Final report

Google Landmark Retrieval Challenge

Abstract

In this report, we overview different approaches for finding the same landmarks from a large dataset that corresponds to a given query image. We apply several traditional and non-traditional models to our Google Landmark Retrieval dataset and make a comparison between them based on speed, performance rate, and simplicity. As we had an established amount of time for project implementation, our main goal was to make our approach perform as fast as possible due to computing capability constraints.

Introduction

Image retrieval is a fundamental problem in computer vision: given a query image, can you find similar images in a large database? This is especially important for query images containing landmarks, which accounts for a large portion of what people like to photograph.

Importance of the problem and (potential) impact of your work

Datasets

During the process of project development, we used 3 datasets: 2 for testing purposes and 1 main for final submission and checking how our models and approaches would work with really big data.
Our datasets:

- Paris6K
    - The Paris Dataset consists of 6412 images collected from Flickr by searching for particular Paris landmarks. There are 12 unique landmarks over which an object retrieval system can be evaluated.
- Oxford5K
    - The Oxford Buildings Dataset consists of 5062 images collected from Flickr by searching for particular Oxford landmarks. There are 11 unique landmarks over which an object retrieval system can be evaluated.
- Google Landmark Retrieval Dataset
    - train set - 1 116 904 images. (359 GB)
    - test set - 117 703 images. (40 GB)

The dataset we use is the largest worldwide dataset for image retrieval research, comprising more than a million images of 15K unique landmarks

Modeling

Firstly, we start with RootSIFT.

RootSIFT

Instead of using a simple SIFT, we use RootSIFT, that can be used to dramatically increase object recognition accuracy, quantization, and retrieval accuracy. Best about the RootSIFT extension, that it sits on top of the original SIFT implementation and does not require changes to the original SIFT source code. It just changes an output of original SIFT.

Here is the simple algorithm to extend SIFT to RootSIFT:

1. Compute SIFT descriptors using your favorite SIFT library.
2. L1-normalize each SIFT vector.
3. Take the square root of each element in the SIFT vector.

So, we got descriptors of every image in the Paris-5k dataset with RootSift and found a number of good enough matches (that we calculate by applying ratio test) between every image using Brute-Force Matcher. Then we have built similarity matrix, but it was quite bad. But we did not give up!

Our next attempt was HardNet!

HardNet

This is the model for learning local feature descriptors. Applying the novel loss to the L2Net CNN architecture results in a compact descriptor named HardNet. It has the same dimensionality as SIFT (128) and shows state-of-art performance in wide baseline stereo, patch verification, and instance retrieval benchmarks. It is fast enough, computing a descriptor takes about 1 millisecond on a low-end GPU.

Our final approach -

Partition Min-hashing

The algorithm is quite simple in understanding and implementation:

1. Image preparation. We convert an image into greyscale to have only one channel instead of three, by doing this we improve the speed of the algorithm. After that, we resize the image to 256x256 to easily divide them into squares, also better speed as we have fewer pixels. Last part of image preparing is cropping it into squares - partitions.
2. Hashing. The main part of the algorithm. We take M different hash functions. Take first hash function and calculate a hash for N partitions and calculate a true hash, repeat for all hash functions, where the true hash is the minimum hash value from all partition of specific hash function. The resulting array would be the set of length M containing true hashes.

Evaluation

We have tried plenty of metric systems:

- Hamming distance

$$\Delta(x, y) := \sum_{x_i \neq y_i} 1, i = 1, \ldots, n.$$

- Our own similarity metric SumOne, which we considered the right choice for our algorithm.

Here is how it works. Calculate the hamming distance of every pair with the next logic : IF 1 - diff (h11, h21) > t = 1 else 0, where t is a threshold established for particular hash function.
After comparison of every pair, we get a binary array of the length M. Distance would be the sum of ones in the binary array.

- Normalized hamming distance

$$\Delta_n(x, y) := \frac{1}{n} \sum_{x_i \neq y_i} 1, i = 1, \ldots, n.$$

If we want we can use normalized hamming distance by dividing hamming distance on hash length.

- Equality Percentage (EP)

$$EP := 100 \cdot \Delta_n.$$

To calculate equality percentage we just multiply normalized hamming distance by 100.

- mean Average Precision @ 100 （ mAP@100 ）

$$mAP@100 = \frac{1}{Q} \sum_{q=1}^{Q} \frac{1}{min(m_q, 100)} \sum_{k=1}^{min(n_q, 100)} P_q(k) rel_q(k)$$

References

1. David C. Lee, Qifa Ke and Michael Isard : on the base of "Partition Min-Hash for Partial Duplicate Image Discovery"
2. João Augusto da Silva Júnior, Rodiney Elias Marçal and Marcos Aurélio Batista : Image Retrieval: Importance and Applications
3. Guoqing Wang * and Jun Wang : SIFT Based Vein Recognition Models: Analysis and Improvement
4. R. Venkatesan1 , S.-M. Koon2 , M. H. Jakubowski1 , and P. Moulin2 : ROBUST IMAGE HASHING
5. Christoph Zauner : Implementation and Benchmarking of Perceptual Image Hash Functions
6. Anastasiya Mishchuk1 , Dmytro Mishkin2 , Filip Radenovic´ 2 , Jiri Matas ˇ 2 : Working hard to know your neighbor's margins: Local descriptor learning loss
7. Adrian Rosebrock : Image hashing with OpenCV and Python