3.3.2: Feature extraction and indexing

Topic Progress: (https://gurus.pyimagesearch.com/topic/defining-your-image-descriptor/) (https://gurus.pyimagesearch.com/topic/indexing-your-dataset/) (https://gurus.pyimagesearch.com/topic/defining-your-similarity-metric/) (https://gurus.pyimagesearch.com/topic/searching/)

← Back to Lesson (https://gurus.pyimagesearch.com/lessons/the-4-steps-of-building-any-image-search-engine/)

Feedback

Our last lesson reviewed the first step of building an image search engine: **Defining Your Image Descriptor (https://gurus.pyimagesearch.com/topic/defining-your-image-descriptor/)**.

We then examined the three aspects of an image that can be easily described:

- **Color:** Image descriptors that characterize the color of an image seek to model the distribution of the pixel intensities in each channel of the image. These methods include **basic color statistics (https://gurus.pyimagesearch.com/lessons/color-channel-statistics/)** such as mean, standard deviation, and skewness, along with **color histograms (https://gurus.pyimagesearch.com/lessons/color-histograms/)**, both "flat" and multi-dimensional.
- **Texture:** Texture descriptors seek to model the feel, appearance, and overall tactile quality of an object in an image. Some, but not all, texture descriptors convert the image to grayscale, then compute a Gray-Level Co-occurrence Matrix (GLCM) and compute statistics over this matrix, including contrast, correlation, and entropy, to name a few (**Haralick texture**) (https://gurus.pyimagesearch.com/lessons/haralick-texture/). More advanced texture descriptors, such as **Local Binary Patterns (https://gurus.pyimagesearch.com/lessons/local-binary-**

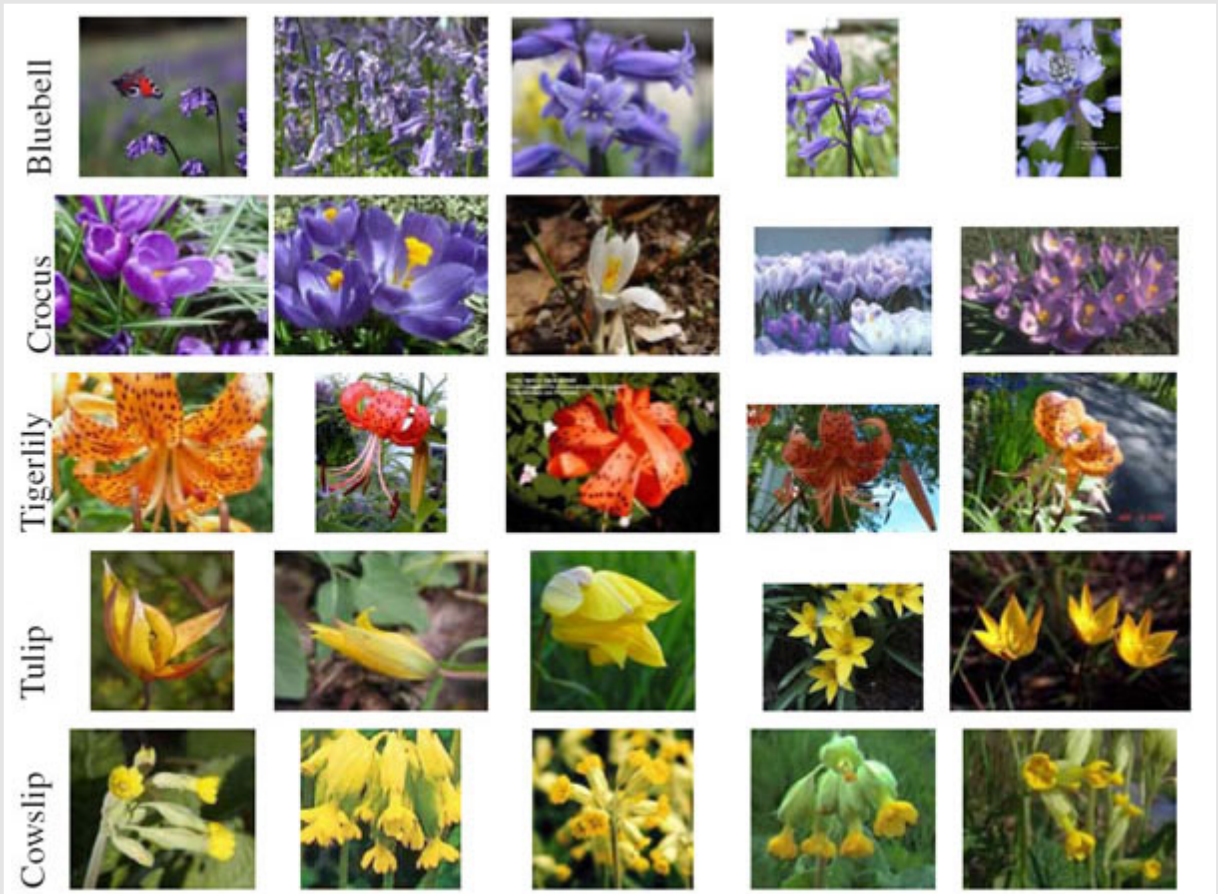
patterns/), attempt to model the *pattern* as well. Even further advanced texture descriptors such as Fourier and Wavelet transforms also exist, but still utilize the grayscale image.

- **Shape:** The vast majority of shape descriptor methods rely on extracting the contour of an object in an image (i.e. the outline). Once we have the outline, we can then compute simple statistics to characterize the outline, which is exactly what **Hu Moments (https://gurus.pyimagesearch.com/lessons/hu-moments/)** and **Zernike Moments (https://gurus.pyimagesearch.com/lessons/zernike-moments/)** do. These statistics can be used to represent the shape (outline) of an object in an image. In the context of machine learning and object recognition, **Histogram of Oriented Gradients (https://gurus.pyimagesearch.com/lessons/histogram-of-oriented-gradients/)** is also a good choice.

When selecting a descriptor to extract features from our dataset, we have to ask ourselves: what aspects of the image are we interested in describing? Is the color of an image important? What about the shape? Is the tactile quality (texture) important to returning relevant results?

Let's take a look at a sample of the Flowers 17 dataset (http://www.robots.ox.ac.uk/~vgg/data/flowers/17/index.html), a dataset of 17 flower species, for example purposes:

Feedback



(https://gurus.pyimagesearch.com/wp-content/uploads/2015/06/cbir_indexing_flowers17_sample.jpg)

FIGURE 1: A SAMPLE OF THE FLOWERS 17 DATASET. AS WE CAN SEE, SOME FLOWERS MIGHT BE INDISTINGUISHABLE USING COLOR OR SHAPE ALONE (I.E. TULIP AND COWSLIP HAVE SIMILAR COLOR DISTRIBUTIONS). BETTER RESULTS CAN BE OBTAINED BY EXTRACTING BOTH COLOR AND SHAPE FEATURES.

Feedback

If we wanted to describe these images with the intention of building an image search engine, the first descriptor I would use is **color**. By characterizing the color of the flower petals, our search engine will be able to return flowers of similar color tones.

However, just because our image search engine will return flowers of similar color does not mean all the results will be relevant. Many flowers can have the same color but are of an entirely different species.

In order to ensure more similar species of flowers are returned from our search engine, I would then explore describing the **shape** of the flower petals.

Now we have two descriptors — **color** to characterize the different color tones of the petals, and **shape** to describe the outline of the petals themselves.

Using these two descriptors in conjunction with one another, we would be able to build a simple image search engine for our flowers dataset.

Of course, we need to know how to index our dataset — right now we simply know what descriptors we will use to describe our images. But how are we going to apply these descriptors to our entire dataset?

In order to answer that question, today we are going to explore the second step of building an image search engine: **Feature extraction and indexing**.

Objectives:

In this lesson, we will review the high-level process of extracting image features from our dataset. We'll also look at some important considerations when storing our features that can dramatically improve the speed of our queries.

Feature extraction and indexing

> ***Definition:* Feature extraction** *is the process of quantifying your dataset by applying an image descriptor to extract features from each and every image in your dataset. Normally, these features are stored on disk for later use and* **indexed** *using a specialized data structure (such as an inverted index, kd-tree, or random projection forest) to facilitate faster queries.*

Feedback

Using our flowers database example above, our goal is to simply loop over each image in our dataset, extract some features, and store these features on disk.

Feature extraction is quite a simple concept in principle, but in reality it can become very complex, depending on the size and scale of your dataset. For comparison purposes, we would say that the Flowers 17 dataset is *small* — it has a total of only 1,360 images (17 categories x 80 images per category). By comparison, image search engines such as TinEye (http://www.tineye.com/) have image datasets that number in the ***billions***.

Let's start with the first step of feature extraction and indexing: *instantiating your descriptor.*

1. Instantiate your descriptor

If you've ever read through my "*Hobbits and Histograms (http://www.pyimagesearch.com/2014/01/27/hobbits-and-histograms-a-how-to-guide-to-building-your-first-image-search-engine-in-python/)*" post on the PyImageSearch blog, you'll notice I mentioned that I like to abstract my image descriptors as *classes* rather than *functions*.

Furthermore, I like to put relevant parameters (such as the number of bins in a histogram) in the *constructor* of the class.

Why do I bother doing this?

The reason for using a class (with descriptor parameters in the constructor) rather than a function is because it helps ensure that the *exact same descriptor* with the *exact same parameters* is applied to each and every image in my dataset.

This is especially useful if I ever need to write my descriptor to disk using a Python library like `cPickle` and load it back up again further down the line, such as when a user is performing a query.

In order to compare two images, you need to represent them in the same consistent manner using your image descriptor. It wouldn't make sense to extract a histogram with 32 bins from one image and then a histogram with 128 bins from another image if your intent is to compare the two for similarity.

Feedback

For example, let's take a look at the skeleton code of a generic image descriptor in Python:

Generic image descriptor skeleton — Python

```
class GenericDescriptor:
    def __init__(self, paramA, paramB):
        # store the parameters for use in the 'describe' method
        self.paramA = paramA
        self.paramB = paramB

    def describe(self, image):
        # describe the image using self.paramA and self.paramB
        # as supplied in the constructor
        pass
```

The first thing you notice is the `__init__` method. Here, I provide my relevant parameters for the descriptor.

Next, you see the `describe` method. This method takes a single parameter: the `image` we wish to describe.

Whenever I call the `describe` method, I know that the parameters stored during the constructor will be used for each and every image in my dataset. This ensures my images are described consistently with identical descriptor parameters.

While the class vs. function argument doesn't seem like it's a big deal right now, when you start building larger, more complex image search engines that have a large codebase, using classes helps ensure that your descriptors are consistent.

2. Serial or parallel?

A better title for this sub-section might be "Single-core or Multi-core?"

Inherently, extracting features from images in a dataset is a task that can be made parallel.

Depending on the size and scale of your dataset, it might make sense to utilize multi-core processing techniques to split up the extraction of feature vectors from each image between multiple cores/processors.

However, for small datasets using computationally simple image descriptors, such as color histograms, using multi-core processing is not only overkill, it also adds extra complexity to your code.

Feedback

This is especially troublesome if you are just getting started working with computer vision and image search engines. So why bother adding extra complexity? Debugging programs with multiple threads/processes is substantially harder than debugging programs with only a single thread of execution.

Unless your dataset is quite large (or your dataset is small and your image descriptors are quite expensive to run) and could greatly benefit from multi-core processing, I would stay away from splitting the indexing task up into multiple processes for the time being. It's not worth the headache — ***yet***. Later in this module we'll explore how to safely and reliably split feature extraction across multiple cores of a system. We'll also learn how to leverage Hadoop to perform feature extraction from large datasets.

3. Writing to disk

This step might seem a bit obvious, but if you're going to go through all the effort to extract features from your dataset, it's best to write your index to disk for later use.

For small datasets, using a simple Python dictionary will likely suffice. The key can be the image filename (assuming that you have unique filenames across your dataset) and the value the features extracted from that image using your image descriptor. Finally, you can dump the index to file using `cPickle` .

If your dataset is larger or you plan to manipulate your features further (i.e. scaling, normalization, dimensionality reduction), you might be better off leveraging a `.csv` file (like we did in our **earlier lesson (https://gurus.pyimagesearch.com/lessons/your-first-image-search-engine/)**) or using h5py (http://www.h5py.org/) to write your features to disk.

My personal rule is that if the dataset is under 10,000 images (or less than 8gb of feature vectors), I use `cPickle` since it is much more simplistic. However, if I have over 10,000 images, my resulting feature vectors are larger than 8gb, or if I plan further processing my features, I will almost always go with h5py.

If you haven't used it yet, h5py is an amazing tool that allows you to write and access ***gigantic*** arrays to disk — arrays so large that they may not even fit into main memory. The h5py library also has the benefit of being is *extremely fast*. All this speed does come at a price, though. When creating a h5py dataset, the number of feature vectors (i.e. rows in your matrix) must be known ahead of time. Once the dataset has been created, you cannot (easily) resize it and accommodate more/less feature vectors.

Feedback

So given these particular caveats, is one method better than the other?

To be totally honest, it really depends.

If you're just starting off in computer vision and image search engines and you have a small dataset, I would use Python's built-in dictionary type and `cPickle` for the time being. Another good starter alternative is to use simple `.csv` files.

But if you already have experience in the field and have experience with NumPy, then I would suggest giving `h5py` a try and then comparing it to the dictionary approach mentioned above.

For the time being, I will be using a combination of `csv` and `cPickle` in my code examples; however, as we move to the more advanced topics in this module, we'll switch over to `h5py` .

4. Indexing and specialized data structures

At this point, we have (1) selected an image descriptor, (2) applied our image descriptor to every image in our dataset, and (3) written the resulting feature vectors to file.

We're now ready to perform a search, right?

Well, not quite. We still have to **define your similarity metric (https://gurus.pyimagesearch.com/topic/3-3-3-defining-your-similarity-metric/)**, but there's another step we should consider: *efficiency*.

As we saw from our previous lesson on building a **basic CBIR system (https://gurus.pyimagesearch.com/lessons/your-first-image-search-engine/)**, performing a search required that we compare our *query features* to ***each and every feature vector in the database***. Regardless of whether you are using a `.csv` file, a `cPickle` 'd Python dictionary, or a optimized `h5py` dataset, this is still a *O(N)* (linear) operation. As the size of our searchable dataset increases, the amount of time it takes for a search to perform will increase as well. It's not a great situation to be in if you're trying to optimize your CBIR system.

Luckily, we have optimized data structures that can facilitate faster querying at search time. By using methods such as *inverted indexes*, *kd-trees*, and *random projection forests*, we can knock down that linear *O(N)* search time to a sub-linear *O(log N)*.

Sound too good to be true?

It's actually not. These types of algorithms are used by large search companies such as Google, Bing, and DuckDuckGo *every single day*.

We won't dive into these algorithms right this second. Just keep in the back of your mind that after performing *feature extraction*, we'll often pass the resulting feature vectors to specialized data structures to speed up our searches. This step is entirely optional, but for large datasets, the performance gains are often worth the hassle.

Summary

This lesson, we explored (at a high level) how to extract features and index an image dataset. Feature extraction is the process of computing feature vectors from a dataset of images and then writing the features to persistent storage, such as your hard drive. Indexing then takes these feature vectors and applies specialized data structures to facilitate faster querying.

Feedback

The first step to feature extraction/indexing a dataset is to determine which image descriptor you are going to use. You need to ask yourself: what aspect of the images are you trying to characterize? The color distribution? The texture and tactile quality? The shape of the objects in the image?

After you have determined which descriptor you are going to use, you need to loop over your dataset and apply your descriptor to each and every image in the dataset, extracting feature vectors. This can be done either serially or parallel by utilizing multi-processing techniques.

Finally, after you have extracted features from your dataset, you need to write your index of features to file. Simple methods include using Python's built-in dictionary type and `cPickle` or using a basic `.csv` file. More advanced options include using `h5py` .

In our next lesson, we'll move on to the third step in building an image search engine: **defining your similarity metric**.

Quizzes	Status
1	Feature Extraction and Indexing Quiz (https://gurus.pyimagesearch.com/quizzes/feature-extraction-and-indexing-quiz/)

← Previous Topic (https://gurus.pyimagesearch.com/topic/defining-your-image-descriptor/) Next Topic → (https://gurus.pyimagesearch.com/topic/defining-your-similarity-metric/)

Course Progress

Ready to continue the course?

Click the button below to **continue your journey to computer vision guru**.

I'm ready, let's go! (/pyimagesearch-gurus-course/)

Resources & Links

Feedback

- PyImageSearch Gurus Community (https://community.pyimagesearch.com/)
- PyImageSearch Virtual Machine (https://gurus.pyimagesearch.com/pyimagesearch-virtual-machine/)
- Setting up your own Python + OpenCV environment (https://gurus.pyimagesearch.com/setting-up-your-python-opencv-development-environment/)
- Course Syllabus & Content Release Schedule (https://gurus.pyimagesearch.com/course-syllabus-content-release-schedule/)
- Member Perks & Discounts (https://gurus.pyimagesearch.com/pyimagesearch-gurus-discounts-perks/)
- Your Achievements (https://gurus.pyimagesearch.com/achievements/)
- Official OpenCV documentation (http://docs.opencv.org/index.html)

Your Account

- Account Info (https://gurus.pyimagesearch.com/account/)
- Support (https://gurus.pyimagesearch.com/contact/)
- Logout (https://gurus.pyimagesearch.com/wp-login.php?action=logout&redirect_to=https%3A%2F%2Fgurus.pyimagesearch.com%2F&_wpnonce=5736b21cae)

Search

© 2018 PyImageSearch. All Rights Reserved.

Feedback

/