

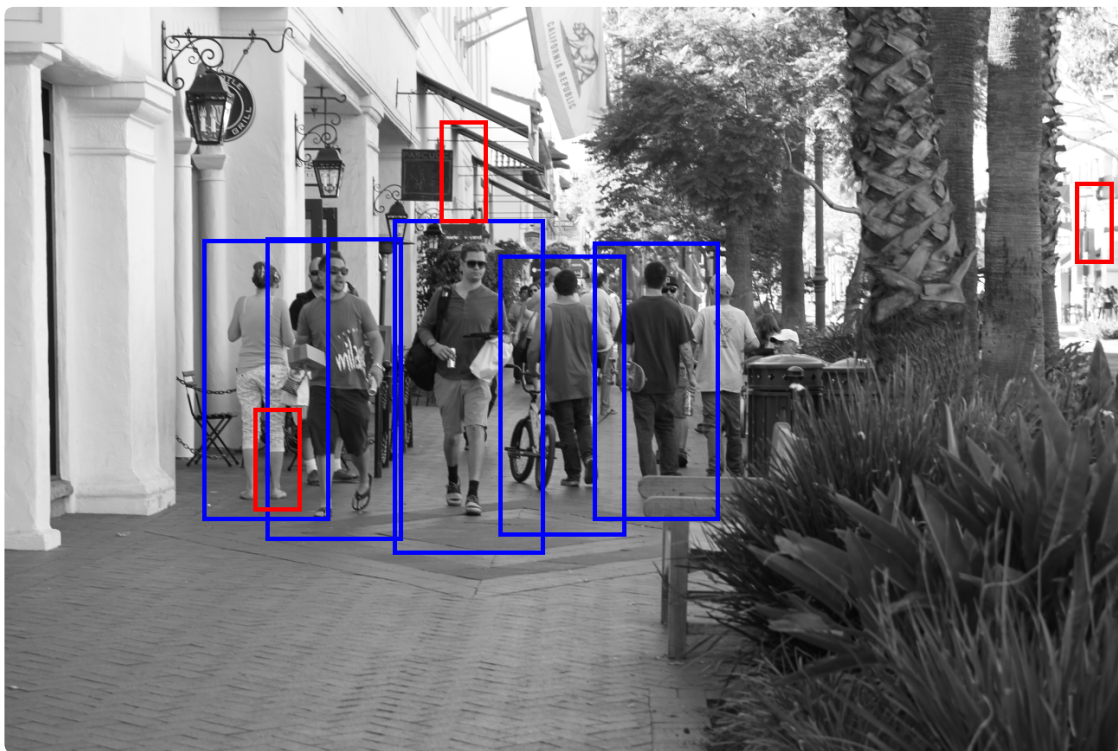
Chris McCormick [About](#) [Tutorials](#) [Archive](#)

OpenCV HOG Detector: Result Clustering

07 Nov 2013

The final step of the HOG detection process is to cluster the search results.

Here is the result of running the OpenCV HOG detector on a frame with and without result clustering. I've colored the results blue or red based on whether they represent a true positive or a false positive, respectively.



With clustering



Without clustering

The above searches were performed with a scaling factor of 1.05 (increase the detection window size by 1.05 for each search scale). For the clustering, I used a minimum cluster size of 3 and used the default window overlap method for clustering.

Result clustering has two purposes. The first is to deal with multiple detections of the same person at different scales or at slightly shifted detection windows. Ideally we'd like to have just one detection per person.

The second purpose of result clustering is to reduce false positives. If we perform a dense search of the image, we expect to detect each person multiple times at different scales and possibly with slightly shifted detection windows. If a window is only recognized once, it's more likely to be a false positive that we want to reject.

In the above images, note the false positives towards the bottom right and top left of the image which are effectively eliminated by clustering.

There are also a large number of "body part" recognitions towards the center of the image which end up folded into the true positive recognitions because of the clustering step.

The OpenCV HOG detector supports two different methods of result clustering. The default mode isn't given a name in the OpenCV documentation that I can find, so I'll just call it the "Window Overlap" method. The second mode is called "Mean Shift Grouping".

If you look in `objdetect.hpp` in the OpenCV source, you'll see that the final argument to `HOGDescriptor::detectMultiScale` is a parameter `'bool useMeanshiftGrouping = false'`. So the HOG detector uses the Window Overlap method by default.

The Window Overlap method simply uses the rule that if two detection windows overlap sufficiently, then they belong to the same cluster. The Mean Shift Grouping method takes a more complex approach which incorporates the detection confidences from the SVM classifier.

I haven't had much luck finding documentation or articles describing how the two clustering methods are implemented in detail, so I've had to resort to inspecting the code to gain some understanding. I'll be referencing the OpenCV source files in my explanations below.

For now, this post just covers the window overlap method, but I hope to dig into the mean shift approach as well.

Window Overlap Method

OpenCV Source

The following OpenCV files and functions are relevant to the result clustering task.

- `modules\objdetect\src\hog.cpp`
- `modules\objdetect\src\cascadedetect.cpp`
- `modules\objdetect\include\opencv2\objdetect\objdetect.hpp`
- `modules\core\include\opencv2\core\operations.hpp`

Function / Class	**Definition**	**Declaration**
<code>HOGDescriptor::detectMultiScale</code>	<code>hog.cpp</code>	<code>objdetect.hpp</code>
<code>groupRectangles</code>	<code>cascadedetect.cpp</code>	<code>objdetect.hpp</code>
<code>SimilarRects</code>	<code>cascadedetect.cpp</code>	<code>objdetect.hpp</code>
<code>partition</code>	<code>operations.hpp</code>	<code>operations.hpp</code>

The last step of `'detectMultiScale'` is to pass all of the detected windows to the `'groupRectangles'` function to perform the clustering.

The last parameter to `groupRectangles` is epsilon `'eps'`, which controls how much overlap is required. The default value of `'eps'` is set to 0.2 in its declaration (in `objdetect.hpp`). The OpenCV HOG detector does not modify this `'eps'` value.

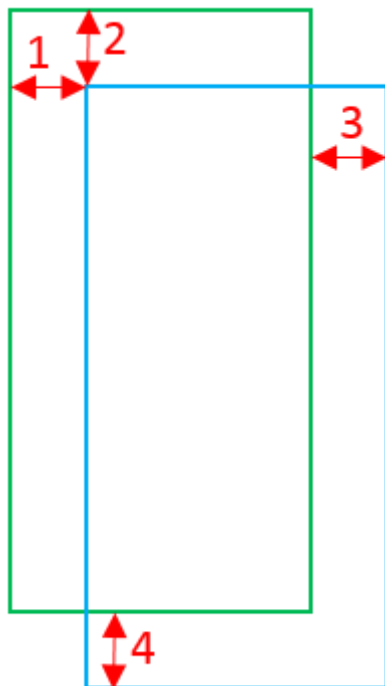
The 'groupRectangles' function essentially calls down to the 'partition' function to do the clustering. The 'SimilarRects' class defines the metric for determining whether two rectangles belong to the same cluster.

Algorithm

To determine if two rectangular detection windows belong in the same cluster, first a value 'delta' is computed using the following equation, (where epsilon is 0.2):

$$\delta = \varepsilon * \frac{\min(w_1, w_2) + \min(h_1, h_2)}{2}$$

The rectangles correspond to a single detection if each of these four labeled distances is less than or equal to delta:



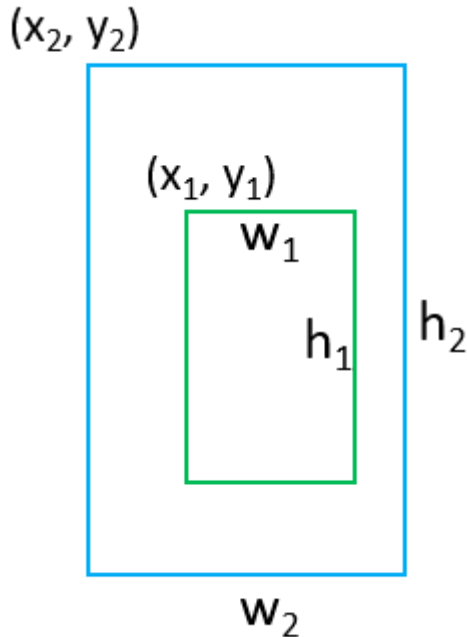
Note that if epsilon is smaller, then delta is smaller, and we are requiring more overlap in order to place both rectangles in the same cluster.

In the above illustration, the rectangles are 1 in. x 2 in., and are offset by 0.25 in. This gives a delta of 0.3 in., so these two rectangles are close to the limit of what is considered a match (using the default epsilon).

Note that if the two rectangles are very different sizes, they may not be put in the same cluster, even if they overlap entirely. This situation is handled later in the 'groupRectangles' function.

After calling partition to cluster the rectangles, the 'groupRectangles' function goes through the results and rejects any clusters with \leq 'groupThreshold' results. It also computes the average position and dimensions of the rectangle to use as the final results.

Finally, it looks for small rectangles that are encompassed by larger ones and removes the smaller rectangles from the results. Consider the below two rectangles:



To test if rectangle 1 should be removed because it's encompassed by rectangle 2, the 'groupRectangles' function uses the following logic.

First, it defines two deltas:

$$dx = \varepsilon * w_2$$

$$dy = \varepsilon * h_2$$


Then, it filters out rectangle 2 if all of the following hold:

```
x1 >= (x2 - dx) &&  
y1 >= (y2 - dy) &&  
(x1 + w1) <= (x2 + w2 + dx) &&  
(y1 + h1) <= (y2 + h2 + dy) &&  
(n2 > max(3, n1) || n1 < 3)
```

The fifth condition ensures that we don't remove rectangle 1 if it has more cluster members than rectangle 2, or if rectangle 2 has only 3 or fewer cluster members. The number of members in each cluster is a measure of our confidence in the result, so we don't want to filter out a high-confidence result just because it's encompassed by a larger rectangle. Going back to the two images at the top of the post, the woman's leg is preserved as a result (even though it's encompassed by another rectangle) because of the high number of results in that cluster.

3 Comments

mccormickml.com

 Login ▾ Recommend 3 Share

Sort by Best ▾



Join the discussion...

**Bartek** • 4 months ago

Great article, exactly what I was looking for! I had to rewrite this algorithm without using external libraries (opencv) and it works exactly the same like original.

^ | ▾ • Reply • Share ›

**Chris McCormick** Mod → Bartek • 4 months ago

Very cool! What language did you write it in?

^ | ▾ • Reply • Share ›

**Bartek** → Chris McCormick • 3 months ago

In C++. I use Zynq platform and I didn't want to run Linux on it (I think it would be na overkill), so I wrote it in "pure" C++ to run it as a bare metal application. Pedestrian detection is based on HOG as well, however that part is running in FPGA part of Zynq.

^ | ▾ • Reply • Share ›

ALSO ON MCCORMICKML.COM

Radial Basis Function Network (RBFN) Tutorial

21 comments • a year ago•



ايهاب ربايعة ابو مهند — amazingthanks for informative explanations

Deep Learning Tutorial - Sparse Autoencoder

3 comments • a year ago•



Choung young jae — For a given hidden node, it's average activation value (over all the training samples)

RBFN Tutorial Part II - Function Approximation

9 comments • a year ago•



Chris McCormick — Great, just sent you an e-mail!

AdaBoost Tutorial

9 comments • a year ago•



Huey Kwik — This is really well-explained, thank you!! thought showing graphs of alpha vs. error

Related posts

Concept Search on Wikipedia 22 Feb 2017

[Getting Started with mlpack](#) 01 Feb 2017

[Word2Vec Tutorial Part 2 - Negative Sampling](#) 11 Jan 2017

© 2017. All rights reserved.