INTRODUCTION

A lot of image datasets are created by compiling tons of images either from already available datasets & online sources or by actually capturing the images ourselves. That’s a quite good approach until we run into “*SIMILAR IMAGES*”. Unlike what others assume this not only includes all the images that are different and resemble some other image/s but also duplicates. Well this poses some problem as the results can deliver some other image that resembles the actual image that we want. Hence, it is required by some applications to detect and list out the similar images as per their requirements. This article focuses on a faster and efficient tackling of fine grained image similarity search.

PROBLEMS WE FACE & NECESSITY

- Similarity search finds application in specialized database systems handling complex data such as images or videos, which are typically represented by high-dimensional features and require specific indexing structures. While GPUs excel at data-parallel tasks, prior approaches are bottlenecked by algorithms that expose less parallelism, such as k-min selection, or make poor use of the memory hierarchy.

- Images and videos constitute a new massive source of data for indexing and search. Extensive metadata for this content is often not available. Search and interpretation of this and other human-generated content, like text, is difficult and important. A variety of machine learning and deep learning algorithms are being used to interpret and classify these complex, real-world entities. Popular examples include the text representation known as word2vec, representations of images by convolutional neural networks, etc.
- One of the **most expensive operations** to be **performed** on **large collections** is to **'compute a k-NN graph'** which is a directed graph where each vector of the database is a node and each edge connects a node to its k nearest neighbors.
- The **requirement** is to **tackle the problem of better utilizing GPUs** for this task.

SOLUTION

- In this article we are gonna have a look at one of the most robust libraries created by the social media giant Facebook and that is "***Facebook AI Similarity Search(FAISS)***", a toolbox made for serving the purpose we desire i.e ***fine grained image similarity detection***.

- The design is proposed for **k-selection that operates at up to 55% of theoretical peak performance**, enabling a nearest neighbor implementation that is **8.5× faster than prior GPU state of the art**.
- Through this implementation researchers have been able to construct a high accuracy k-NN graph on 95 million images from the Yfcc100M dataset in 35 minutes, and of a graph connecting 1 billion vectors in less than 12 hours on 4 Maxwell Titan X GPUs.

THE IMPLEMENTATION

I) ***DATA PREPARATION*** :

- The dataset used in this implementation is the **'Caltech_256 Image Dataset'** that consists of images belonging to around 256 categories/classes listed in separate class wise subdirectories.

- Alternatively if one decides to use a dataset of their own then each image should be listed under their respective class subfolder and not mixed up.
- <u>**Very Important**</u> - **Each class is selected** in a way that they **contain a minimum number of *different* images that come under the category of *similar*** i.e they **resemble some other query image from the same class**. In other words **all images** are **not distinct**.
  NOTE: It is required that each directory should contain at least one duplicate sample of a query image from the same class. This is required to make sure that the model works while testing.
- The time taken by the algo to train and get ready for testing depends on the size of the dataset. For example on a dataset of size 1300, it will take approximately 10 min(with good internet speed of course!).

II) *DATA AUGMENTATION* :

Data augmentation can prove to be beneficial since it will help in making the model more robust in recognising the trends and patterns in the images. However this implementation was not performed on augmented data and instead on the original one.

NOTE: In the **‘How to use’** section, one of the features included in the **index factory** does the job of **automatically augmenting the data before processing**.

III) *MODELLING*

- The model used in this implementation makes use of a tool that builds up the **“index”** when they are *nested* and that is the **“index_factory”**.
- The index is kind of a data structure that stores and operates on the vectorized versions of database images including training. It is further used for detecting and displaying the similar image/s of a query image. Refer to the “how to use” section for further info.
- The **index_factory function interprets a string to produce a composite Faiss index**. The string is a comma-separated list of components. It is intended to **facilitate the construction of index structures, especially if they are nested**. The **index_factory argument** typically includes a **preprocessing component**, an **inverted file** and an **encoding component**.
- **index = index_factory(128,**

"OPQ16_64,IMI2x8,PQ8+16"): takes 128D vectors, applies an OPQ transform to 16 blocks in 64D, uses an inverted multi-index of 2x8 bits (= 65536 inverted lists), and refines with a PQ of size 8, then 16 bytes. The above faiss index is used in this tutorial that is trained on the database image vectors after which all the vectors are added to it. Refer to the “how to use” section to gain more intuition on the type of features.

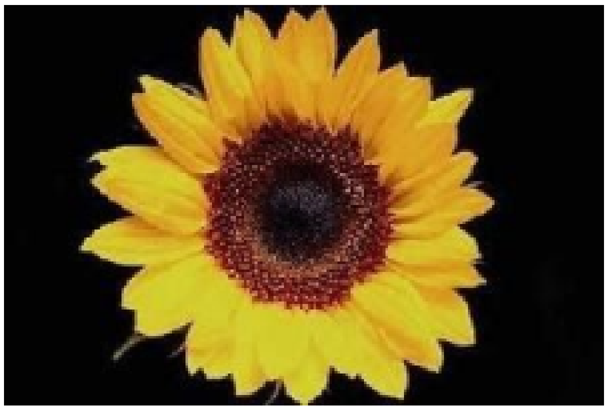
- The preprocessing component includes only one element - IDMap. For an inverted file we have options such as PCAWR64, OPQ16_64, RR64, etc as vector transforms. It also includes IVF4096, IMI2x9, IVF65536_HNSW32 that are used as non-exhaustive search components and finally the encodings are carried out by components like Flat, PQ16_64, SQ4, etc. For further info refer - https://github.com/facebookresearch/faiss/wiki/The-index-factory

COLAB NOTEBOOK

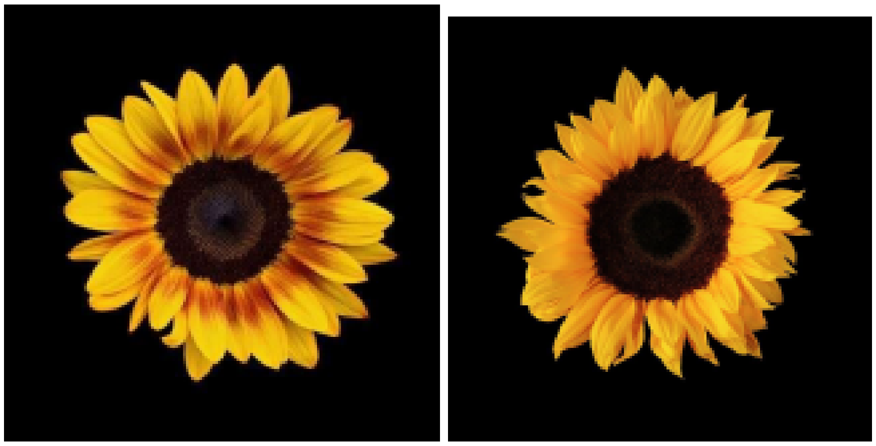
https://colab.research.google.com/drive/1OLdDYk-TzVNhpyL3pAONtkHnKGI-GUsU?usp=sharing

RESULTS WITH ACCURACY

Query Image -



Results -



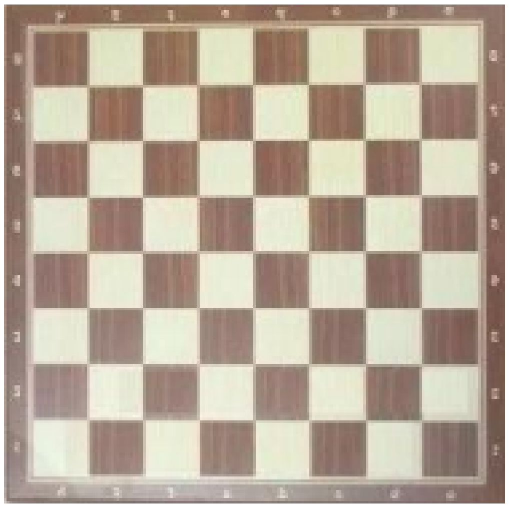
For testing purposes a query image of a sunflower with black background was posted and the results delivered were pretty good as shown above. A ‘k’ parameter was specified which is the no of nearest neighbors to be detected and in this case was k = 4. As another example, a ‘blimp’ image was posted as a query image and the results were as follows -

Query Image -

Results -

In this case though we specified k = 4, the results showed only 3 images which were a best match to the query image as shown above.

Query Image -



Results -