Athens Exchange 2018
Telecom ParisTech
From Complexity to Intelligence

Never GIF up

Dhruv Batheja

2018/11/27

1 Introduction

The rate at which we're producing data in this age is growing exponentially year by year. Moore's law is failing. We will soon not be able to keep up with the amount of data we're producing. Our hardware is struggling to keep up with storing the huge amounts of data and let alone process it. We need better compression techniques.
A bunch of data that we generate consists of videos and GiFs and burst-shots which are all sequences of images. For this project, I tried to create an encoding representation of a sequence of images that takes up lesser space than storing the sequence of images separately. Therefore, this technique aims to reduce the number of bits required to store the sequence of images and thereby reducing the Kolmogorov Complexity.

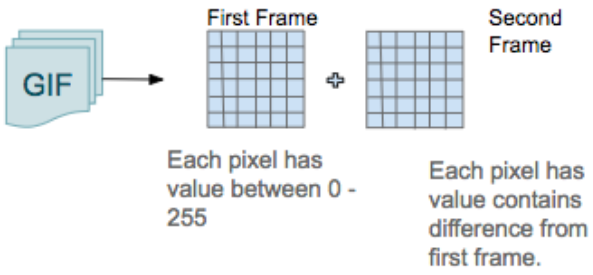

Figure 1: Storing only difference values

2 Project

Each image is a grid of pixels and each pixel has 3-values for the color bands R-G-B. These values are in the range (0-255). For this project, I transformed the image to black and white so each pixel has only one value instead of 3 (which is in the range (0-255)). The basic idea is that instead of storing these image pixel values(0-255), we'll only store the difference values w.r.t. the previous frame. For a sequence of images like in GIFs, these difference values are expected to be small(much smaller than 255), since there is not a lot of variation from frame to frame.
The basic idea of the project is shown in the figure 1. I used the imaging libraries Pillow [5] and imageio [3] for loading and producing images.

2.1 Implementation

This section contains the steps I followed while implementing this approach. For this project, I converted all frames to black and white and re-sized each frame to a generous (128 x 128).

2.1.1 GIF generation

I implemented fetching frames from the webcam to generate a new GIF. The implementation can be found in the file **webcam.py**. This file also has function utilities (*create_gif*, *split_gif*) for creating-GIFs from a list of images and getting a list of images from a GIF. This file also has a function (*prepareImage*) to re-size the images and convert them to black and white. I used opencv [4] (*import cv2*) for using the webcam [2] and image resizing etc.

Figure 2: *dark.jpg* and *light.jpg*

Usage:
$ python webcam.py sample.gif
(Press space-bar to add the frame and Escape when you're done)

2.1.2 Encoding Images

Please refer to the file **lossless.py**. I used the DoublingCode as I needed a prefix-code as provided in *FCI/PrefixInt.py*. Once I had a sequence of 0s and 1s from an image, I wrote it to a binary file by splitting the 0s and 1s into chunks of 8-bits(bytes). Please look at the functions *writeSequenceToBinary* and *readSequenceFromBinary* for this.
For generating the sequence of numbers, I wrote utilities for encoding a sequence of numbers (for pixel values in the image). For DoubleCoding an image, the first two values in the final-sequence are the DoubleCoded shape values of the image i.e. the *rows and columns* in the image followed by the DoubleCoded pixel values. Look at the functions *encodeImage and decodeImage* for these.

2.1.3 Encoding GIFs

Each GIF has a sequence of images with sequenceLength = number of images in the sequence. Now for encoding a GIF, I stored the DoubleCoded values of the sequenceLength followed by DoubleCoded values of rows and columns in the images which is followed by the the DoubleCoded pixel values of all the frames. Look at the functions *doublingToGif and gifToDoubling*.

Comparing the size of these encoded files to storing the numpy image collection as expected, is quite small.

2.1.4 Lossless Compression

This is where I implement the idea discussed in the figure 1. For this encoding, I stored the following doubleCoded values in the encoded representation:

1. SequenceLength (Number of images in the sequence)
2. Rows (Number of rows in each image)
3. Columns (Number of columns in each image)
4. All Pixels of the first frame (number of pixels in the first frame = Rows * Columns)
5. For all but the first frame, for each pixel, the difference-value (subtraction) from the previous frame is stored (range = -255 to 255). So an extra bit(0 or 1) is prepended to each pixel's doubling code which indicates if the difference value is +ve(0) or -ve(1).

The implementation for this can be found in the functions *gifToDoublingWithDifference and doublingToGifWithDifference.*

Usage:
$ python lossless.py sample.gif
(It encodes the gif to a DoublingCode and then encodes it with the proposed code and then compares the proposed representation with the DoublingCode and shows a Compression% result.)

Figure 3: Frames from *webcam.gif*

2.2 Experiments

For the images, a darker image (with pixel-values closer to 0) must be smaller in size compared to a lighter image (with pixel values closer to 255) since the doubling code for 0 is 100 (length=3) and the doubling code for 255 is 1100000111111111(length=16).
I used a light and a dark image (***light.jpg*** *and* ***dark.jpg***) for this experiment as shown in the figure 2. As expected, I got a smaller encoding for the darker image.

```
>>> encodeImage("light.jpg")
-------------------- [ ENCODING light.jpg ] --------------------
ENCODED SIZE:32603
-------------------- [ DONE ] --------------------
>>> encodeImage("dark.jpg")
-------------------- [ ENCODING dark.jpg ] --------------------
ENCODED SIZE:23423
-------------------- [ DONE ] --------------------
```

For my experiments, I used GIFs with 2 kinds of variations in simultaneous frames. The first GIF that we'll look at is the **webcam.gif**. The figure 3 shows that the frames have slight differences, so our approach of storing difference values should work well here. And as expected, compared to the normal doubling coded representation, this technique gives us a compression of about 30%.

```
(venv3)   athens python lossless.py webcam.gif
==================== [ Creating DoublingCode for < webcam.gif > ] ====================
\/ All sizes in bytes \/
```

Figure 4: Frames from *random.gif*

```
ENCODED: 105848 < gif_doubling >
ORIGINAL(numpy image-list pickle): 524576 < gif_pickle >
DECODED: 524576 < gif_decoded_pickle >
==================== [ Lossless-Compression using difference values < webcam.gif > ] ===============
\/ All sizes in bytes \/
ENCODED: 74366 < gif_doubling_difference >
ORIGINAL(numpy image-list pickle): 524576 < gif_pickle >
DECODED: 524576 < gif_decoded_pickle_difference >
[ COMPRESSION with respect to DoubleCoded GIF = (105848 - 74366)/105848 = 29.74 percent ]
```

Figure 5: Frames from *similar.gif*

The next GIF is **random.gif**. As the figure 4 shows, the sequence of frames have nothing in common, so the approach of storing the differences instead of actual pixel values actually leads to a negative compression.

```
(venv3)   athens python lossless.py random.gif
==================== [ Creating DoublingCode for < random.gif > ] ====================
\/ All sizes in bytes \/
ENCODED: 124452 < gif_doubling >
ORIGINAL(numpy image-list pickle): 655689 < gif_pickle >
DECODED: 655689 < gif_decoded_pickle >
==================== [ Lossless-Compression using difference values < random.gif > ] ===============
\/ All sizes in bytes \/
ENCODED: 132907 < gif_doubling_difference >
ORIGINAL(numpy image-list pickle): 655689 < gif_pickle >
DECODED: 655689 < gif_decoded_pickle_difference >
[ COMPRESSION with respect to DoubleCoded GIF = (124452 - 132907)/124452 = -6.79 percent ]
```

As a test, I also tested it on **similar.gif** which has the same frame over and over(Figure 5). And our approach shines by giving a compression of 34%.

3 Reflection

As a future work of this project, I could implement a lossy version of compression which would involve clustering [6] the pixel difference values and storing the cluster-means instead of storing the actual difference values. This would compress the image quite a lot(based on the number of clusters we chose (The higher the better)). But, the compression would come at the cost of quality. This could be feasible maybe for archiving the image-sequences where a degraded version could suffice or maybe be used when the network connection is poor.
These lectures on Kolmogorov Complexity completely changed the way I perceive randomness. As Chaitin said, "Comprehension is compression", I feel I have comprehended quite a lot about complexity,

turing machines, encodings, compression etc. and hope to use them in my research. I feel, I understand the importance of context now.
My notes for the lectures can be found here [1]. The zip contains all the files mentioned in this report.
This turing machine(me) is now coming to a halt!

References

[1] Batheja Dhruv. Notes. *https://github.com/live-wire/journal/tree/master/complexity$_t$o$_i$ntelligence*, 2018.

[2] Taylor D. Edmiston. Video capture. *https://gist.github.com/tedmiston/6060034*.

[3] Imageio. examples. *https://imageio.readthedocs.io/en/stable/examples.html*.

[4] Opencv. images. *https://opencv-python-tutroals.readthedocs.io/en/latest/py$_t$utorials/py$_g$ui/py$_i$mage$_d$isplay/py$_i$mage*

[5] PIL. Docs. *https://pillow.readthedocs.io/en/5.3.x/*.

[6] Sklearn. K-means clustering. *https://scikit-learn.org/stable/modules/generated/sklearn.cluster.KMeans.html*.