Dennis Du, Jared Ni, Sezim Yertanatov

CS2241 Final Project Status Report

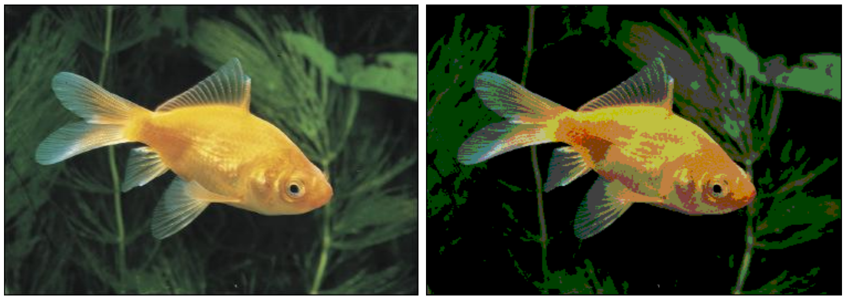
Our project aims to develop and evaluate a novel image compression technique tailored for resource-constrained edge devices performing object detection tasks over limited networks. The core motivation is that for many object detection scenarios, an edge device (such as AR glasses or a robot performing object recognition in a controlled environment) may not have the computational resources to run an image classification model, so the device sends the image over a network to a server to perform the classification. The server then responds with the model's prediction. In order to reduce communication on the network, we want to be able to send as compact a representation of the image as possible while still allowing the classification model to produce predictions of comparable accuracy. That is, the compressed image file need not look "nice" to the human eye—we simply want the classification model to be able to produce the correct result. The model should be trained on these compressed images for the best performance.

The procedure is described in detail below, but to start, we give a general overview of the direction. In many of the steps, we create a value that is taken to be the default value over some property or section of the image; this value is either stored explicitly in the compressed file or is implied by the algorithm. The bulk of the storage space is devoted to storing deviations from this default—the "surprising" values. Thus, we want to transform the image data into a form that reduces the number of surprising values, both in terms of reducing the size of the universe of possible values (allowing us to use fewer bits for each one) and in terms of reducing the number of times a surprising value actually occurs in our data. Ultimately, the idea is to store surprises in Bloom filters, which, for a given false positive probability, requires a constant number of bits per element, so we want to minimize the number of elements stored.

Depending on the results of experiments, we may narrow down the number of steps we take, eliminating those that do not help with the compression or make it worse, as measured through a tradeoff between the compression rate and the classification accuracy.

1. Quantization:

An obvious first step is to quantize the color values in the image. This can greatly reduce the space of possible values, allowing us to use fewer bits to represent each value. Reducing the space of possible values also means it is easier to identify good default values in the subsequent steps that are able to cover a large portion of the image, reducing the number of surprises. Images can look recognizable even after each RGB color channel is quantized to only four unique values (which means a total of 4^3=64 unique colors). The number of values to use in the quantization can be adjusted based on the results of experiments. As an example, quantizing the image on the left using four unique values on each channel produces the image on the right. This is a uniform quantization, but it may also be valuable to consider non-uniform quantizations.

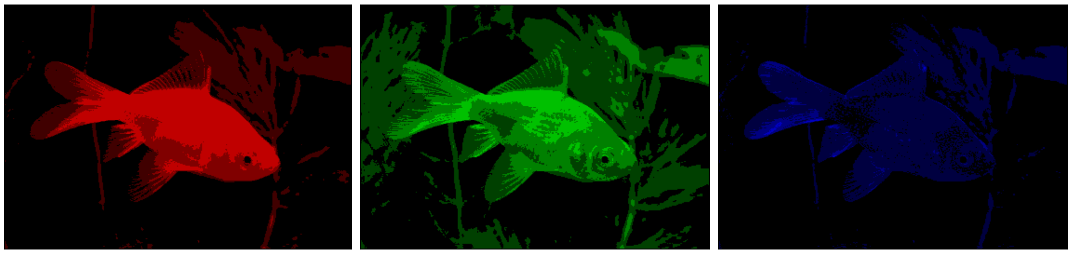


2. **Segmentation:**

Different portions of the image show different things, but each localized section tends to look fairly similar. Hence, even though it may be difficult to find good default values for the whole image, each localized region may be well-represented by a default value, with few surprises. Thus, we can divide the image into segments, and treat each segment separately in the steps that follow. A segment might be a 16×16 block of the image, for example.

3. **Linear Regression**:

Naturally, there is some amount of correlation, and thus redundancy, in the red, green, and blue channels of the image, as we see below.



We can take advantage of this by expressing two of the channels in terms of the third one. Specifically, we fit two linear models, one using the red channel to predict the green channel and one using the red channel to predict the blue channel. We then keep track of the residuals of the linear models. Thus, the green and blue channels are represented by the coefficients of their corresponding linear models as well as an array of residuals. The red channel remains untouched by this step. In all subsequent steps, we apply the transformations to each channel separately; in other words, we treat each channel as a separate grayscale image. However, whereas for the red channel we simply use the array of red channel values, for the green and blue channels we use the array of *residuals* of green and blue channel values. The default value in this case is represented by the linear model's prediction (computed using the coefficients and the red channel), while the residuals play the role of the surprise.

4. **Produce Sparse Matrices:**

In this step, we choose either Delta Encoding or Background/Mode Removal to apply to each channel of the image, with the goal of producing a sparse matrix from each section of the image that we have segmented earlier, maximizing the number of zeroed-out pixels in each color channel of each image segment.

With a quantized image, it's expected that many neighboring pixels will have the same value; thus, we can capitalize on this by applying delta encoding to produce a sparse matrix that is easily compressed when compared to a normal image. For example, on a sample array of the values [1, 1, 1, 2, 2, 3], the delta encoding will be [1, 0, 0, 1, 0, 1].

In a later step, we use Bloom filters, which introduce false positives. Small error rates of the Bloom filter matter much more when using delta encoding due to error carried forward, since delta encoding encodes the difference between adjacent pixel values, so errors would increase the further we move from the initial value. Thus, we will implement periodic anchor values that ground each section of the encoding as an error correction mechanism to reset the potential error carried forward by the Bloom filter's false positives. (If we choose small enough segments in the segmentation step, this may already be taken care of.)

We also observe a positive interaction between delta encoding and the linear regression step: a typical image quantized to four values and then delta-encoded has 10% non-zero values in the encoding of the green channel, while applying the linear regression and thus delta-encoding the residuals instead brings this down to 3%. Though these values vary highly by image, using linear regression consistently produces a smaller number of nonzero values.

Mode removal is a similar but orthogonal idea, where we first find the most common color (the mode) in each RGB channel in each image segment, and then we normalize that segment channel with the mode color as the base value. The expectation is that the mode pixel value will be zeroed out, and every other pixel will be adjusted around this mode value as the base. For a sample array of values [1, 1, 1, 2, 2, 3], the mode normalization will be [0, 0, 0, 1, 1, 2]. Bloom filters' false positives will have less of a negative impact in the mode removal case because of no error carried forward.

5. Smearing

We came up with SMEAR to reduce the possible range of values when we find the differences between values, and it helps us to decrease the number of Bloom filters we need in the next step. For that, we define a MAX_JUMP value that covers both the positive and negative values. It is the largest difference that we allow between any two consecutive numbers. So, when we find the difference between any two values x and y, we only put **min((y - x), MAX_JUMP)** or **max((y - x), -MAX_JUMP)** depending on the sign of the difference, and if **|y - x|** is larger than the maximum jump size, we save the remaining **(y - x) - MAX_JUMP** or (**y - x) + MAX_JUMP,** depending on the sign, for the encoding of the next number. For example, suppose we have an array [0, 1, 3, 8, 4, 2, 1] and MAX_JUMP = 3. We are trying to do the Delta Encoding for it, which basically means we find the differences between consecutive values. Then, the resulting array without using the SMEAR idea is [0, 1, 2, 5, -4, -2 , -1]. If we use the SMEAR algorithm, then it would be [0, 1, 2, 3, -2, -2, -1]. In general, smearing ensures we have a total of 2*MAX_JUMP unique nonzero values.

6. **Bloom Filters:**

At this point, the idea is that the image is now encoded in such a way that there are a small number of nonzero values, both in terms of the total count and in terms of the number of unique such values. Thus, it makes sense to store a set containing the indices of all the 1s, a set containing the indices of all the -1s, and similarly for 2, -2, 3, and -3 (and so on if there are more unique nonzero values). We can compactly store such sets as Bloom filters.

If we have n possible quantized values, we apply (n-1) Bloom filters, one for each of the (n-1) nonzero values. In order to retrieve the values stored in the filters during decoding, we query each Bloom filter, asking whether it contains the current pixel, and assigning to the pixel the value associated with the Bloom filter containing it. If the pixel appears to be a member of more than one Bloom filter (which can happen when we get false positives), we randomly select one of the values. If we do not get a “yes” from any of the (n-1) filters, then we know that the value must have been 0.

Theoretical Analysis:
For our method to make sense, we need it to be better than established ways of compressing information. After we have produced a sparse matrix representation of the image, we have several ideas of how to compress the data further.

- **Apply lossless byte encoding to sparse matrix**.
    - The first method is to use a lossless byte-encoding algorithm such as LZ77. Depending on the data in the sparse matrix, it is reasonable to assume a 2-5x additional compression ratio on the sparse matrix. For the sake of simplicity, we assume that each element of the sparse matrix is a 2-bit number and 0.5-bit after applying compression. On a 512x512 image, this would result in 512*512*0.5 = 131k bits.
- **Applying Bloom filter.**
    - Assuming 4-bucket quantization, we need 3 Bloom filters per segment, so 2-bit x 3 = 6-bit / segment for metadata.
    - Assuming the optimal bits equation, for a tolerance of 10% error rate, we need -2.08 ln(0.1) = 4.8 bits / non-zero elements.

$$\frac{m}{n} = -\frac{\ln(\varepsilon)}{\ln(2)^2} \approx -2.08\ln(\varepsilon)$$

    - For a 512x512 image, if 10% of the image is non-zero, it results in 512 x 512 x 0.1 x 5 bits = 131k bits.
- **Hybrid approach.**

- A hybrid approach would have the theoretical guarantee of lossless byte compression in the worst case, but the optimization of the Bloom filter when the non-zero element rate is small in a segment given that the non-zero element percentage is small, with a tradeoff for pixel accuracy in the false positive case (it is complicated to calculate the true error rate of a pixel's value if it's present in multiple Bloom filters, so we will have developed this part later in the project).
- For segments where the non-zero element percentage is smaller than x% (as we have shown, Bloom filter performs better with x<10%). We apply Bloom filters to find non-zero members. Otherwise, we call the byte encoding approach on the current image segment.
- For an example of this, consider a 512x512 image with 256 counts of 32x32 segments.
    - If the non-zero rate in an image segment is 5%, the number of bits in a segment would be 32 x 32 x 0.05 x 5 bits = 256 bits, compared to the byte encoding average of 512 x 512 x 0.1 x 5 bits = 512 bits, a 2x space reduction. If the non-zero rate in the next segment is 15%, we rely on the byte encoding average of 512 bits instead of a worse Bloom-filter complexity.
- Thus, we can achieve theoretical guarantees of traditional lossless encoding in the worst case and the Bloom filter's enhancement in the best case.

Next steps:
We plan to develop the theory fully and consider the tradeoffs we listed by expanding on our theoretical framework. We will be expanding the mathematics, and especially listing the tradeoffs between the different approaches in each step of our theoretical pipeline thus far.

Empirically, we plan to run our compression algorithm on a subset of ImageNet and use those compressed images to train a classification model in order to evaluate what compression rates we can achieve and how accurately we can handle our compressed images. This helps us gather real-world data on the true compression tradeoffs we have been comparing in theory.