Feature Matching methods comparison in OpenCV

Introduction

First, let's remind ourselves how we can extract and match features from two images. Humans have a natural ability to recognize different objects. We can do this because our brain is triggered by the most distinct features of an image. In image processing, these feature points will assist us to compare and detect objects in images or videos.

To extract the features from an image we can use several common feature detection algorithms. In this post we are going to use two popular methods: Scale Invariant Feature Transform (SIFT), and Oriented FAST and Rotated BRIEF (ORB). For feature matching, we will use the Brute Force matcher and FLANN-based matcher. So, let's begin with our code.

1. **Brute-Force Matching with ORB detector**

we are going to extract features using **Oriented FAST and Rotated BRIEF (ORB)** detector and we will use the Brute-force method for feature matching. First, let's import the necessary libraries and load our images. Also, we will convert images into grayscale.

```
import cv2
import numpy as np
from matplotlib import pyplot as plt

cv2_imshow(img1_gray)


def display_image(window_name, specified_image):
  """This function display any image from a given path"""
  cv2.imshow(window_name, specified_image)
  cv2.waitKey(0)



img1 = cv2.imread("Picture1.jpg")
img2 = cv2.imread("Picture2.jpg")

img1_gray = cv2.cvtColor(img1, cv2.COLOR_BGR2GRAY)
img2_gray = cv2.cvtColor(img2, cv2.COLOR_BGR2GRAY)


display_image(window_name="gray image", specified_image=img1_gray)

display_image(window_name="gray image", specified_image=img2_gray)
```

So, this should display the first image. Now, let's show the second image.

```
cv2_imshow(img2_gray)
```

Now, we're going to apply the Brute-Force matching with ORB descriptors. First, we need to create the ORB detector using the function `cv2.ORB_create()`

```
# Create our ORB detector and detect keypoints and descriptors
orb = cv2.ORB_create()
```

This is now our detector object. Next, we will detect keypoints and descriptors using the function orb.detectAndCompute(). We are going to pass two parameters. The first parameter is the input image and the second parameter is the mask. In our case, we are going to pass None because we're don't need a mask here. Then, we are going to do the same thing for the second image.

Find keypoints and descriptors with ORB

keypoints1, descriptors1 = orb.detectAndCompute(img1, None)

keypoints2, descriptors2 = orb.detectAndCompute(img2, None)

The next step is to create the BFMatcher object using the cv2.BFMatcher_create() function. This function consists of an optional parameter normType that specifies the distance as a measurement of similarity between two descriptors. For binary string-based descriptors like ORB, we usually use cv.NORM_HAMMING. This parameter calculates the Hamming distance between the arrays. **The Hamming distance between two strings of equal length is the number of positions at which the corresponding symbols are different**. To better understand this distance metric, have a look at the following image.

The second parameter is crossCheck. By default, it is set to False. In this case, BFMatcher will find the *k* nearest neighbors for each query descriptor. On the other hand, if crossCheck==True, then the knnMatch() method will return only the best matches. It will return matches with values (*i*,*j*) such that *ith* descriptor in a set *A* (descriptors from the first image) has j-th descriptor in a set *B* (descriptors from the second image) as the best match and vice-versa.

```
# Create a BFMatcher object.
# It will find all of the matching keypoints on two images
bf = cv2.BFMatcher_create(cv2.NORM_HAMMING,crossCheck=True)
```

Now, we can check where the matches occur by using the function bf.match(). Then, we are going to pass descriptors of the first image, and descriptors of the second image.

```
matches = bf.match(descriptors1, descriptors2)
```

So, now we have our matches. The next step is to sort them according to their distance. In our code, we will use the function sorted(). It has one required parameter iterable (in our case that are our matches) and several optional parameters. Another parameter that we use in our code is key. We use this parameter to decide the sort order (by default, this argument is set to None). In our case, we will use lambda x:x.distance expression. This expression will return the sorted list containing the items from the

matches. In that way, the items will be sorted in ascending order of their distances so that best matches (with low distance) come to the front.

To better understand this let's have a look at these matches. All matches have a lot of different attributes like a training index query index, but more importantly, they have a distance attribute. So, we can print the distance of a one-match using the following code.

```
single_match = matches[0]
single_match.distance
```

A smaller distance means that it is a better match, and the longer distance means that it is a less likely match. Therefore, with the following expression, we're going to sort our matches by that distance attribute.

```
matches = sorted(matches,key=lambda x:x.distance)
```

Now, let's continue with our post. The next step is to draw our matches using the function draw_matches(). This function consists of the following parameters:

img1, img2 – the input images
keypoints1 – keypoints detected in the first image
keypoints2 – keypoints detected in the second image
matches – all detected matches

In case you don't want to draw all matches you can choose the best ones at the top of the list of matches. We will choose the first 30 by indexing parameter `matches` in our function, The next parameter we're going to provide is `mask` . We are going to pass `None` because we don't want to use a mask here. And then finally, we'll set parameter `flags=2` . In that way, we will define how to draw the matching points.

```
ORB_matches =cv2.drawMatches(img1, keypoints1, img2, keypoints2,
matches[:30], None, flags=2)
cv2_imshow(ORB_matches)
```

2. Brute-Force Matching with SIFT detector and Ratio test

Now, we are going to run a similar code. However, this time we're going to do is use **Scale Invariant Feature Transform (SIFT)** descriptors. This method is perfectly suitable for our goal because **it detects features that are invariant to image scale and rotation**. Moreover, features are local and based on the appearance of the object in certain interesting points. They are also robust to changes in illumination, noise, and minor changes in viewpoint.

SIFT was first presented in 2004, by David G. Lowe from the University of British Columbia in the paper, Distinctive Image Features from Scale-Invariant Keypoints, This algorithm was patented several years ago, and since then it is included in the non-free module in OpenCV. That is why we need to install the older version of OpenCV because SIFT is not included in the new OpenCV library.

```
!pip install opencv-python==3.4.2.16
!pip install opencv-contrib-python==3.4.2.16
```

Now we will continue by creating a SIFT object with the function

```
cv2.xfeatures2d.SIFT_create()
```

```
# Create our SIFT detector and detect keypoints and descriptors
sift = cv2.xfeatures2d.SIFT_create()
```

Then, just like we did with ORB, we're going to find the keypoints and descriptors with SIFT.

```
# Find the key points and descriptors with SIFT
keypoints1, descriptors1 = sift.detectAndCompute(img1, None)
keypoints2, descriptors2 = sift.detectAndCompute(img2, None)
```

Next, we're going to calculate these matches using the Brute-Force method.

```
bf = cv2.BFMatcher()
```

Now, we're going to calculate our matches using a slightly different function `bf.knnMatch()`. This function finds the k best matches of number for each descriptor from a query set.

The function bf.knnMatch() provides two matches for each of these descriptors when the parameter k=2.

```
matches = bf.knnMatch (descriptors1, descriptors2,k=2)
```

To keep only the strong matches, we will use David Lowe's ratio test. Lowe proposed this ratio test in order to increase the robustness of the SIFT algorithm. The goal of this test is to get rid of the points that are not distinct enough. The general idea is that there needs to be enough difference between the first best match and the second-best matches.

Now by using indexing we can extract one first and one second-best match and compare their distance measurements.

```
AA1 = matches[131][0]
AA1.distance
```

```
AA2 = matches[131][1]
AA2.distance
```

Here we can see that distance of the first best match is far away from the distance of the second match. Therefore, the entire descriptor of that point is probably distinct enough and it should be a good match.

On the other hand, if the first best match is pretty close to the second match, then this point probably is not distinct enough.

```
BB1 = matches[1][0]
BB1.distance
```

```
BB2 = matches[1][1]
BB2.distance
```

Now, we can apply David Lowe's ratio test.

```
good_matches = []

for m1, m2 in matches:
  if m1.distance < 0.6*m2.distance:
    good_matches.append([m1])
```

So, we created a loop that defines which matches will be discarded and which matches will be preserved. So, if m1 distance is less than 60% of m2 distance, then the descriptor for that particular row will be preserved. On the other hand, if m1 distance is greater than 60% of m2 distance, then it's probably not a good match, and the descriptor for that particular row will be discarded. Therefore, less distance means a better match.

Now, let's print our good matches.

```
good_matches
```

Also, we can print the length of that list. We can see that we actually discarded a large number of poor matches.

```
len(matches)
len(good_matches)
```

Now it's time to draw these matches and see how they performed. This time we will use a similar function `cv2.drawMatchesKnn()`. We will pass the same parameters as we did with the ORB detector.

```
SIFT_matches =cv2.drawMatchesKnn(img1, keypoints1, img2, keypoints2,
good_matches, None, flags=2)
display_image(window_name='ORB Match',specified_image=ORB_matches)
```

3. FLANN based Matcher

Using a FLAN based matcher instead of a Brute-Force matcher we're going to introduce a more complex parameter drawing mechanism, that allows us to draw only the clear matches.

First, we are going to detect features with SIFT and this part of the code will be identical to the previous one.

```
import cv2
import numpy as np
```

```
from matplotlib import pyplot as plt

def display_image(window_name, specified_image):
    """This function display any image from a given path"""
    cv2.imshow(window_name, specified_image)
    cv2.waitKey(0)

img1 = cv2.imread("image1.png")
img2 = cv2.imread("image2.png")

img1_gray = cv2.cvtColor(img1, cv2.COLOR_BGR2GRAY)
img2_gray = cv2.cvtColor(img2, cv2.COLOR_BGR2GRAY)

display_image(window_name="gray image", specified_image=img1_gray)
display_image(window_name="gray image", specified_image=img2_gray)

#First, we are going to detect features with SIFT.
# Create our SIFT detector and detect keypoints and descriptors
sift = cv2.xfeatures2d.SIFT_create()

# Find the key points and descriptors with SIFT
keypoints1, descriptors1 = sift.detectAndCompute(img1, None)
keypoints2, descriptors2 = sift.detectAndCompute(img2, None)
```

Now, we need to define the FLANN parameters. FLANN stands for Fast Library for Approximate Nearest Neighbors. It contains a collection of algorithms optimized for fast nearest neighbor search in large datasets and for high dimensional features. It works faster than BFMatcher for large datasets. However, it's not going to find the best possible matches. Instead, it's just going to find good matching candidates.

In order to try to increase the precision or the quality of the matches, we can play with FLANN parameters. However, that will slow down the algorithm. That is why we will use the following default parameters for a FLANN matcher.

To construct FLANN parameters we will use a **k-dimensional tree** which is the alternative way to organize data structures. In computer science, a k-dimensional tree is a space-partitioning data structure for organizing points in a k-dimensional space.

We will set `FLAN_INDEX_KDTREE = 0` . Also, we are going to define two sets of parameters: index parameters, and search parameters. In the `index_params` we will create a dictionary by passing `FLAN_INDEX_KDTREE` into the `dict()` dictionary algorithm. We will also set the number of trees which in our case is equal to 5. Finally, we will say that `search_params` is equal to the dictionary, and we'll set parameter `checks=50`.

```
#construct FLANN parameters we will use a k-dimensional tree

#which is the alternative way to organize data structures.

FLAN_INDEX_KDTREE = 0
```

```
index_params = dict (algorithm = FLAN_INDEX_KDTREE, trees=5)

search_params = dict (checks=50)
```

So, now we have our parameters. Next, we create the FLANN based matcher object using the function `cv2.FlannBasedMatcher()`. As parameters, we will pass index parameters as well as the search parameters.

```
#create the FLANN based matcher object using the function
cv2.FlannBasedMatcher()

flann = cv2.FlannBasedMatcher(index_params, search_params)
```

To calculate matches we will use `flann.knnMatch` the k nearest neighbor matches. Next, we will create a similar ratio test as we did with the SIFT detector.

```
#calculate matches we will use flann.knnMatch the k nearest neighbor matches.
#Next, we will create a similar ratio test as we did with the SIFT detector
matches = flann.knnMatch (descriptors1, descriptors2,k=2)
good_matches = []

for m1, m2 in matches:
  if m1.distance < 0.6 * m2.distance:
    good_matches.append([m1])

good_matches
```

Now, when we have our good matches it's time to draw them. Note, that parameter `flags=2` and we can also set this parameter to `flags=0`. In the first case, that function will draw only lines between matching points.

```
#Now it's time to draw these matches and see how they performed.

#This time we will use a similar function cv2.drawMatchesKnn().

flann_matches   =cv2.drawMatchesKnn(img1,   keypoints1,   img2,   keypoints2,
good_matches, None, flags=2)

display_image(window_name="Flann Matches", specified_image=flann_matches)
```

Now let's pass several additional parameters that will help us to better visualize our matches. We can draw these lines in one specific color, and then show all those points from `flag=0` as a different color. The way we can do that is by simply adding the parameter `mask`. So, the first several lines of code will be identical.

```
def display_image(window_name, specified_image):
    """This function display any image from a given path"""
    cv2.imshow(window_name, specified_image)
    cv2.waitKey(0)

img1 = cv2.imread("image1.png")
```

```
img2 = cv2.imread("image2.png")

img1_gray = cv2.cvtColor(img1, cv2.COLOR_BGR2GRAY)
img2_gray = cv2.cvtColor(img2, cv2.COLOR_BGR2GRAY)

display_image(window_name="gray image", specified_image=img1_gray)
display_image(window_name="gray image", specified_image=img2_gray)

#First, we are going to detect features with SIFT.
# Create our SIFT detector and detect keypoints and descriptors
sift = cv2.xfeatures2d.SIFT_create()

# Find the key points and descriptors with SIFT
keypoints1, descriptors1 = sift.detectAndCompute(img1, None)
keypoints2, descriptors2 = sift.detectAndCompute(img2, None)
```

If we take a look at this object, we can see that it is a matrix that consist of only zeros.

```
FLAN_INDEX_KDTREE = 0

index_params = dict (algorithm = FLAN_INDEX_KDTREE, trees=5)

search_params = dict (checks=50)

flann = cv2.FlannBasedMatcher(index_params, search_params)

matches = flann.knnMatch (descriptors1, descriptors2, k=2)

#create a new object called matchesMask.

matchesMask = [[0,0] for i in range(len(matches))]

matchesMask
```

Now, we want to change some of these zeros to one, depending on if we have a good match or not. To do that we need to slightly modify our ratio test. We will change `m1` and `m2` to a single tuple by putting parentheses around it. And then we will enumerate our matches. In that way, we can actually keep track of the index marker. Then, we can create the mask by setting `matchesMask` at index `i` to be equal to `[1,0]`. By doing this we are going to label lines where we actually have a good match as 1. So, all we're doing is we have this giant list of zeros.

```
#Now, we want to change some of these zeros to one, depending on if we have a
good match or not.

#To do that we need to slightly modify our ratio test.

for i,(m1, m2) in enumerate (matches):

    if m1.distance < 0.5 * m2.distance:
```

```
        matchesMask[i] = [1,0]
```

Now we're going to use this mask to create a drawing parameter dictionary. Then, we specify the color of single points and the color of the line. Next, we're going to do is say pass in the `matchesMask` which defines where we had a match and where we just have a single point. Finally, we will set `flags=0` because we want to show all these single points.

```
draw_params = dict (matchColor = (0,0,255), singlePointColor = (0,255,0), matchesMask = matchesMask, flags=0)
```

Now, we need to make just a few corrections to the function `cv2.drawMatchesKnn()` First, instead of `good_matches` we will pass in all matches. In that way instead of appending matches to a list to a list `good_matches`, we will use `matchMask` and `draw_params` to filter them out based on their color. Also, instead of parameter flags, we will write `**draw_params`.

```
flann_matches = cv2.drawMatchesKnn(img1, keypoints1, img2, keypoints2, matches, None,**draw_params)

display_image(window_name="Flann Matches", specified_image=flann_matches)
```