the benchmark for the three different image test sets. For the set misc the scaling factor 6 results in the best benchmark value. For aerials it is 8 and for textures it is 4.

B. Finding the optimal parameter values for BitCompress

BitCompress has two parameters which must be chosen carefully in order to achieve good results. The first parameter n is the number of bits used to represent a pixel of the image. The second parameter r is the number of bits that are used to decode the length of a run. Therefore we analyzed the errors and compression rates varying both parameters from 3 bits to 8 bits. Notice that using 8 bits for

[1] http://sipi.usc.edu/database, 15.6.2012

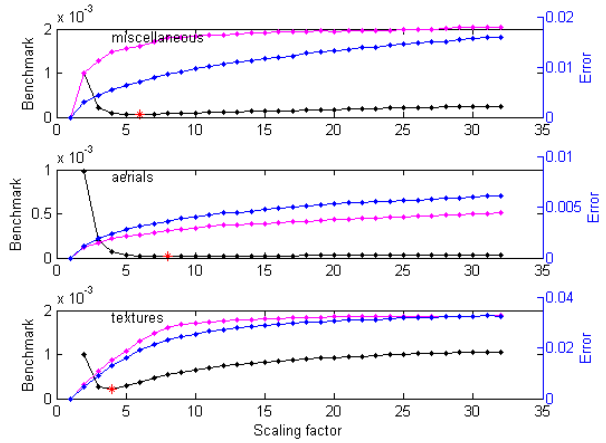

Figure 2. Average quadratic error (blue) and error standard deviation (pink) and benchmark (black) for scaling factors from 1 to 32. The optimal benchmark value is marked with a red star.

storing the pixels will make BitCompress useless as this is anyway the representation used by Matlab. Figure **??** shows how the mean error evolves when changing the number of bits per pixel. It can be seen clearly that the error is much higher for the textures set, as our compression techniques are not optimal for that kind of images. The mean error for the other two test sets are reasonably low and the mean error stabilize for all test sets at a value of 4 bits per pixel. Notice that the number of bits used to encode the run length has no influence on the error because the run-length encoding is loss-free.

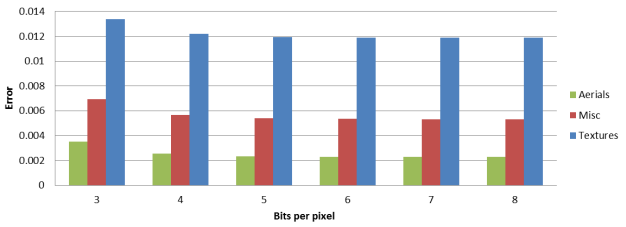

Figure 3. Mean error when changing the number of bits per pixel

When encoding with 4 bits per pixel, we see in figure **??** that the compression rates stay almost the same when using 3 or 4 bits to encode the run lengths. When using more bits, the compression rate decreases. We also varied the number of bits per pixel, but because the error grows quite fast for values lower than 4 bits and higher values do not lead to a significantly lower error, we decided to use 4 bits per pixel.

The standard deviation does not vary much with the parameter selection. For $n = 4$ and $r = 4$ we calculated a standard deviation of 0.0130 for the error, which is much higher than the mean of the errors. This is the case because some images within the test sets (4 of 44 images for the misc test set) differ extremely from the mean error. The compression rate is more stable and has a standard deviation of 0.0009.

The example in Figure **??** shows the result of applying

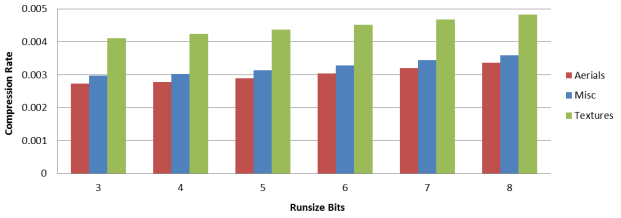

Figure 4. Compression rates when changing the number of bits to encode the run lengths

ScaleCompress to a test image. As a baseline, there is the original image and an image created with a PCA-based algorithm shown. The parameters of the PCA algorithm have been choosen to achieve the same compression rate as ScaleCompress does for this particular image. The best result using PCA has been achieved by using two eigenvalues and a patch size of 10 pixels.

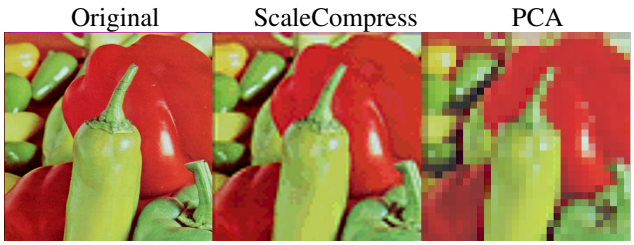

Figure 5. Comparison between PCA and ScaleCompress

C. Performance evaluation

For the performance evaluation ScaleCompress was compared to two baseline algorithms (using our implementations from the assignments): an algorithm using PCA and one using SVD factorization.

In a first experiment, we applied ScaleCompress to the three image test sets and chose the parameters for the SVD and PCA algorithms in such a way that we achieved approximately the same mean error as with ScaleCompress. The comparison of the achieved compression rates with a fixed error can be seen in table **??**. For the textures test set, where ScaleCompress produces a rather large error, ScaleCompress still provides a compression rate which is about 22 times better than the one achieved with SVD, and twice as good as PCA. The aerials test set shows the best results in this test. Here, the compression rate of ScaleCompress is 32 times better than with SVD, and again about two times better than PCA.

In a second experiment, we fixed the compression rate that had been achieved by ScaleCompress and compared the mean error of ScaleCompress to the mean error of PCA and SVD. The results are shown in table **??**. ScaleCompress produced a mean error which was between 2.6 (textures) and 3.7 (aerials) times smaller than the mean error of SVD. For the misc test set, we could not even reach the fixed compression rate (which was 0.003) with SVD. Compared to PCA, the mean error produced by ScaleCompress was between 1.25 (textures) and 1.4 (aerials) times smaller.

Comparison of compression rates with fixed error					
	ScaleCompress	PCA	PCA/SC	SVD	SVD/SC
Aerials	0.0028	0.0059	2.12	0.088	31.6
Misc	0.0030	0.0056	1.85	0.073	24.1
Textures	0.0042	0.0082	1.94	0.093	22.0

Table I
AVERAGE COMPRESSION RATES OVER ALL IMAGE TEST SETS FOR FIXED ERROR

Comparision of errors with fixed compression rate					
	ScaleCompr.	PCA	PCA/SC	SVD	SVD/SC
Aerials	0.0025	0.0036	1.40	0.0093	3.67
Misc	0.0056	0.0078	1.37	–	–
Textures	0.012	0.015	1.25	0.032	2.64

Table III
AVERAGE ERRORS OVER ALL IMAGE TEST SETS FOR FIXED COMPRESSION RATE

Comparison of compression rates with fixed error			
	ScaleCompress	PCA	SVD
Aerials	3.122 (3.227)	0.606 (0.492)	3.117 (4.922)
Misc	0.497 (0.013)	0.238 (0.106)	0.301 (0.277)
Textures	0.922 (0.557)	0.569 (0.185)	0.537 (0.421)

Table II
AVERAGE RUNTIME (IN S) OVER ALL IMAGE TEST SETS FOR FIXED ERROR (STANDARD DEVIATION IN BRACKETS)

Comparison of errors with fixed compression rate			
	ScaleCompress	PCA	SVD
Aerials	3.122 (3.227)	0.751 (0.517)	3.126 (4.882)
Misc	0.497 (0.013)	0.156 (0.095)	–
Textures	0.922 (0.557)	0.229 (0.129)	0.541 (0.429)

Table IV
AVERAGE RUNTIMES (IN S) OVER ALL IMAGE TEST SETS FOR FIXED COMPRESSION RATE (STANDARD DEVIATION IN BRACKETS)

IV. DISCUSSION

For all three image test sets, ScaleCompress yielded better results than PCA and SVD. However, we can see that the difference in performance compared to PCA/SVD is clearly less significant for the textures test set than for the other two test sets. ScaleCompress loses more precision when compressing images with lots of fine-grained details (such as textures), while it behaves much better on images dominated by coarse-grained patches of similar coloring or color gradients. This is exactly what we expected.

One has however to be aware that ScaleCompress was only compared to basic PCA- and SVD-based algorithms from the assignments (which were most probably not implemented in the best possible way). The comparison to the solutions of the other project groups will certainly be more informative.

Compared to PCA and SVD, ScaleCompress reaches a significantly better compression rate for a fixed mean error. This is due to the fact that scaling as well as bit compression significantly decreases the size of the compressed image, while only introducing a small error.

We can for example look at 4-bit compression, i.e. representing each entry of a matrix with only 4 bits instead of the standard 8 bits. Assuming that we use the full spectrum of 8-bit numbers, this corresponds to mapping the interval [0, 255] to the interval [0, 15]. Let's assume we have a 8-bit value V, compress it to a 4-bit value, and then decompress this value back to D. We then have $D = floor(V/16)*16$, where D differs from V by at most 15 units. Thus, the upper bound for the mean quadratic error which is generated by 4-bit compressen is $1/256$.

One main parameter of ScaleCompress is the scaling factor. The benchmarks evaluation suggests using a scaling factor between 4 and 8. Since a factor higher than 4 erases a lot of structural details in an image and since the benchmark for a factor of 4 is close to the optimum for all three image test sets, we decided to use a scaling factor of 4.

It is imaginable that parameters that perform best for a partial algorithm are not necessarily optimal values for the combined algorithm ScaleCompress. Tests however have shown that ScaleCompress does not improve when using other values. Therefore, the main parameters that define ScaleCompress are 4 for the scaling factor, 4 for the amount of bits encoding a single pixel and 4 for the amounts of bits encoding the length of a run of equal pixels.

V. SUMMARY

ScaleCompress is an efficient and novel image compression algorithm. It provides a fair tradeoff between runtime, compression rate and compression error. Its parameters have been chosen to guarantee both a low quadratic mean error and compression rate, weighting both factors equally. ScaleCompress returns a well predictable compression rate for an image. Furthermore, the algorithm is easily adjustable. Increasing the scaling factor leads to an immediate lower compression rate and lower runtime.

REFERENCES

[1] D. Pountain, "Run-length Encoding", *Byte*, vol. 12, no. 6, pp. 317-319, 1987.
[2] D.A. Huffman, "A Method for the Construction of Minimum-Redundancy Codes", *Proceedings of the I.R.E.*, pp. 1098-1102, 1952.