PNG Images Compression Through a Convolutional Autoencoder with Adjustable Error Correction

Giovanni Ciampi, Francesco Mogavero

July 9, 2019

Contents

1 Introduction **3**

2 Literature Review **5**

3 Autoencoder **7**

3.1 Implementation 8

4 Compressor **9**

4.1 Error correction 9

5 Decompressor **11**

6 Hardware and Software Platforms **12**

7 Performances **13**

7.1 Graphic performances 13

7.2 Temporal performances 14

7.3 Compression ratio performances 15

7.4 Decoder error 15

7.5 Compression Ratio 16

7.6 PSNR 16

7.7 Comparison with JPEG 17

1 Introduction

The proposed work has been focused on the construction of an Autoencoder for the compression and decompression of PNG images, starting from the study of the article *Lossy Image Compression with Compressive Autoencoders*, a convolutional autoencoder has been built using specific Python libraries. The first six layers of the network (the input layer plus the first 5 hidden layers) constitute the *Encoder*, the main building block of the compressor. The encoder compresses blocks of the input images of size (128, 128, 3) in blocks of size (32, 32, 16). The input blocks are matrices of 128x128 pixels, each of which is represented by a triple of normalized RGB values. The next 7 layers of the network (6 hidden layers plus the output layer) constitute the *Decoder*, the main component of the decompressor. The decoder takes as input the blocks compressed by the encoder, of size (32, 32, 16), and brings them back to the original size (128, 128, 3). The decompressed blocks are then rearranged in the original fashion in order to get the output of the decompression.
The network has been trained for a total of 40 epochs, with the 800 images of the train set provided by the *HR Images* dataset[1]. After the training phase, the trained network has been tested on a validation set consisting of 100 images not seen during the training by the network, provided by the same dataset used to get the training set. Results on the validation set showed that for the average image of the validation set, the *mean absolute error* was generally ≤ 0.06, while the *standard deviation* of the *absolute error* was ≤ 0.07. In order to allow for a better visual experience, an *error correction* algorithm has been realised in order to correct the biggest decompression errors. In particular, the compressor, after compressing an image through the encoder, decompresses it through the decoder, and computes an estimation of the decompression error. Through a parameter called *quality*, the user can choose the level of error correction to apply. As regards the alpha channel of the images, where present, it is separated from the rgb channels when the image is loaded and it is not compressed by the network. The compressor will then send three matrices to the decompressor, representing the encoded blocks of the rgb channels of the image, the error correction and the alpha channel, if present. In order to get a better compression ratio, each matrix is transformed in a sequence of bytes and compressed lossless through the *Lempel-Ziv-Markov chain* (*LZMA*) algorithm. In addition to that, the compressor will send to the decompressor some RAW metadata, that will be used in order to bring the decompressed image back to the original shape. After that, the error correction will be applied and the alpha channel, if present, will be reintroduced. As regards the performances of compressor and decompressor, compression ratio and PSNR have been evaluated on some standard images (*Lena*, *Mandrill*, *Parrots*) varying the quality parameter; in addition to that, also temporal performances have been evaluated. The results show that for the three images, by setting the quality parameter to the minimum, a compression ratio ≥ 2.75 is obtained; as regards the PSNR, by setting the quality parameter to its maximum, it is possible to get scores above $60dB$. In order to get a comparison with the state

[1] https://data.vision.ee.ethz.ch/cvl/DIV2K/DIV2K_train_HR.zip

of the art, the results have been compared to the performances obtained by the JPEG standard on the same images.
All the source code is available online at the address
http://github.com/FraMog/ProgettoCompressioneDati .

2 Literature Review

The developed compression algorithm is mainly based on the study of the article *Lossy Image Compression with Compressive Autoencoders*[2].

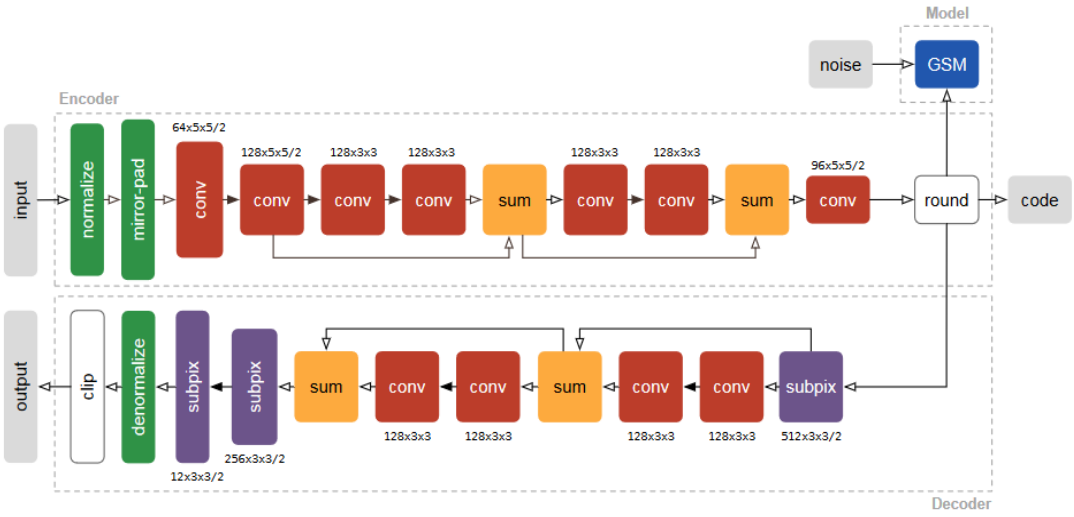

Figure 1: **Encoder (top) and Decoder (bottom)** depicted in the article in *Lossy Image Compression with Compressive Autoencoders.* The label C x K x K above every convolutional layer indicates that C filters are applied (so C output channels are obtained), each one with shape K x K. The value following the "/" symbol indicates the stride (obtained by pooling)

In the article, the neural network shown in *Figure 1* is realized. The network takes as input blocks of size 128x128 of a RAW image, consisting of RGB pixels (so the blocks have three channels). Each block is normalized and given as input to the encoder that compresses it, independently from the others. In order to compress the input block, the encoder applies mirror padding to it, and a series of convolutional layers progressively shrink the spatial dimensions of the block, increasing the number of channels up to 96. It is clear at this point that the compression is lossy, since the hidden layers in the middle of the network are not able to represent all the information given as input. At this point, the values output of the central layer of the network are *quantized*, by mapping them to a small set of values chosen carefully. A greater quantization factor contributes to a further distortion of the decompressed image, although it guarantees a higher compression ratio. The quantized block is then losslessly compressed through a range coder, and transmitted to the decompressor independently from the other blocks.
The decoder, receiving a compressed block as input, decompresses it though the range (de-)coder at first, then though the decoder (nertwork); the block is then de-

[2]https://arxiv.org/abs/1703.00395

The research work of the authors of the paper, once defined the architecture of the network, was mostly focused on finding an optimal quantization factor, in order to get the best balance between compression ration and distortion.

3 Autoencoder

Autoencoders are artificial neural networks trained in order to reconstruct their inputs. In particular, these networks have an output layer of the same size of the input layer, and at least one hidden layer with a smaller number of units. These characteristics make autoencoders very popular for dimensionality reduction and/or data compression. One of their major advantages, compared to classical algorithms, like PCA, is that they can adapt to the context: they can be fitted to a particular kind of data, like images, and to any subset, like portraits, landscapes, or cat pictures.
In a more formal fashion, an autoeencoder is a feed-forward artificial neeural network that follows the schema represented in *Figure 2*: it has an input layer and an output layer with N units, and a central hidden layer with M units, with $M < N$; this central hidden layer is often referred as *code*. The hidden layers that connect the input layer to the code usually contain a number of units that progressively decreases as the layer gets closer to the code. This part of the network can be seen as an independent network and is called *Encoder*. Conversely, the hidden layers that connect the code to the output layer contain a number of units that progressively increases as they get closer to the output layer; this part of the network is called *Decoder*. Encoder and decoder are often represented as functions, and, if I is the input of the network, the output is represented as $d(e(I))$.

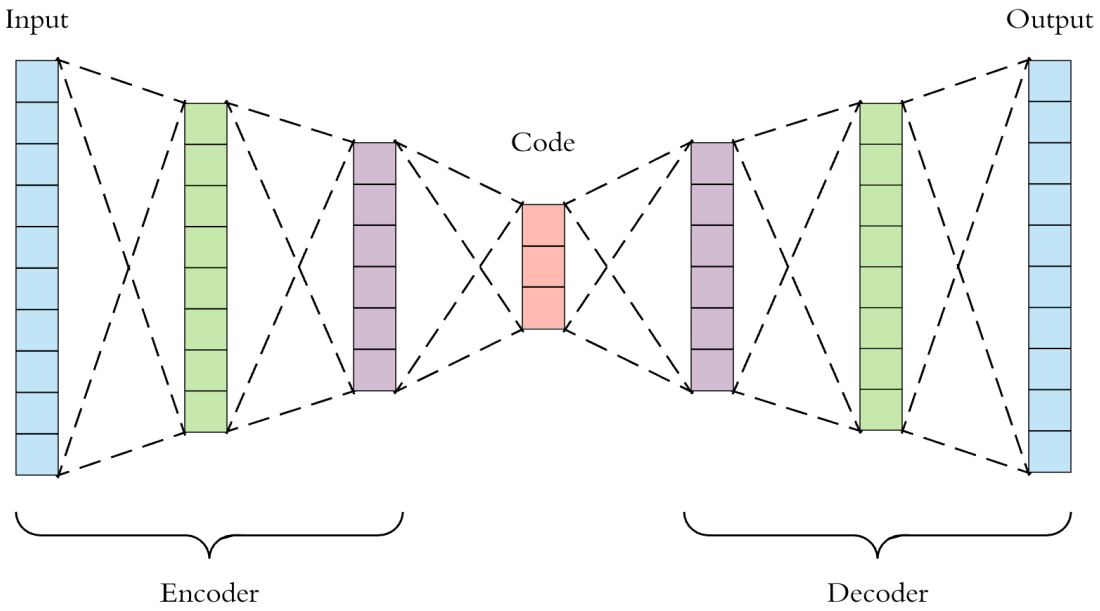

Figure 2: Standard model of an Autoencoder.

In these networks the central layer acts as a bottleneck as regards the quantity of representable information, and forces the network to *learn synthetic representations* of the input points. Of course, if the input data are not excessively redundant, this synthetic representation will imply a certain loss of information.

3.1 Implementation

For this project an autoencoder with the following architecture has been realized:
Encoder:

- Input layer (128, 128, 3)
- 2D Convolution (128, 128, 8)
- Max Pooling
- 2D convolution (64, 64, 16)
- Max Pooling
- 2D convolution (32, 32, 16).

Decoder:

- 2D convolution (32, 32, 16).
- Upsampling
- 2D convolution (64, 64, 16)
- Upsampling
- 2D Convolution (128, 128, 8)
- Output layer (128, 128, 3)

The autoencoder has been trained for a total of 40 epochs on the train set of the HR Images dataset, consisting of 800 PNG images. For the validation, the validation set included in the same dataset has been used. As regards the design choicees of the network, the block size of 128x128 has been chosen in order to be able to deal with images of any size, after ad-hoc padding. The 128x128 size looked the best compromise between being able to work with small enough blocks, in order to avoid a heavy padding, and big enough blocks in order to avoid to add a significant computational overhead due to a too big number of them for an average image. As regards the number of layers and the number of feature maps, choices have been made in order to have a network complex enough in order to avoid significant loss of information, but small enough in order to be able to train it on common hardware in reasonable times.

4 Compressor

The compressor takes as input a PNG image and a value in the range $[0, 100]$ for the quality parameter, and operates according to the following steps:

1) The alpha channel, if present, is remporarily taken apart, leaving just a matrix reoresenting the three channels of the normalized RGB color space.
2) Padding is applied to the rgb matrix, if necessary. The matrix is then split in blocks of size 128x128.
3) The blocks are compressed by the encoder.
4) The compressed blocks are given as input to the decoder, that brings them back to the original size.
5) The rgb matrix is recreated from the blocks of the previous point.
6) The decompression error is estimated, an *error correction* matrix is then generated according to the quality parameter. (Further details of this step are presented in the rest of the section)
7) The error correction matrix, the original alpha channel, and a vector containing the blocks generated at step 3 are compressed losslessly through the algorithm *LZMA*.
8) The output of the previous step, together with some metadata necessary to reconstruct the image (like thee number of rows and columns) are transmitted to the decompressor.

4.1 Error correction

The error correction procedure has been briefly anticipated at the step 6 of the compression algorithm. After executing steps 1 to 5, the compressor has the original rgb matrix and the decompressed rgb matrix; it can then compute the decompression errors. In order to get the distribution of the decompression errors, instead of computing them all, that can be computationally expensive, a sampling is performed on 10000 randomly selected pixels. The value of the errors on these 10000 pixels are then substituted by their *abs* value and sorted. At this point, the parameter quality allows us to select the entity of the error correction. Formally, if $ErrorDistribution$ represents the sorted vector of the absolute values of the errors, and $error_{i,j}$ represents the value of the decompression error on the generic i, j pixel, the error will be *corrected* if and only if the following inequality holds:

$$|error_{i,j}| \geq ErrorDistribution\,[10000 - quality \times 100]$$

As an example, if the quality parameter is set to 0, the only errors to be corrected will be the ones that are bigger than the maximum of the sampled errors. If we assume that the sampled distribution is close enough to the true distribution, then we can conclude that no errors will be corrected. Conversely, if we perform a compression with the quality parameter set to 100, the error correction will correct all the errors bigger or equals to the smallest of the sampled errors; assuming, as before, that the sampled distribution is close enough to the true one, we can conclude that all the errors will be corrected, and the compression will be approximately lossless.

5 Decompressor

The algorithm of the decompressor is even more straightforward than the one of the compressor; it operates according to the following steps:

1) The metadata and the code compressed through the $LZMA$ algorithm are separated.

2) The code compressed through $LZMA$ is decompressed, recovering the blocks produced at step 3 of the compressor, the error correction matrix and the aplha channel, if present.

3) The blocks compressed by the encoder are decompressed by the decoder, and rearranged in the original disposition using the information from the metadata; thus (re-)obtaining a matrix of normalized rgb values.

4) Elementwise addition is performed between the rgb matrix and the error correction matrix.

5) The alpha channel, if present, is added to the matrix obtained at the previous point as the fourth channel.

6) The matrix obtained at the previous point is returned as the output of the decompression.

6 Hardware and Software Platforms

To build the network the library *Keras* has been used, with *tensorflow* as back-end. The entire application has been developed within the *Jupyter Lab* environment, through the *Anaconda* platform. The version of tensorflow that has been used is the *tensorflow-2.0.0-beta1*, with support for dedicated GPU.

The autoencoder has been trained on an *HP Pavilion Lapton cs1007nl*, with 16 GB RAM DDR4, *CPU Intel Core i7 8565u*, GPU *Nvidia GeForce MX 150*. During the phases of development, validation and testing, also an Apple MacBook Pro 15" with 16GB of RAM and CPU Intel Core i7, 6 core has been used.

7 Performances

Subsections 7.1, 7.2, 7.3, 7.4 refer to the Lena.png image, while subsections 7.5, 7.6 illustrate the variations of the compression ratio and PSNR score varying the quality parameter. The last subsection then compares the PSNR and compression ratio of the presented algorithm to lossy JPEG.

7.1 Graphic performances

Figure 3: Lena.png

Figure 4: Decompression quality 0

Figure 5: Decompression quality 20

Figure 6: Decompression quality 50

7.2 Temporal performances

```
Compressor STARTS
Image loaded. Elapsed: 0.022726431998307817
Image padded. Elapsed: 0.012689994997344911
Image separated into blocks. Elapsed: 5.2948002121411264e-05
Image blocks compressed through network. Elapsed: 0.30777222399774473
Blocks recompressed through LZMA. Elapsed: 1.498584289001883
Image decompressed. Elapsed: 0.10018495700205676
Proper treshold computed. Elapsed: 0.052747323003131896
Treshold is 0.2491818070411682
Relevant errors computed. Elapsed: 0.10500063699146267
Relevant errors LZMA compressed. Elapsed: 0.0842733710014727
Compressor ENDS. Total compressor computation time is 2.186192240988021

Decompressor STARTS
Blocks LZMA decompressed. Elapsed: 0.32615076600632165
Blocks decompressed through decoder network. Elapsed: 0.10354380198987201
Network decompressed image saved in predicted/network_decompression.png. Elapsed: 0.41137154400348663
Relevant errors decompressed through LZMA. Elapsed: 0.0026222010055789724
Image error corrected. Elapsed: 0.002343069005291909
Error-corrected image saved in predicted/decompression_error_corrected.png. Elapsed: 0.425615017011296
Decompressor ENDS. Total decompressor computation time is 1.272672455990687
```

Figure 7: Decompression quality 0

```
Compressor STARTS
Image loaded. Elapsed: 0.01794108099420555
Image padded. Elapsed: 0.015530671007581986
Image separated into blocks. Elapsed: 4.420900950208306e-05
Image blocks compressed through network. Elapsed: 0.32332345699251164
Blocks recompressed through LZMA. Elapsed: 1.4526076020119945
Image decompressed. Elapsed: 0.10050469999259803
Proper treshold computed. Elapsed: 0.04979869599628728
Treshold is 0.030020478367805484
Relevant errors computed. Elapsed: 0.10040188900893554
Relevant errors LZMA compressed. Elapsed: 2.599636972998269
Compressor ENDS. Total compressor computation time is 4.6630689579906175

Decompressor STARTS
Blocks LZMA decompressed. Elapsed: 0.32499054199433886
Blocks decompressed through decoder network. Elapsed: 0.10066868500143755
Network decompressed image saved in predicted/network_decompression.png. Elapsed: 0.4170436419954058
Relevant errors decompressed through LZMA. Elapsed: 0.394801964997896
Image error corrected. Elapsed: 0.5606396300136112
Error-corrected image saved in predicted/decompression_error_corrected.png. Elapsed: 0.33847166899067815
Decompressor ENDS. Total decompressor computation time is 2.1387289029953536
```

Figure 8: Decompression quality 20

```
Compressor STARTS
Image loaded. Elapsed: 0.010478010997758247
Image padded. Elapsed: 0.014954928003135137
Image separated into blocks. Elapsed: 0.0001238879922311753
Image blocks compressed through network. Elapsed: 0.29970873599813785
Blocks recompressed through LZMA. Elapsed: 1.4969917419948615
Image decompressed. Elapsed: 0.09627247499884106
Proper treshold computed. Elapsed: 0.04983930601156317
Treshold is 0.015549078583717346
Relevant errors computed. Elapsed: 0.05324030299379956
Relevant errors LZMA compressed. Elapsed: 5.6513541269960115
Compressor ENDS. Total compressor computation time is 7.675370840006508

Decompressor STARTS
Blocks LZMA decompressed. Elapsed: 0.32145640400995035
Blocks decompressed through decoder network. Elapsed: 0.10518107100506313
Network decompressed image saved in predicted/network_decompression.png. Elapsed: 0.40711721300613135
Relevant errors decompressed through LZMA. Elapsed: 0.6981980270065833
Image error corrected. Elapsed: 1.1685739870008547
Error-corrected image saved in predicted/decompression_error_corrected.png. Elapsed: 0.37082586299220566
Decompressor ENDS. Total decompressor computation time is 3.072308709000936
```

Figure 9: Decompression quality 50

7.3 Compression ratio performances

```
compression_ratio = original_size/compressed_size
original_size = size_alpha_channel + size_rgb_image
compressed_size = size_compressed_blocks_lz + size_important_errors_lz + size_alpha_channel_lz
2.767896313698576
```

Figure 10: Decompression quality 0

```
compression_ratio = original_size/compressed_size
original_size = size_alpha_channel + size_rgb_image
compressed_size = size_compressed_blocks_lz + size_important_errors_lz + size_alpha_channel_lz
1.34286368029397
```

Figure 11: Decompression quality 20

```
compression_ratio = original_size/compressed_size
original_size = size_alpha_channel + size_rgb_image
compressed_size = size_compressed_blocks_lz + size_important_errors_lz + size_alpha_channel_lz
0.8766554014417296
```

Figure 12: Decompression quality 50

7.4 Decoder error

```
mean error 0.007667504
error std 0.027514532
mean absolute error 0.020500632
asolute error std 0.019888796
```

Figure 13: Decoder error on input Lena

7.5 Compression Ratio

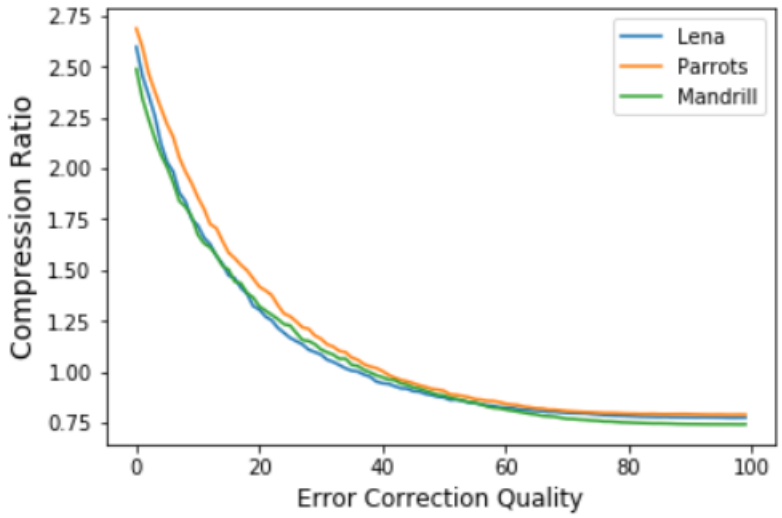

Figure 14: Compression Ratio compared to error correction quality

7.6 PSNR

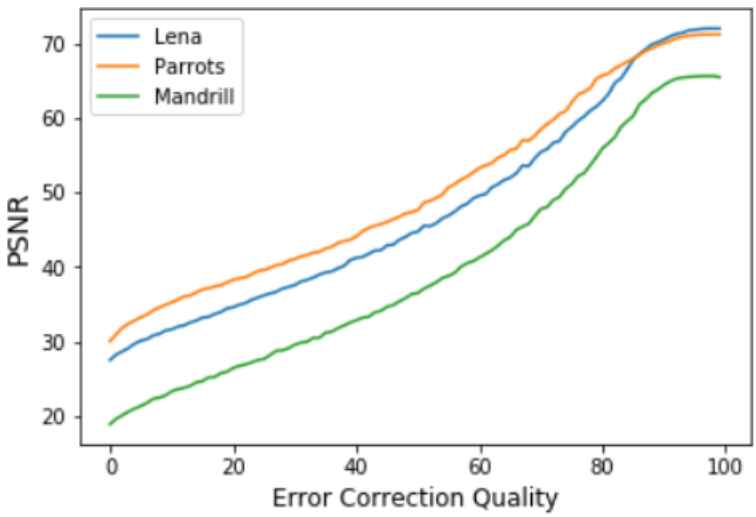

Figure 15: PSNR compared to error correction quality

7.7 Comparison with JPEG

In *Figure 16* the correlation between PSNR and Compression ratio is shown for the Lena.png image. The PSNR and compression ratio are computed for each possible value of the quality parameter. In the *Figure 17* the same plot is presented for values obtained using the lossy JPEG algorithm [3].

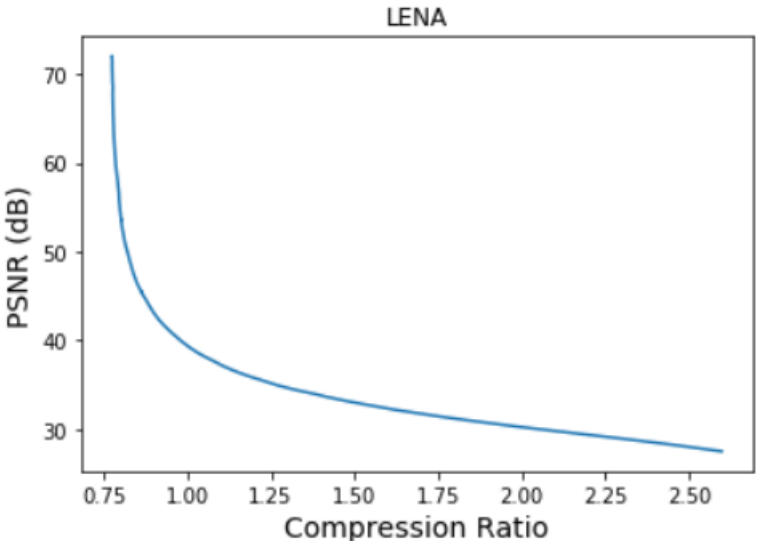

Figure 16: Our solution: PSNR compared to Compression ratio

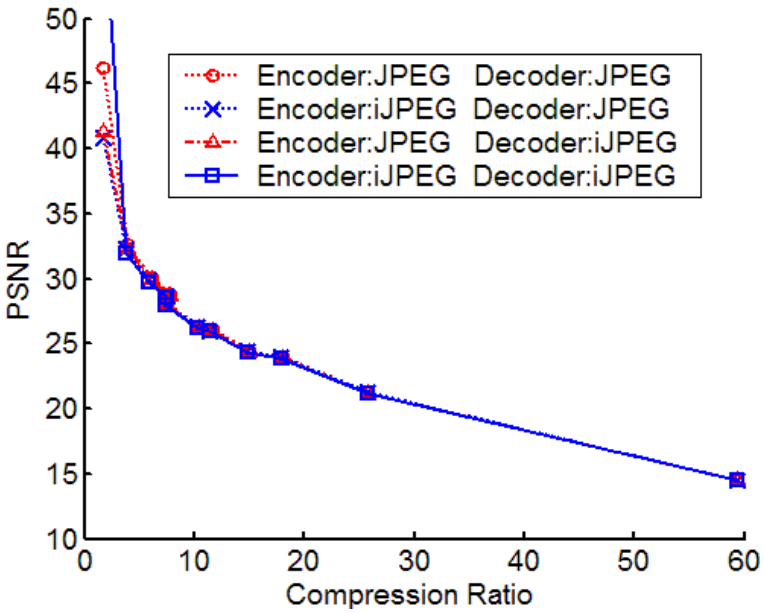

Figure 17: Lossy JPEG: PSNR compared to Compression ratio

[3]The values for the jpeg algorithm have been taken from the article Integer reversible transformation to make JPEG lossless, by Chen, Ying and Hao, Pengwei