Differentiable Entropy Codes for Trained Image Compression

Isaac Liao
MIT
77 Massachusetts Avenue, Cambridge MA
iliao@mit.edu

Abstract

Information compression is a key goal in the study of machine learning and artificial intelligence, since it implies the possession of knowledge. We explore in this report a novel class of lossless image compression schemes to extract as much knowledge and structural information as possible from an image. Our main goal is to explore the feasability of compression through flexible forms of image generation, in contrast to current methods which tend to be restrictive and less adaptive. The compression scheme which we rely on allows us to concurrently train the content and quantity of encoded information at the same time, using a traditional machine learning optimizer to effectively optimize the file contents to minimize their required size. The direct training of the encodings circumvents our need to design an encoder, allowing us to use all sorts of decoder architectures which we would otherwise not be able to come up with an encoder for.

1. Introduction

A simple way to guage an upper bound for the amount of knowledge that an intelligent system has learned from a dataset is to ask the system to compress the dataset into as small a file as possible. This is because in order to compress data, the system must be able to at least partially predict the data, and to do this it must hold knowledge about the data. This means that a better compression ratio necessarily indicates better knowledge of the dataset. Thus, compression ratios can serve as a metric to measure the amount of knowledge or intelligence contained within a system. This is why the machine learning researchers often study compression, since a good compression ratio necessarily implies that we have constructed a system which is intelligent or knowledgeable.

In this report, I will present a class of lossless image compression schemes which makes a direct attempt to optimize a compression ratio. Unlike in typical image compression algorithms, I do not aim for speed of compression and decompression so long as the problem can still be solved within a reasonable amount of time, as I have no intent to use my method for the typical purposes of compression, but only to extract knowledge from images. The main innovation which I present is a special format for the representation of encodings containing information about an image. An approximation of the encoding and decoding process is fully differentiable, allowing us to include the entropy of the encodings and the encoded data both as part of gradient calculations. This opens up the possibility of compression by direct minimization of the total entropy by tuning the encodings, and lifts restrictions on the structure of the decoder.

2. Related Work

This work is inspired by the Hutter prize for compressing human knowledge[1] which is a contest which aims for the same goals which I have stated, but with a database of text rather than images.

The current most popular method of lossless image compression the portable network graphic (PNG) file format. PNG uses text compression methods such as Lempel-Ziv dictionary coding and Huffman coding. It also implements a rudimentary form of prediction of pixel colors and stores only prediction errors. Other methods such as [3], [1], and [2] include the use of a convolutional neural network to generate encodings and a deconvolutional neural network to restore the image. Neural methods currently hold state-of-the-art compression ratios. Unfortunately though, these neural methods may only be used for fixed-size images, and thus are not as flexible as PNG. Note that lossy compression algorithms may be transformed into lossless algorithms by appending tables recording the error for every pixel to the compressed files. A caveat is that these “error tables” must be entropy coded to reduce their size, and thus a distribution of the errors must be known in advance, which makes this approach difficult.

[1]More information can be found at http://prize.hutter1.net/

3. Method

I will first describe the information contained within the encodings, then how this is used by the decoder, and finally how the encoder simply performs gradient descent on an approximation of the decoder.

3.1. Codec

The default representation of the file which we work with consists of a list of encodings, appended with an entropy-coded per-pixel error table.

We use arithmetic coding for the error table, and thus the expected distribution of errors must be given in advance in order to decode the error table. This distribution takes the form of a per-pixel per-channel Cauchy distribution, with scaled means and spreads taken as separate channels of the output of a decoder CNN which we describe later.

Each encoding is some element of a predefined sample space of finite measure. For example, an encoding might be 0.78 on the interval from 0 to 1 of finite length, or $(1/2, \sqrt{3}/2)$ on the unit circle of finite perimeter. To store information in these encodings, we specify that each encoding must be restricted to a certain part of the sample space, for example that the point on the circle is in the first quadrant. Each encoding in a list may be restricted to its own portion of its own size: some may be unrestricted, while others may be narrowed very nearly to a point. The amount of entropy contained in each of these encodings is then the negative log of the fraction of the sample space that the encoding is restricted to. For instance, specifying the first quadrant of the circle adds two bits to the total entropy.

In this work, we only use the unit interval and the unit circle as sample spaces, and contiguous intervals with a low and high bound (angular bounds for the unit circle) as the restricted regions. Their dimensionality of one allows for us to approximate gradients with respect to the restricted region bounds, facilitating the training process central to our work.

3.1.1 File Storage

When we store encodings into a file, we only specify that they must each fall within their restricted intervals. A theoretically entropy-efficient way to store these encodings is to seed a random number generator with a fixed seed, and use it to guess entire lists of encodings until we find a list which puts each encoding within its respective restriction in the sample space. The number of lists we try is then an integer which can be stored in a file, and a set of valid encodings can then be restored by reading the number of tries from the file and generating that many lists of encodings with the same seeded random number generator. Furthermore, if the entropy of the list of encodings is known to the decoder beforehand, then this number of tries can be determined to come from an exponential distribution whose decay depends on this entropy, and therefore this number of tries can be compressed through arithmetic coding. In theory, this allows us to store arbitrarily long lists of encodings, each with arbitrarily large or small entropies, in a file of the same total entropy, with a negligible constant entropy overhead of about one nat[2] in expectation.

In practice, it becomes infeasible to store any more than about 10 nats at a time through this method because this would require us to generate copious amounts of random encodings. Our best option is to partition the encodings into segments of $\sim$8 nats, produce the integer number of tries for each segment, and then arithmetically encode the entire sequence of integers, given that the entropies of each segment are known to the decoder.

Two more layers of additional partitioning, segmentation, and other forms of separation such as intermediate file storage must be used to overcome the computational restrictions of large integer multiplication and RAM size, respectively. These additional techniques must also be used for the arithmetic encoding of the error table, since it suffers from the same computational restrictions.

Since it is possible in theory to store the list of encodings in a file of nearly the same entropy, we know that the encodings can only possibly contain the amount of information indicated by their entropy. Empirically though, these computational workarounds result in an addition of prohibitively large amounts of entropy to the file. This results in a highly degraded compressed file size, despite the fact that the codec could have theoretically compressed it to be much smaller. In this case, we know that the additional entropy is merely due to the codec, and is not representative of the amount of knowledge we have extracted from images, which is better measured by the theoretical entropy. Thus, for the sake of the study of intelligence, we argue that the theoretical entropy as the relevant measure of success, but for completeness we will report both the theoretical entropy and the file size.

3.1.2 Gradients of Encodings

Often, we need to differentiate across the encoding process in order to train the center and size of the restricted region of each encoding, as these are the variables which produce the compressed file and determine what image gets generated. The error table is not trained; instead it is merely there to correct for any "mistakes" that the file of encodings may tell the decoder to make, in order to enforce the losslessness of the compression. But since the codec consists of discrete steps, the best we can do is to approximate the behavior

[2] Just as a bit can be guessed correctly half the time, a nat can be guessed correctly $1/e$ of the time.

of the codec during training in a way such that a "pseudo-gradient" can be taken.

Our method of doing this is to sample random uniform encodings on the unit interval, and then linearly map them onto the restricted intervals of each encoding using their bounds. It is essential to sample first and then map to the region, and care must be taken not to sample directly from within the bounds, or no gradients will flow to the bounds and training will not happen. The default representation of a restricted interval is a pair (a, b) where a is the center and $\ln(1 + e^b)$ is the entropy, from which the interval width can be computed. This allows for extremely large or small entropies to be learned, as both extremes may be useful in our method.

3.2. Decoder

The decoder algorithm mostly consists of a CNN which performs repeated superresolution of a small uniform fixed-size seed image of many channels, until it reaches the size of the desired image. It then applies a CNN head which outputs an image of 6 channels: a Cauchy distribution's mean and width for each color for each pixel. An error table is finally generated through arithmetic coding according to these distributions, and its entropy may be calculated. Encodings may be used in a variety of ways to affect the output of the decoder. The two options which we explore are the use of encodings to store the weights of the CNN, and the adding of per-pixel encodings as new channels within the layers of the superresolution CNN and the CNN head.

For each superresolution, the following steps are performed:

- We begin with an image with many channels.
- The image is doubled in size through upscaling.
- Optionally, one row and/or column of the image is cropped away. The choice whether or not to crop is taken such that we end up with a final image of the correct dimensions.
- The image passes through several convolutional layers, each with a stride of 1, a kernel size of 3, and no padding. This reduces the image size by double the number of convolutions in both dimensions. The final number of channels is the same as that of the image we started with.

Before performing the superresolutions, we first backward-compute the sequence of image sizes which are required to produce the output image size. These image sizes decay exponentially until they reach a fixed minimum size, which we take to be the size of the seed image.

All parts of the network other than those specifically said to be part of the encodings are assumed to be part of the decoder. These parts must be trained using Adam during training time, and must function well during test time regardless of what image is being decoded. To train the decoder, we concurrently train the encodings' restricted intervals along with all parts of the decoder, to minimize the log of the total entropy of the encodings and the error table. We have found it useful to begin by weighing the entropy of the encodings very lightly early on in training, and to gradually increase to full weight later on, since otherwise the optimizer immediately learns to encode nothing before it is able to learn that the encodings may be useful to it.

3.3. Encoder

The encoder simply uses the Adam optimizer to train the encodings' restricted intervals with a fixed decoder in order to minimize the log of the total entropy of both the encodings and the error table.

4. Method Variants

We have tested two variants of encoder/decoder pairs, which both make use of the idea of differential entropy codes, but in different ways.

4.1. Per-Pixel Circular Encodings

The first method we attempted was to concatenate per-pixel encodings to the image within the convolutions of the superresolver and the head. The encodings were represented as angles on the unit circle, and the cosines and sines of the encodings were taken before concatenating them onto the images as new channels.

The superresolver CNN began with an image of 20 channels. It then upsampled and passed the image through a convolution with 40 channels, a relu activation, then a convolution with 20 channels. 5 of these channels were then used to create encoding interval sizes, and the sines and cosines of 5 encodings per pixel were concatenated to the image. It then passed the image through another 40 channel convolution, a relu activation, and a final convolution back to the original 20 channels. The head CNN did the same operations, but with a different set of weights, and ending with 6 channels. During the final stages of training, we froze the stages of the superresolution one by one and generated their true encodings rather than their random approximations, so that the later stages could fine tune to the true encodings that the earlier stages had received. The superresolver, head, and encodings for up to only a quadrant of each of eight images were trained at the same time due to RAM restrictions.

This method was designed with the idea in mind that the simple superresolver would predict some of the details introduced during upscaling, and request encodings wherever needed to fill in anything missing at that scale. This would allow for the encodings to produce image features of all sizes. During encoding, we initialized 3 channels of each layer of encodings with the target image, scaled and resized as necessary. We needed to train the first 20 steps purely with the objective of maximizing the encodings' entropy to encourage the encoder to request encoded information, as it did not do this by default. We also omitted the encodings' entropy from the loss function thereafter until the last 500 steps where this would be quickly trained, because otherwise the encoder would revert and again decide not to request the encodings, before it learned how to use them.

4.2. Unit Interval CNN Weight Encodings

While debugging the CNN in the first method, we noticed that it was powerful enough to perform compression without using any corrections at all. Moreover, the file size of the raw CNN weights was less than the size of the image, so we decided that another possible approach would be to use the encodings to encode only the CNN weights instead, and not concatenate any encodings to the image in between convolutions as before. The CNN weight encodings lied on the unit interval, and the decoder used a small neural network of layer sizes $(1, 40, 40, 1)$ and relu activation to map these to raw CNN weights. These were then inserted into the CNN, which superresolved the seed image into the desired image. To break translational symmetry, we had to add zero-padding to the convolutions, which had the side effect of reducing the seed image size to a single pixel. Every time an image was compressed, a full CNN would be trained to generate the entire image. The superresolver CNN had layer sizes $(40, 200, 100, 40)$, while the CNN head had layer sizes $(40, 100, 6)$, and all activations were relu. The CNN weights' restricted interval centers were first trained for 3500 steps to minimize only the error table entropy, and then their interval lengths were trained for 2000 steps to minimize the total entropy, for the same reasons as in the first method. A major downside to using this technique is that although the learned CNN weight encodings may have a low entropy indicating that they efficiently store information about the image, their restricted interval lengths cannot be predicted in advance, so it becomes practically infeasible to arithmetically encode the weights for file storage using the techniques previously described, without introducing copious amounts of entropy.

5. Evaluation

The images used for compression come from the Kodak image dataset[3], consisting of 24 images. We have split the dataset by assigning the first group of eight as training data, the second group of eight as validation data, and the last group of eight as test data. The per-pixel encoding method was trained on one quadrant of each of the eight training images, and the CNN weight encoding method was trained on only the first image of the training data. Both methods were used to compress the images as small as possible. The resulting file sizes, averaged over the eight test images, are shown in Table 1. The CNN weight encodings cannot be efficiently entropy-coded into a file for storage because their distributions are not known during decoding, so their file sizes are replaced with their theoretical counterparts. Despite this, their theoretical entropy is still indicative of how much the CNNs have learned about the images. We also visualize the per-pixel means and errors produced by the CNNs for both methods, as a way to depict the respective contributions of the encodings and the error tables in generating the images. These visualizations are shown in Figures 1, 2, 4, 3, and 5. An intermittent bug in our arithmetic coding also corrupted large amounts of encodings for the per-pixel encoding method, causing pixellation artifacts which ruined the small error table sizes, since larger corrections were needed to remove the artifacts. While we were still able to compress the images to a smaller size than the original raw bitmaps for both methods, we were unfortunately unable to surpass the quality of the commonly used PNG algorithm as we stated as our original goal.

Several observations can be made from these results about the information content of raw uncompressed bitmap images. Firstly, much of the image entropy takes the form of noise captured in the error tables rather than structure

Table 1: Average entropies of compressed test images.

Method	Average Entropy
No Compression	1152KB
PNG	629KB
Per-Pixel Encodings	1001KB
CNN Weight Encodings	947KB

(a) 602KB (b) 781KB (c) 671KB (d) 492KB

(e) 637KB (f) 702KB (g) 558KB (h) 706KB

Figure 1: Test dataset of images to compress. PNG file sizes are shown below each image. The file size of the raw uncompressed 8-bit bitmap image is 1152KB for all images.

[3]More information can be found at http://r0k.us/graphics/kodak/

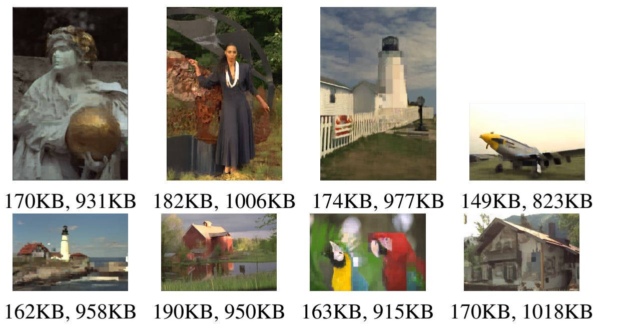

Figure 2: Predicted image means generated by the CNN weight encoding method. The theoretical entropies of the weight encodings and the total entropies are shown, respectively.

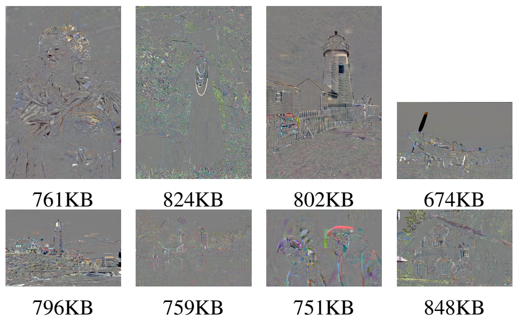

Figure 3: Error tables and their sizes, from the CNN weight encoding method.

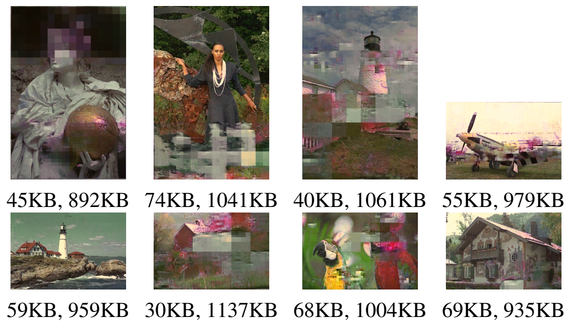

Figure 4: Predicted images means generated by the per-pixel encoding method. The actual file sizes of the encodings and the total entropies are shown, respectively. Due RAM limitations, we actually store hundreds of files for each image, and the sum of their sizes is what is shown.

captured by the encodings, and is therefore not very succeptible to compression. Secondly, the CNN weight encoding method tends to capture only large scale objects, but is still able to compress, indicating that a significant portion of the entropy is present as large-scale structure, and is compressible. The performance of the per-pixel encoding method analogously shows that a significant portion of the image

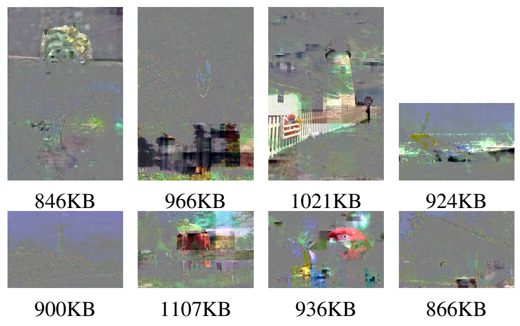

Figure 5: Error tables and their sizes, from the CNN per-pixel encoding method. Artifacts from a bug in our arithmetic encoding force us to use a larger error table.

entropy is present as compressible small-scale structure too.

6. Conclusion

We have demonstrated in this paper how one can reduce the problem of image compression into an optimization problem for which we may apply traditional machine learning methods. The codec which we have introduced has a differentiable approximation, allowing the content and quantity of information to be trained concurrently. This gives us tremendous flexibility in choice of decoder architecture, since we can always use an encoder like our to train the encodings to minimize their entropy, and subsequently convert them into a file. In particular, we have studied generative decoders, which repeatedly superresolve a seed image to produce the target image from very little information and then apply corrections. Our codec could potentially be used to provide increased flexibility in learned generative compression of other types of data too, not only images. We hope that our methods may therefore become useful in future work on compression.

References

[1] Fabian Mentzer, Eirikur Agustsson, Michael Tschannen, Radu Timofte, and Luc Van Gool. Practical full resolution learned lossless image compression, 2020. 1

[2] Scott Reed, Aäron van den Oord, Nal Kalchbrenner, Sergio Gómez Colmenarejo, Ziyu Wang, Dan Belov, and Nando de Freitas. Parallel multiscale autoregressive density estimation, 2017. 1

[3] George Toderici, Damien Vincent, Nick Johnston, Sung Jin Hwang, David Minnen, Joel Shor, and Michele Covell. Full resolution image compression with recurrent neural networks, 2017. 1