Improving JPEG Compression Using Mutations and Deep Learning

Shahab Hamidi-Rad
shahab.hamidi-rad@technicolor.com

Technicolor AI Research Lab, 3000 El Camino Real, Unit 4, Suite 200, Palo Alto, CA 94306

Abstract

JPEG is the most popular lossy compression method both in the Internet and digital camera industry. Despite development of more efficient and robust image compression algorithms none of them has become popular enough to replace the original JPEG, mostly because of backward compatibility concerns.

In this paper we show a new method that allowed us to improve JPEG encoding algorithm for better quality and compression without touching the decoding side. Based on the standard JPEG pipeline, we define a Mutation Error Function and explain its properties. The exploitation of these properties are then combined with some deep learning techniques to develop an algorithm applied on the encoding side of the JPEG pipeline. This algorithm improves JPEG compression ratio by up to 26% without any degradation in quality. The compressed files are completely compliant with JPEG standard and can be decoded by any standard JPEG decoder.

Keywords: *JPEG, Image Compression, Mutations, Deep Learning, Convolutional Neural Network.*

1. Introduction

With popularity of mobile devices and emergence of social networking websites, very large and increasing amounts of image data is produced every second. Almost all of these images are compressed and stored in JPEG format [1]. Since its approval as a standard in 1992, several new standard formats have been introduced with better performances both in compression and quality. For example, JPEG 2000 [2] introduced at the turn of the century uses a sophisticated encoding algorithm resulting in more robust compression while preserving more details, with higher compression ratios. In 2014, Fabrice Bellard introduced BPG [3] file format that produces smaller files for a given quality and has low memory requirements.

Recently Machine Learning methods in general and Deep Neural Networks (DNNs) [4] in particular have become the de facto approach to solving different real world problems ranging from image classification [5] and voice/speech recognition [6], to playing games [7] and autonomous driving.

Success of convolutional neural networks (CNNs) in several computer vision fields, has recently motivated the efforts for CNN-based image compression approaches. However, almost all of these approaches involve either adding additional post processing units to the standard decoding pipeline or completely designing new encoding/decoding frameworks.

The first category focuses on reducing the artifacts in the reconstructed image after decoding. For example, Dong et al. [8] designed a deep convolutional network (ARCNN) to reduce the artifacts in reconstructed JPEG images. Wang et al. [9] and Guo and Chao [10] used dual-domain deep networks designed specifically based on JPEG compression knowledge again for artifact reduction after decoding the images. All de-blocking and super-resolution approaches fall in this category [11, 12, 13, 14].

On the other hand, Li et al. [15] designed a network architecture that creates and uses an importance map to perform a content-weighted compression. Ballé et al. [16] optimize a weighted sum of rate and distortion using a generalized divisive normalization (GDN) for non-linearity. Theis et al. [17] propose an image compression framework by designing an auto-encoder based on the super-resolution ideas from Shi et al [14]. Jiang et al. [18] improve the quality of the reconstructed image by adding preprocessing neural networks to the encoding process and the corresponding post processing layers after the decoding stage. Another example in this category is the work of Toderici et al. [19].

2. Related Work

As mentioned earlier despite development of more efficient image compression algorithms, JPEG is still the most popular lossy image compression standard. According to w3techs.com[1] about %73 of websites use JPEG images. The backward compatibility issue makes it

[1] W3Techs: Usage of JPEG for websites: https://w3techs.com/technologies/details/im-jpeg/all/all (August 2018)

very difficult for new algorithms to replace JPEG. For example, 16 years after its introduction and despite its universal improvements, JPEG 2000 has failed to replace JPEG. The most important problem preventing new compression methods from becoming popular is a chicken or the egg dilemma: Camera manufacturers and web sites aren't ready to accept new formats until they become widely adopted; consumers, however, aren't interested in new formats until they are widely available.

That's why our approach and the related work mentioned in this section focus on improving the JPEG encoding process without any changes to the decoding side. The work in this area started in 1993 [22] and most of the effort had been focused on improving the quantization tables. These efforts have been revived recently, as newer and more accurate psychovisual image quality assessment methods such as SSIM [23], MS-SSIM [24], FSIM [25], and Butteraugli [26] emerged. For example, Hopkins et al. [27] designed a method based on Simulated Annealing to find quantization tables that perform better than JPEG's standard quantization tables at different quality factors. Google Research recently started a new JPEG compression project called "Guetzli" [26]. They use 3 different "optimization opportunities" to improve JPEG: 1) Improving quantization tables, 2) Increasing the number of zeros to improve run-length encoding, and 3) Occasional downsampling of color components. Other approaches have tried to optimize JPEG compression for special purposes such as web deployment[2] or images fed to DNNs [29].

Unlike the works explained above we use fixed quantization tables (JPEG standard) and improve the JPEG compression only by modifying the DCT coefficient values. While "Guetzli" [26] replaces some of the DCT coefficients with zeros to improve run-length and entropy encoding, we provide a systematic method for "mutating" the DCT coefficients to optimize the weighted sum of number of bits for each coded block (rate) and the distance between the reconstructed and original block (distortion). As a result we designed and implemented a JPEG standard-compliant and backward-compatible encoding algorithm (Called "Mutated JPEG") that improves the compression by up to 26 percent[3] without degrading the image quality. As a future exploration work, we believe combining the ideas from this paper with the ones used in [26] and [27] could improve the compression performance even more.

3. JPEG Compression

The standard JPEG encoding pipeline (Fig. 1) consists of converting image colorspace to one luma (Y) and two chroma (Cr/Cb) channels, breaking up each channel to blocks, transforming each block to frequency domain (DCT), quantizing the results by element-wise devision to the quantization tables and rounding, followed by lossless entropy coding. The decoding pipeline does the corresponding inverse processes to reconstruct the original image. At the end of decoding process, the values at the output of the inverse DCT transformation may be less than 0 or more than 255. These values are clipped to make sure the results are between 0 and 255.

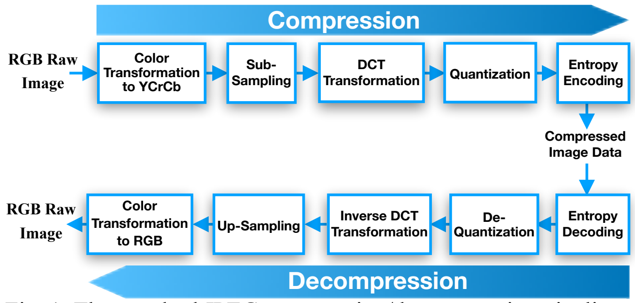

Fig. 1: The standard JPEG compression/decompression pipeline.

We performed all experiments using only baseline sequential color JPEG images without sub-sampling the color components; however the same methods can be applied to other cases with progressive and sub-sampling features. Based on this JPEG pipeline, we define the following functions that are used in the rest of this paper:

Original Input Block is a matrix B_o in $\mathbb{Z}^{8x8}$ representing a block in one of the Y/Cr/Cb components. The elements of B_o are integers between 0 and 255 inclusive.

Encoding Transform Function represents the encoding process:

$$T_{Enc}(B): \mathbb{Z}^{8x8} \rightarrow \mathbb{R}^{8x8} \qquad (1)$$
$$T_{Enc}(B) = DCT(B-128) \div Q$$

where DCT is 2D Discrete Cosine Transform, Q is the quantization table, and subtraction and division operations are element-wise. We also define the **Encoded Original Block** as $X_o = T_{Enc}(B_o)$.

Decoding Transform Function represents the decoding process:

$$T_{Dec}(X): \mathbb{R}^{8x8} \rightarrow \mathbb{Z}^{8x8} \qquad (2)$$
$$T_{Dec}(X) = RnClp(IDCT(X*Q)+128)$$

[2] The mozjpeg project, Mozilla Research, https://research.mozilla.org/2014/03/05/introducing-the-mozjpeg-project/ (2014)

[3] Please refer to our best results included in the Supplementary Materials.

where IDCT is inverse DCT transform, multiplication and addition are element-wise, and the “RnClp” function clips all matrix elements between 0 and 255 and then rounds them to the nearest integer.

Block Reconstruction Error represents the reconstruction error when matrix X is used to reconstruct the original block:

$$E(X): \mathbb{R}^{8x8} \rightarrow \mathbb{R} \qquad (3)$$
$$E(X) = MSQ(T_{Dec}(X)-B_o)$$

where MSQ(A) is the mean square of all elements in the matrix A.

The block reconstruction error for any block B in standard JPEG algorithm is given by E(round(T_{Enc}(B))). Based on the above definitions, it is obvious that:

$$T_{Dec}(\ T_{Enc}(B)\ ) = B \qquad (4)$$
$$E(X_o) = E(\ T_{Enc}(B_o)\ ) = 0 \qquad (5)$$

Since the Zigzag reordering in JPEG pipeline does not effect the values of these functions, we ignore it in our mathematical equations.

4. Rounding Problem

To get the best quality, we want to minimize the Block Reconstruction Error for every block of the image. As mentioned above, the block reconstruction error would be zero if we could use the real values of the encoded block (i.e. if $X \in \mathbb{R}^{8x8}$). Unfortunately, entropy coding requires integer values as input. The standard JPEG uses the “Round” function to move X to integer domain. However, for the best results, we need to solve the following Integer Non-Linear Programing problem:

$$\text{Minimize: } E(Z) \text{ subject to } Z \in \mathbb{Z}^{8x8} \qquad (6)$$

Adding integrality constrains to the linear programing in most cases results in increased complexity. Integer Linear Programing has been studied for a long time and solutions such as Branch and Bound can be used to solve them in polynomial time by using the relaxation of the Integer Programing (IP) to Linear Programing (LP). Integer Non-Linear Programing is more complex. Hübner et al. [30] define a “Rounding Property” for a function and prove that if a function has rounding property, you can find the solution to the relaxed non-linear problem in $\mathbb{R}^n$ and then round the results to get the solution in $\mathbb{Z}^n$.

The Block Reconstruction Error function E(X) does not have the “Rounding Property” (As this is shown later simply by counterexamples). This means rounding the optimal solution X* in $\mathbb{R}^{8x8}$ does not always result in the optimal solution Z* in $\mathbb{Z}^{8x8}$.

The diagram in Fig. 2 shows a 1-D example where rounding the real optimal solution in $\mathbb{R}$ does not results in the integer optimal solution in $\mathbb{Z}$.

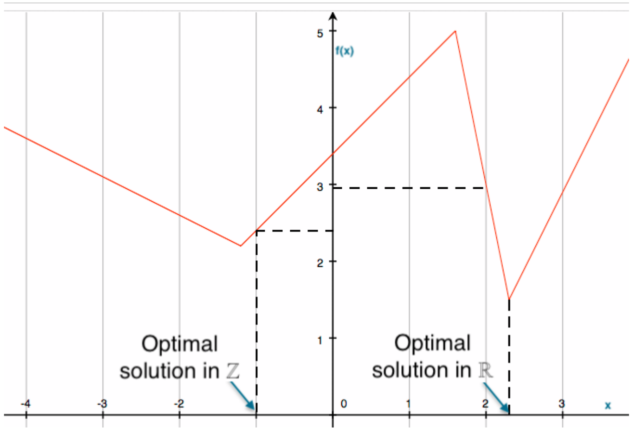

Fig. 2: The optimal solution in real domain is 2.3 which results in the global minimum (≈1.5). Rounding it gives the integer value 2, which results in: f(2)≈3. In this example the optimal solution in the integer domain is -1 because: f(-1)≈2.4 which is less than f(2)≈3.

5. Mutation Error Function

Our method of finding the optimal integer solution for (6) is by adding small “mutations” - real 8x8 matrixes - to the encoded original block X_o before rounding. Now the question is which mutation results in minimum error? Here is one idea from Fig. 2: What if we could get a mutation that moves us near a local minimum of the error function? After all there is a much higher chance of the optimal integer solution happening somewhere near a local minimum than any other random point.

We define the **Mutation Error Function** as:

$$E_{Mut}(M): \mathbb{R}^{8x8} \rightarrow \mathbb{R} \qquad (7)$$
$$E_{Mut}(M) = E(X_o + M)$$

where M is an 8x8 mutation matrix that is applied to the Encoded Original Block X_o.

Using (5), we can see that $E_{Mut}(M)$ has a global minimum at M=0 because $E_{Mut}(0)=E(X_o)=0$. After studying this function, we noticed that first, $E_{Mut}(M)$ has many local minima; and second, the locations of these local minima are almost independent of the original block.

As M moves away from the global minimum, $E_{Mut}(M)$ increases. However, since $E_{Mut}(M)$ is a mixture of non-linear functions such as clipping and rounding, the increase is not always monotonic which means $E_{Mut}(M)$ has many local minima. As you can see in Fig. 3, as we move slightly away from the global minimum, the shape of function changes drastically which results in many local minima. In the integer domain, these local minima provide opportunities similar to Fig. 2 where an optimal solution in the integer domain can happen away from the global minimum in the real domain.

One interesting property of the mutation error function is that the shape of function and location of its local minima only depends on the quantization matrix and is almost independent of the actual block values. To show this independence, we can rewrite the function using the definitions in section 1:

$$\begin{aligned} E_{Mut}(M) &= E(X_o + M) = MSQ(T_{Dec}(X_o + M) - B_o) \\ &= MSQ(RnClp(IDCT((X_o + M)*Q) + 128) - B_o) \\ &= MSQ(RnClp(IDCT(X_o*Q) + IDCT(M*Q) + 128) - B_o) \\ &= MSQ(RnClp(B_o + IDCT(M*Q)) - B_o) \qquad (8) \end{aligned}$$

Now suppose b is an integer between 0 and 255. For any small δ, we can use the following approximation for rounding and clipping:

$$RnClp(b+\delta) \approx b + Round(\delta) \qquad (9)$$

The equation above is exact for most cases. The approximation happens only when $|\delta|>0.5$ *and* b is near the range boundaries 0 and 255.

Back to equation (8), since B_o contains only integers between 0 and 255 (By definition), and IDCT(M*Q) is small (this can easily be derived from $M<<X_o$), we can plug this approximation into (8):

$$\begin{aligned} E_{Mut}(M) &= MSQ(\ RnClp(B_o + IDCT(M*Q)) - B_o\ ) \\ &\approx MSQ(\ B_o + Round(IDCT(M*Q)) - B_o\ ) \qquad (10) \end{aligned}$$

Therefore:

$$E_{Mut}(M) \approx MSQ(Round(IDCT(M*Q))) \qquad (11)$$

This means $E_{Mut}(M)$ is almost independent of the block values. In other words the locations of local minima of $E_{Mut}(M)$ depend only on the quantization matrix. Fig. 3 shows a verification of equation (11).

Now suppose we somehow acquired a list of locations of local minima of $E_{Mut}(M)$ for a specific quantization matrix. Because of the rounding phenomena highlighted in Fig. 2, there is a good chance that moving to one of these local minima before rounding, could result in: a) less number of bits in the coded block, or b) lower block reconstruction error (or both).

6. Brute Force Algorithm

As in any other compression method, there is always a tradeoff between rate and distortion. So we define a reconstruction cost function C(X) for each encoded block X:

$$C(X) = N(X) + \lambda.E(X) \qquad (12)$$

Where N(X) is the number of bits in the entropy-coded block, and E(X) is the Block Reconstruction Error defined in (3). The parameter λ is used to optimize the encoding for compression or quality. Generally, higher values of λ result in improved quality while lower values cause better compression.

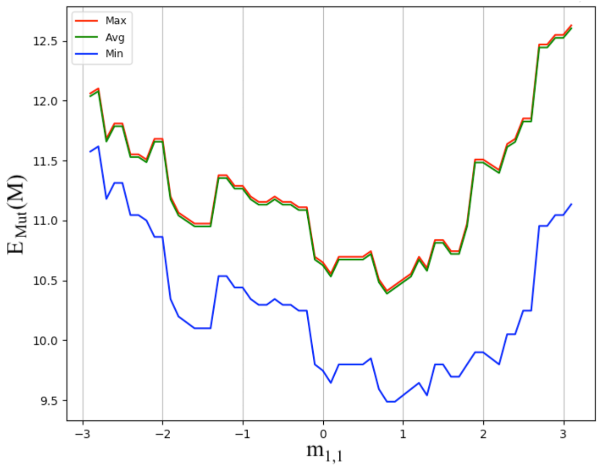

Fig. 3: The mutation error function $E_{Mut}(M)$ near the global minimum (at M=0) as a function of 10th element ($m_{1,1}$ in the 8x8 matrix) when the other elements are fixed at 0.1. The diagrams show the minimum, average, and maximum of the block errors for about 2700 different blocks of an image. These diagrams are based on the standard JPEG quantization matrix for luma (Y) with a quality factor of 90. The fact that average is very close to max proves that there is very little variation in $E_{Mut}(M)$ for different blocks (B_o) as shown in (11)

The algorithm 1 uses a list $\mathscr{M}$ of local minima of $E_{Mut}(M)$ for a specific quality factor to compress an image. In line 4, we try every mutation M from the list $\mathscr{M}$ and choose the one that results in the lowest cost as defined in (12).

Algorithm 1: Brute Force JPEG compression using mutations

```
1  $\mathscr{M}$: A list of local minima of $E_{Mut}(M)$ for the given quality factor.
2  **foreach** component in [Y, Cr, Cb] **do**
3      **foreach** block $B_o$ in component **do**
4          $M^* = \underset{M}{\text{argmin}}\ C(\ Round(T_{Enc}(B_o)+M)\ ) \quad \forall M \in \mathscr{M}$
5          $X = Round(\ T_{Enc}(B_o) + M^*)$
6          Pass X to entropy coding stage
```

7. GPU Based Algorithm

Unfortunately the brute force algorithm is very slow. That is because of extensive computational load for calculating the block error and number of bits in the entropy-coded blocks. For example, it takes about 2 hours on a MacBook Pro to compress the “Fruits” image in Fig. 6. This makes the brute force algorithm impractical for most applications.

For each block, the number of bits in the entropy-coded block needs to be calculated ‘m’ times (m: number of mutations). The only way to know the number of bits is to actually do the entropy coding and then count the number of bits in the coded block.

We can improve the speed significantly by running the algorithm on powerful GPUs. This allows us to parallelize the process and work on multiple blocks at the same time. Unfortunately porting the entropy coding (calculating number of bits) to GPU platforms such as TensorFlow is not straightforward.

To solve this problem we trained deep convolutional neural networks to predict the number of bits needed for entropy coding of a block. Our goal here is to solve a regression problem: The input is a 64-D vector representation of a mutated 8x8 block and the output is the number of bits in the entropy-coded block. After trying several network configurations it was seen that networks with 1-D convolutional layers provide the best accuracy for the regression problem. The success of 1-D convolutional network can be attributed to the fact that in entropy coding there is a correlation between neighboring elements of input vector because of run-length encoding.

Since in JPEG standard, different Huffman tables are used for luma and chroma, we need to train 2 different networks. In practice we used the same network structure (Fig. 4) for both luma and chroma but trained them separately with different sets of sample blocks. As an example, for quality factor 80, we used 1,107,187 training and 123,021 test samples created from 218 images and the mean absolute error (MAE) over test samples after rounding the output was 0.369 for luma (Y) network and 0.028 for chroma (CrCb).

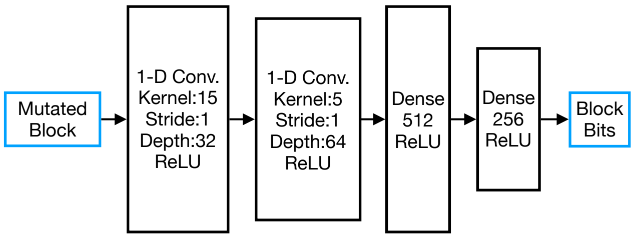
Mutated Block
1-D Conv. Kernel:15 Stride:1 Depth:32 ReLU
1-D Conv. Kernel:5 Stride:1 Depth:64 ReLU
Dense 512 ReLU
Dense 256 ReLU
Block Bits

Fig. 4: The Neural Network configuration used to estimate the number of bits in the entropy-coded blocks. Total Network Parameters: 2,240,065.

After training the networks with many samples, we can now plug them in our GPU based pipeline for the encoding process. Fig. 5 summarizes the final GPU based pipeline that can be used to compress images much faster than the brute force algorithm. As you can see instead of working serially on every block, we process one block-column of the image at a time. For example, for an 800x800 image, a column of 800x8 (100 8x8 blocks) is processed as one batch with a total of 100 batches to complete the process for each luma/chroma component of the image.

8. Mining Mutations

As explained above the mutation error function $E_{Mut}(M)$ has many local minima which is seen in Fig. 3 for one dimension. In 64-D space, shape of this function becomes extremely complex which results in many more local minima. Our optimization method in this paper depends on having a list of these local minima. Unfortunately, there is no straightforward method of finding all these local minima; mostly because $E_{Mut}(M)$ is a non-differentiable piecewise function. Our approach is inspired by the “Particle Swarm Optimization” [31], with 2 major differences: first, we are looking for local minima instead of the global minimum therefore we don’t share information between particles, and second, each particle’s movement in our space is defined by coordinate descent. Both particle swarm and coordinate descent algorithms are known to work well with non-differentiable functions. Once the mutations are found, they can be reused again and again to compress different images.

This section explains the process of “mining” local minima that are applied as mutations to the DCT coefficients before rounding and entropy coding. This process is made up of the following four stages:

8.1. Initial Vectors

The potential mutations are initialized with a uniformly distributed random 64-D vector from a region near the global minimum of $E_{Mut}(M)$ at M=0. We tried different types of bounds for the region and used the one that resulted in most number of mutations in least amount of time. For each block B_o, this region contains all points M_0 with L1-norm less than or equal to L1-norm of encoded original block:

$$||M_0||_1 \leq ||X_o||_1 \tag{13}$$

where: $X_o = T_{Enc}(B_o)$. The L1-norm here encourages sparsity and smaller values for the mutated encoded blocks ($X_o + M$), both of which result in better entropy coding compression. For each block of every image, we create several initial vectors (swarm of particles)

8.2. Coordinate Descent

For each initial vector M_0, we start a coordinate decent on the mutation error function $E_{Mut}(M)$ to reach to a local minimum. In practice we do one coarse coordinate descent (High particle speed) followed by another one with finer resolution (decelerating particles to stop at local minimum).

The result of coordinate descent for each particle is kept as a candidate mutation only if:

1. It is not too close to an existing local minimum already found.

2. It is not too close to the global minimum at M=0.
3. The rounded mutated block error is less than a predefined margin:

$$E(\text{Round}(X_o + M)) < E_{max} \qquad (14)$$

Where E_{max} is based on average block reconstruction errors for standard JPEG encoding with a specific quality factor.

8.3. Repeating and merging

In practice we ran this program in parallel processes on many images to get several lists of candidate mutations. These lists are then merged to get a large list of candidate mutations. During the merging process we drop a mutation if it is too close to an existing one from a different list.

8.4. Filtering out rarely used mutations

We use the merged list of mutations to compress many image files counting the number of times each mutation in the list reduces the reconstruction error. The mutations with highest success rates are then kept as our final list of mutation.

The mining process is a very time-consuming task. As an example it takes about 5 days to mine mutations from 130 images (with image sizes around 512x340) on a server with 36 cores running 3.0 GHz Intel Xeon processors (AWS c5.9xlarge instances)

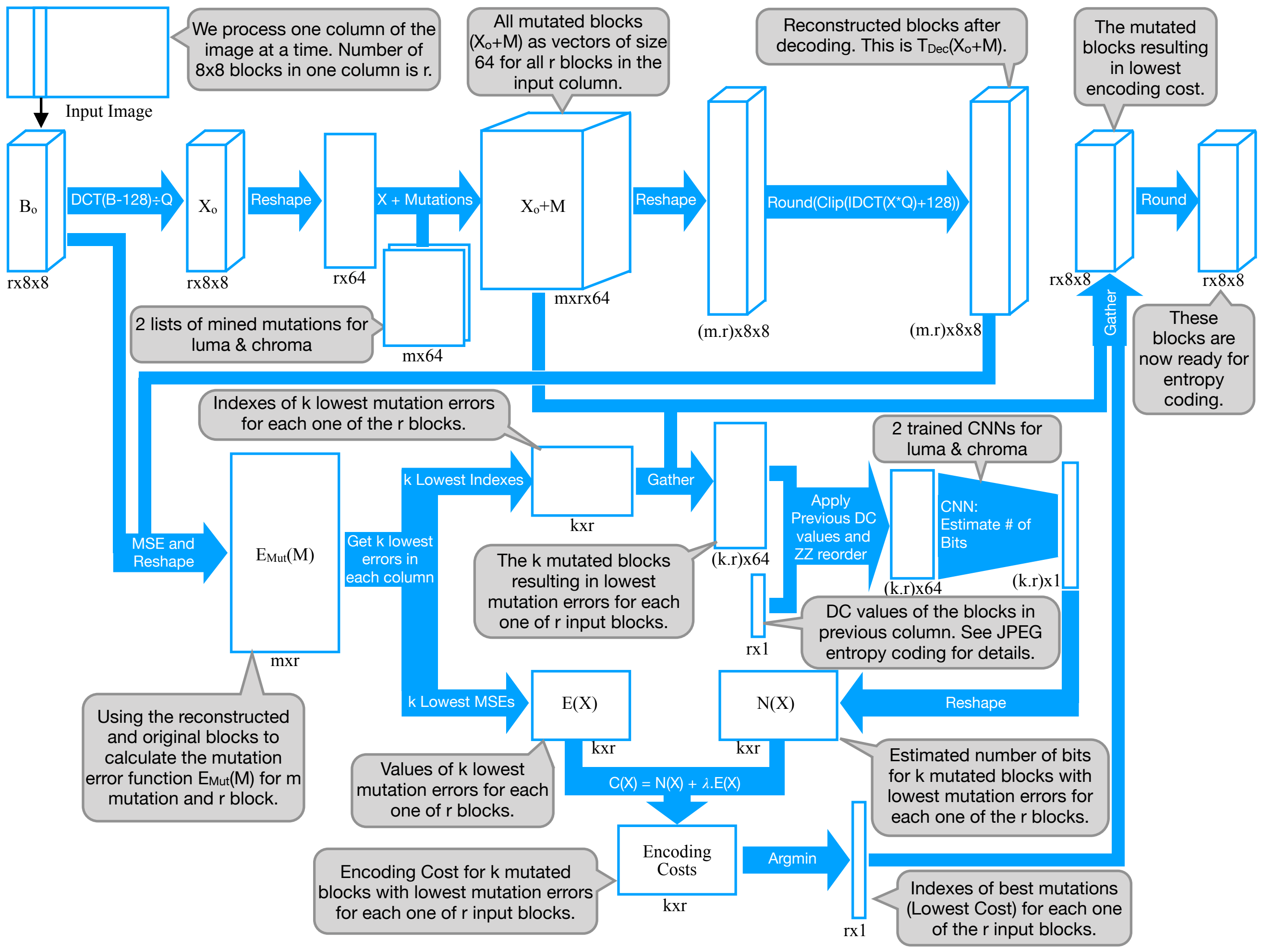
We process one column of the image at a time. Number of 8x8 blocks in one column is r.
All mutated blocks (X₀+M) as vectors of size 64 for all r blocks in the input column.
Reconstructed blocks after decoding. This is T_Dec(X₀+M).
The mutated blocks resulting in lowest encoding cost.
Input Image
B₀
rx8x8
DCT(B-128)÷Q
X₀
rx8x8
Reshape
rx64
X + Mutations
X₀+M
mxrx64
mx64
Reshape
(m.r)x8x8
Round(Clip(IDCT(X*Q)+128))
(m.r)x8x8
rx8x8
Round
rx8x8
Gather
These blocks are now ready for entropy coding.
2 lists of mined mutations for luma & chroma
Indexes of k lowest mutation errors for each one of the r blocks.
MSE and Reshape
E_Mut(M)
mxr
Get k lowest errors in each column
k Lowest Indexes
kxr
Gather
(k.r)x64
2 trained CNNs for luma & chroma
Apply Previous DC values and ZZ reorder
(k.r)x64
CNN: Estimate # of Bits
(k.r)x1
The k mutated blocks resulting in lowest mutation errors for each one of r input blocks.
rx1
DC values of the blocks in previous column. See JPEG entropy coding for details.
Using the reconstructed and original blocks to calculate the mutation error function E_Mut(M) for m mutation and r block.
k Lowest MSEs
E(X)
kxr
N(X)
kxr
Reshape
C(X) = N(X) + λ.E(X)
Values of k lowest mutation errors for each one of r blocks.
Estimated number of bits for k mutated blocks with lowest mutation errors for each one of the r blocks.
Encoding Costs
kxr
Argmin
rx1
Encoding Cost for k mutated blocks with lowest mutation errors for each one of r input blocks.
Indexes of best mutations (Lowest Cost) for each one of the r input blocks.

Fig. 5: GPU based compression pipeline. This pipeline works on a single “block-column” at a time. First, the encoding transformations are applied to the blocks in the column. All the mutations are added to all encoded blocks to form the 3D tensor (X_o+M). It is then used to calculate the Mutation Error Function $E_{Mut}(M)$. To make this process faster when the number of mutations is large, we use only the top K mutations with lowest errors. The selected mutated blocks are then fed to the CNN to estimate the number of bits N(X). The cost for each mutated block is then calculated and the blocks with lowest costs are selected to be passed to the entropy coding stage. The rest of encoding process is exactly similar to the JPEG standard.

Also note that since there are 2 different quantization matrixes for luma and chroma components of the image, we need to mine 2 sets of mutations.

At the end of the mining process we get 2 sets of mutations for luma and chroma components. Each set can be represented by an mx64 matrix where m is the number of mutations found.

This process must be repeated once for every specific quantization matrix. Since the JPEG quality factor modifies the quantization matrix, we mine new sets of mutations for different quality factors. Once we have these lists of mutations, we can compress any image using the same list of mutations.

9. Experimental Results

For different stages of mining, training, and parameter searches, we used a raw image data set consisting of 218 uncompressed images downloaded from different websites. The images cover a wide range of different categories and imaging conditions.

We also used 10 raw images from the publicly available IVC dataset [32] for evaluation purposes. These images were not used in any stage of mining or training. Fig. 6 shows a side-by-side comparison of the compression and quality results as Standard JPEG and our algorithm (Mutated JPEG) were used to compress one of these test images at quality factor 80.

To evaluate the quality performance we used SSIM [23] which gives a measure of similarity between the reconstructed JPEG image and the original raw image. For comparison, we also included other error measures such as MSE and PSNR. For rate performance evaluation we use the JPEG file size and bit per pixel (BPP). Fig. 7 shows the rate-distortion comparison between our algorithm and standard JPEG.

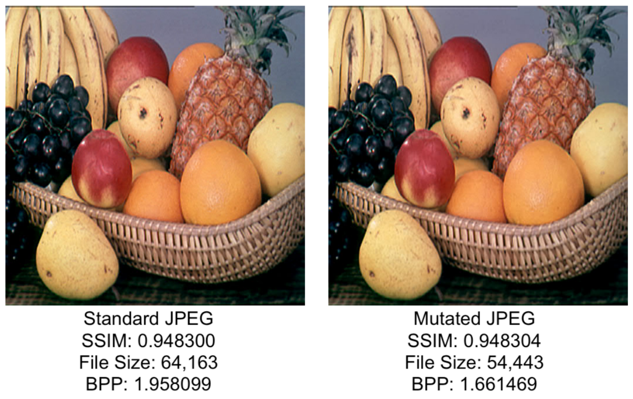

Fig. 6: Side by side comparison of a compressed sample image (from our test image set). The image was compressed with a quality factor of 80. Our method shows an improvement of more than %15 in compression ratio without degrading quality.

We used Tensorflow framework for the implementation of the convolutional neural networks and entire encoding pipeline.

Using the mining method explained in previous section, we created different sets of mutations for different JPEG quality factors. We used about 130 images during the mining process and then merged and filtered them using 218 images.

We used all 218 images to create training and test datasets and used them to train 2 CNN models for each quality factor.

We then implemented a parameter search algorithm and used it with 218 images to find the best values of λ (optimized for compression ratio) in equation (12) for different JPEG quality factors.

Table 1 compares the average results of compression for the 10 test images at different quality factors. The λ

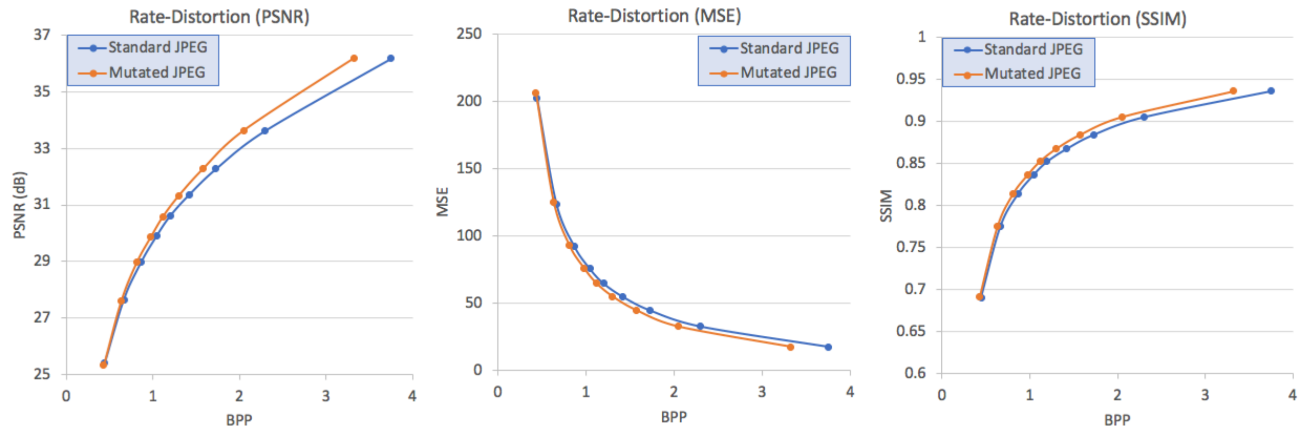

Fig. 7: The rate-distortion charts with PSNR (left), MSE (middle), and SSIM (right) vs. bits per pixel (BPP). These values are calculated at 9 different quality factors (10 .. 90). As you can see the improvement in compression increases as the quality factor increases.

Quality Factor	Standard JPEG		Mutated JPEG	
	BPP	SSIM	BPP	SSIM
10	0.44446	0.69070	0.43181	0.69161
20	0.66463	0.77481	0.63345	0.77496
30	0.86674	0.81379	0.81258	0.81404
40	1.04196	0.83615	0.97574	0.83627
50	1.19712	0.85212	1.11477	0.85220
60	1.41758	0.86716	1.29916	0.86725
70	1.72878	0.88412	1.57602	0.88421
80	2.29902	0.90508	2.04648	0.90515
90	3.74981	0.93602	3.32551	0.93605

Table 1: The average compression (BPP) and quality (SSIM) values over 10 test images for standard JPEG and Mutated JPEG at different quality factors.

values for the mutated JPEG algorithm at each quality factor were adjusted to get almost the same SSIM value as the Standard JPEG. This means the algorithm was optimized for compression. As you can see in all cases the SSIM values are slightly better than standard JPEG while the BPP values are reduced significantly compared to Standard JPEG.

Fig. 8 shows the compression improvement at different quality factors.

We also tried the Mutated JPEG algorithm on a variety of raw images downloaded from the internet. Our best results (with up to %26 improvement in compression) are included in the Supplementary Materials accompanying this paper.

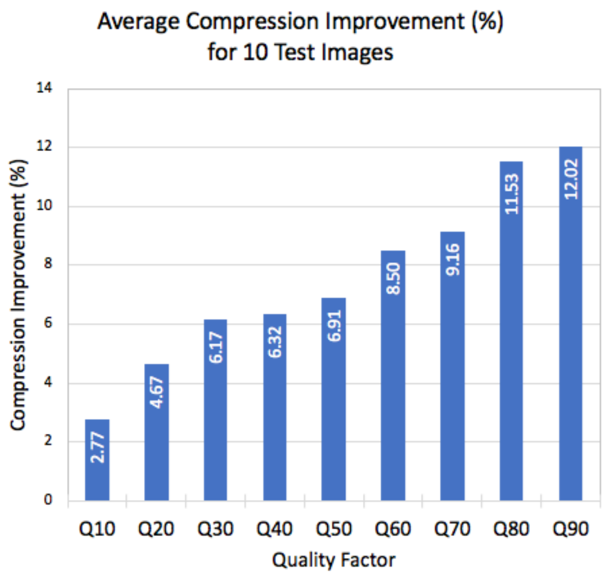

Fig. 8: Average Compression improvement on 10 test images at different quality factors. The λ values for luma and chroma components were adjusted for best compression ratio while keeping the quality (average SSIM values) slightly better than the standard JPEG.

10. Conclusion

We have shown that applying small mutations to the DCT coefficient values in JPEG encoding pipeline can result in better compression ratios and better quality. We also have provided a systematic method to efficiently find the best mutations and incorporate them in a new GPU based JPEG encoding pipeline. We have also shown how we can use convolutional neural networks to estimate the number of bits in an entropy-coded block and improve the performance of the encoding pipeline.

In future work we would like to investigate the performance improvements by combining the methods of this paper with those of [27] and [28]. We are also interested in applying the concept of mutations to modern video encoding standards such as AVC and HEVC.

References

[1] W.B. Pennebaker and J. L. Mitchell. *JPEG: Still image data compression standard*. Springer Science & Business Media. (1992)

[2] A. Skodras, C. Christopoulos, and T. Ebrahimi. *The JPEG 2000 still image compression standard.* IEEE Signal processing magazine 18, no. 5, pp. 36-58. (2001)

[3] F. Bellard. *BPG Image format.* BPG website: https://bellard.org/bpg/. (2014)

[4] Y. LeCun, Y. Bengio, and G. Hinton. *Deep learning.* Nature, volume 521, pp. 436–444. (2015)

[5] A. Krizhevsky, I. Sutskever, and G. E. Hinton. *ImageNet Classification with Deep Convolutional Neural Networks.* In NIPS, pp. 1097–1105. (2012)

[6] W. Gevaert, G. Tsenov, and V. Mladenov. *Neural networks used for speech recognition.* Journal of Automatic Control. 20. 10.2298/JAC1001001G. (2010)

[7] D. Silver and D. Hassabis. *AlphaGo: Mastering the ancient game of Go with Machine Learning.* Google AI Blog. (2016)

[8] C. Dong, Y. Deng, C. C. Loy, and X. Tang. *Compression Artifacts Reduction by a Deep Convolutional Network.* ICCV. (2015)

[9] Z. Wang, D. Liu, S. Chang, Q. Ling, Y. Yang, and T. S. Huang. *D3: Deep dual-domain based fast restoration of jpeg-compressed images.* Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition, pages 2764–2772. (2016)

[10] J. Guo and H. Chao. *Building dual-domain representations for compression artifacts reduction.* ECCV. Springer, Cham. pp. 628–644. (2016)

[11] C. Dong, C.C. Loy, K. He, and X. Tang. *Learning a deep convolutional network for image super-resolution.* ECCV, pp. 184–199. (2014)

[12] C.Y. Yang, C. Ma, and M.H. Yang. *Single-image super-resolution: A benchmark.* ECCV, pp. 372–386. (2014)

[13] C. Wang, J. Zhou, and S. Liu. *Adaptive non-local means filter for image deblocking*. Signal Processing: Image Communication, 28(5): pp. 522–530. (2013)

[14] W Shi, J. Caballero, F. Huszár, J. Totz, A.P. Aitken, R. Bishop, D. Rueckert, and Z. Wang. *Real-time single image and video super-resolution using an efficient sub-pixel convolutional neural network.* Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition, pp. 1874-1883. (2016)

[15] M. Li, W. Zuo, S. Gu, D. Zhao, and D. Zhang. *Learning convolutional networks for content-weighted image compression.* arXiv preprint arXiv:1703.10553. (2017)

[16] J. Ballé, V. Laparra, and E.P. Simoncelli. *End-to-end optimized image compression.* arXiv preprint arXiv:1611.01704. (2016)

[17] L. Theis, W. Shi, A. Cunningham, and F. Huszár. *Lossy image compression with compressive autoencoders.* arXiv preprint arXiv:1703.00395. (2017)

[18] F. Jiang, W. Tao, S. Liu, J. Ren, X. Guo, and D. Zhao. *An end-to-end compression framework based on convolutional neural networks.* IEEE Transactions on Circuits and Systems for Video Technology. (2017)

[19] G. Toderici, S.M. O'Malley, S.J. Hwang, D. Vincent, D. Minnen, S. Baluja, M. Covell, and R. Sukthankar. *Variable rate image compression with recurrent neural networks.* arXiv preprint arXiv:1511.06085. (2015)

[22] D.M. Monro, and B.G. Sherlock. *Optimum DCT quantization.* In IEEE Data Compression Conference, pp. 188-194. (1993)

[23] Z. Wang, A.C. Bovik, H.R. Sheikh, and E.P. Simoncelli. *Image quality assessment: from error visibility to structural similarity.* IEEE transactions on image processing 13, no. 4 pp. 600-612. (2004)

[24] Z. Wang, E.P. Simoncelli, and A.C. Bovik. *Multiscale structural similarity for image quality assessment.* In IEEE The Thrity-Seventh Asilomar Conference on Signals, Systems & Computers, vol. 2, pp. 1398-1402. (2003)

[25] L. Zhang, L. Zhang, X. Mou, and D. Zhang. *FSIM: a feature similarity index for image quality assessment.* IEEE transactions on Image Processing 20, no. 8, pp. 2378-2386. (2011)

[26] J. Alakuijala, R. Obryk, O. Stoliarchuk, Z. Szabadka, L. Vandevenne, and J. Wassenberg. *Guetzli: Perceptually Guided JPEG Encoder.* arXiv preprint arXiv:1703.04421. (2017)

[27] M. Hopkins, M. Mitzenmacher, and S. Wagner-Carena. *Simulated annealing for jpeg quantization.* arXiv preprint arXiv:1709.00649. (2017)

[29] Z. Liu, T. Liu, W. Wen, L. Jiang, J. Xu, Y. Wang, and G. Quan. *DeepN-JPEG: a deep neural network favorable JPEG-based image compression framework.* arXiv preprint arXiv:1803.05788. (2018)

[30] R. Hübner and A. Schöbel. *When is rounding allowed? A new approach to integer nonlinear optimization.* In ORP3 MEETING, CA DIZ, September, pp. 13-17. (2011)

[31] J. Kennedy. *Particle swarm optimization.* In Encyclopedia of machine learning, pp. 760-766. Springer, Boston, MA. (2011)

[32] P. L. Callet and F. Autrusseau. *Subjective Quality Assessment: IVC Database.* (2006)