An Implementation of Autoencoding Neural Networks to the JPEG Compression Stack

Carson Cummins ‘24

JPEG compression standard has been in use since 1994[1], and used on millions of devices worldwide. Its ubiquitous nature has led to a wealth of research on its applications - in fact its official website lists 15 separate groups dedicated to novel applications and improvements of the existing software[1]. Its compression stack consists of several steps, but can be boiled down to three simple, generalized steps (in the grayscale case, which doesn’t include chromatic downsampling[2]): a discrete cosine transform, quantization, and lossless data compression[2]. The first step, a discrete cosine transform, interestingly enough, actually creates more information. The image is broken down into a grid of 8 pixel by 8 pixel squares, each of which is run through the discrete cosine transform (DCT)[2]. This transform converts the bytes representing the grayscale values of the image to an 8 by 8 grid of floating point numbers which each are implemented with 4 bytes. This effectively quadruples the size of the file, but it allows us to complete the next step - quantization[2]. In the quantization step, piecewise division is done with the DCT data and a matrix defined by the jpeg standard. Any remaining decimal information is then discarded by a floor operation[2]. The resulting matrix is unwound according to a zigzag pattern, and added to our final data string. This data string is run through two lossless compression algorithms - run length encoding and huffman encoding[2]. This project attempts to emulate this standard, and approve upon it. Additionally, it will offer comparisons to other published attempts to improve upon this standard.

The code created provides 2 compression modes - a reproduction of most of the standard JPEG stack in grayscale, and the same stack recreated with an additional step based on machine learning. The technical aspects of both of these modes are discussed in detail below and compared to other machine learning based compression techniques, followed by a comparison of performance, and a final discussion of improvements.

Mode 1 - JPEG Compression Stack Emulation

This program is run with the command: `python App.py`. After some time, it will produce three files - `out.aejpg`, `out_tree.json` and `dim`. The purposes of these files are discussed below. The program will then display an image, which is the decompressed version of the file, after it has been run through this version of the jpeg stack. Note that it may be necessary to install the modules in `requirements.txt` to run this code, if these modules are not already installed on your device.

Steps of Execution

1. **Chunking**

   This step is implemented quite simply. The program copies each eight by eight chunk of the image into an array of chunks to be handled later. This will be referred to as the "chunk list" for the remainder of this paper.

2. **Discrete Cosine Transform**

   This step is implemented using the python scipy library - a function from the library, dct2, is run on each chunk in the chunk list and the chunk is set to the result. However, it is worth discussing the reason for this transform's inclusion. The discrete cosine transformation allows us to isolate the frequencies of each area of an image. This can be thought of as the starkness of the changes between the individual pixels. It is known from previous scientific research that the human eye cannot recognize high frequency areas as well as low frequency regions, so we establish information about where these high frequency areas lie, so we can get rid of them in the next step.

3. **Quantization**

   In this step, we divide the frequencies created by the DCT by a quantization matrix. This matrix is defined in the file `utils.py` and contains the values specified by the JPEG standard. The most interesting part about it, however, is that its entries enlarge as they approach the bottom right hand corner of the matrix. An image in frequency space, eg. one that has gone through a DCT, has higher frequency in its bottom right hand corner. Hence, dividing these frequencies by a higher value makes them more likely to reach a number near zero, which it would be rounded down to. By having several repeating zeros in the quantized matrix, the lossless compression algorithms implemented later in the process will be significantly more effective. After this step, we would also generally utilize the standard zigzag flattening method to turn the resulting matrix into a single list of values. However, since run length encoding was chosen to not be used, this is entirely unnecessary, and a standard snaking style flattening was used. However, the implementation of zigzag flattening can be seen in the file `utils.py`.

4. **Transformation into Byte Space**

   Earlier steps have treated each individual pixel value as either an integer or floating point value. In order to write them to a file, however, it is necessary to convert the pixels to byte values - between 0 and 255. DCT will result in values less than zero, as well as possibly (although unlikely) values greater than can be represented by the range 0-255. This step, consequently, applies some simple transformations to these values. It first scales each byte by the constant defined as `max(1, 128/MAX_ABS_VALUE)`. This effectively places all values between

(-128,127). 128 is then added to all values, moving them into the range for bytes of (0,255).

5. **Lossless compression**

   In a traditional JPEG compression stack, lossless compression takes two forms: run length encoding and Huffman encoding. In this stack, solely the latter is implemented. Methods exist in the utils.py folder to implement run length encoding, but were not implemented in this stack, as they generally led to larger file sizes when the machine learning based step in the next mode was implemented, and ultimately this stack exists to support that step. The Huffman encoding step writes the results to three files: `out.aejpg`, `out_tree.json` and `dim`. The file `out.aejpg` contains the actual image information in a binary encoding. The file `out_tree.json` contains the necessary information to create the Huffman tree to decode the binary information in `out.aejpg`. The final file, `dim`, simply represents the dimensions of the image, as well as the byte scaling value that was used to squeeze the quantized DCT values into the (0,255) byte range.

This represents a fairly standard implementation, if somewhat lacking in certain areas, of the JPEG stack. Decompression is executed in the opposite order of this stack, starting with a standard Huffman decompression, a transformation from byte space back to integers, a multiplication by the quantization matrix, an inverse DCT, and finally a simple reassembly of the chunks. Because of this stack's extremely standard nature, its comparison to other techniques will not be discussed at length. However, some performance data will appear later in this paper.

Mode 2 - JPEG Compression Stack in Conjunction with Autoencoding

This version of the JPEG compression stack represents the addition of a significant step after quantization in the standard stack. It can be run with the command: `python App_AE.py`, and will produce the same filenames, and show a compressed version of the test image at the end. The first few steps remain the same in order to produce the quantized values, but the stack changes greatly after. Namely, a deep learning neural network is added to convert the 64 byte chunks into a list of 8 floating point values. Large changes are made to the byte space conversion mechanism, including the addition of a clustering mechanism to allow each byte to most efficiently represent the floating point values created by the neural network. The execution steps are described below, along with a comparison to other AI based image compression algorithms.

Steps of Execution

1. **Chunking**

The image is transformed into 8*8 chunks in the exact same way as described above. These chunks are then used to train a neural network to compress them to a vector of 8 values, if a pre-trained neural network does not exist. The mechanism through which this happens will be discussed in greater detail in subsequent steps.

2. **Discrete Cosine Transform and Quantization**

   These steps operate on the chunk list created in the first step, and behave exactly how they did in the first mode. It is important to recognize that these do still result in 8*8 chunks, same as the first mode.

3. **Autoencoding**

   In this step, a neural network built with a standard autoencoding architecture is utilized to attempt to compress the 64 value chunks down to 8 floating point values. This neural network is built with a 64 value input layer, connecting to 32 and 16 node hidden layers, each utilizing a rectified linear unit (ReLU) activation function, finally connecting to an 8 node output layer for the encoder. This architecture is reversed for the decoder, and as is standard with the autoencoder architecture, the two networks are trained in conjunction to attempt to replicate the input chunks on either side. A mean squared error (MSE) error function was used, as the values attempting to be replicated didn't follow any classical pattern for a separate activation function to be used (eg. one hot encodings leading to the use of binary cross entropy loss). This may have affected the ability of the network to accurately train the data, as it was able to train to match the data well for the first ten or so epochs, but after this it would generally become significantly worse than even a random network at the autoencoding task. A better autoencoder architecture and training procedure could significantly improve this project. The extent to which the autoencoder was able to actually exploit the lower entropy of the information being passed into it due to the DCT and quantization steps could also be explored, to see if the change in entropy is actually worth the loss of processing power.

4. **Transformation into Byte Space**

   In order to decrease the loss from attempting to convert the four byte values provided by the autoencoder into single bytes, a special clustering system was implemented to generate a mapping of byte to float values. Initially, the first 256 floating point values in the compressed image were appended to an array. Each value was assigned a byte equivalent to its position in the array. Each subsequent value was assigned the byte value of the current value in the array closest to it. The value would then be updated to reflect the average value of every value which had been assigned to the corresponding byte. The resulting array is passed to the decompression step in the `dim` file. This system works well for a simple implementation, however a more sophisticated clustering mechanism could definitely be implemented to improve the efficacy of this transformation.

5. **Lossless compression**

Lossless compression occurred in exactly the same fashion for this mode. It should be noted that it was not able to produce the same decrease in file size that it was for the first mode, indicating a higher entropy level in the autoencoded files.

Comparison to Existing Models

One of the original explorations into the utilization of machine learning in image compression entitled “Combining support vector machine learning with the discrete cosine transform in image compression”[3] was markedly similar to this model. However, it was published in 2003, and consequently did not have access to the same computing power that a standard PC has now. Most likely because of this, the authors utilized a support vector machine model rather than a neural network as their primary ML model[3]. This paper is worth mentioning as it’s model works in the frequency domain, as the model in this project does, wherein most researchers opt to work in the spatial domain to reduce the complexity of their programs. This model was able to beat the compression rates of the JPEG standard[3] - which the model presented in this project does not currently do. Additionally, it experienced significantly less quality degradation than the model in this project. This is most likely due to two factors. The first of these is the simplicity of the author’s model - they most likely did not have the issues with loss explosion after a few epochs of training because of this simplicity. Secondly, they focus on compression of the DCT coefficients themselves[3], rather than quantized DCT coefficients, like those focused on in this project. This could have allowed the model a more variable training landscape, with significantly more factors to optimize, rather than the large amount of zeros presented in quantized DCT coefficients. The original motivation of using quantized DCT coefficients - providing the neural network with an input which already has some work done on it to decrease the total amount of work done by the network - seems to have had an extremely degrading effect.

A more recent paper to have come out entitled “Lossy image compression with compressive autoencoders” utilized a similar neural network architecture - the autoencoder - in a slightly different way[4]. This paper ditched traditional lossy compression techniques entirely, opting for an approach that applied an autoencoder to the entire field of data. This had marked success, and, as the author pointed out, could be specifically trained for each image type, which could be extraordinarily beneficial to the algorithm's overall applicability[4]. This model, again, was significantly better than the one presented in this paper. This can be attributed to a common occurrence in the field of machine learning - the authors seemed to just throw computing power at the issue, by implementing a nine layer encoder with thousands of weights per layer, and 8 layer decoder with thousands of weights per layer as well[4]. The actual implementation itself, though the creation of the loss function was mathematically impressive, seemed to rely heavily on the stacking of convolutions rather than justifying the computational power needed to train each layer[4]. Overall, however, the high quality of the final products cannot be argued with.

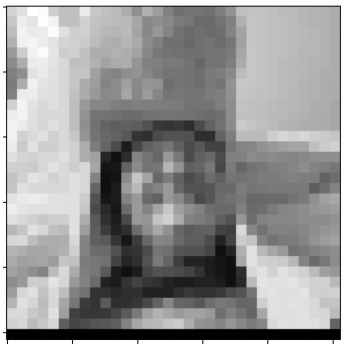
Performance of this Project

Using the same test image, a 256 by 256 pixel image of a human face in a moderately lit room, we see the following compression rates and images for each algorithm.

PNG
File size: 65,536 bytes

JPEG (Standard)
File size: 10,461 bytes

JPEG (Implemented in this project)
File size: 12,860 bytes



Autoencoded JPEG (Implemented in this project)
File size: 9,857 bytes

References

1. JPEG. (n.d.). JPEG. Retrieved November 21, 2021, from https://jpeg.org/jpeg/
2. Tompkin, J. (2019). PDF. Surrey; Simon Fraser University. Accessed at https://www.cs.sfu.ca/~yagiz/cpim/2019-CPIM-03-c-JPEG.pdf
3. J. Robinson and V. Kecman, "Combining support vector machine learning with the discrete cosine transform in image compression," in IEEE Transactions on Neural Networks, vol. 14, no. 4, pp. 950-958, July 2003, doi: 10.1109/TNN.2003.813842.
4. Theis, Lucas, et al. "Lossy image compression with compressive autoencoders." *arXiv preprint arXiv:1703.00395*(2017).