Selective Search for Object Detection (C++ / Python)


Vaibhaw Singh Chandel

In this tutorial, we will understand an important concept called “Selective Search” in Object Detection. We will also share OpenCV code in C++ and Python.

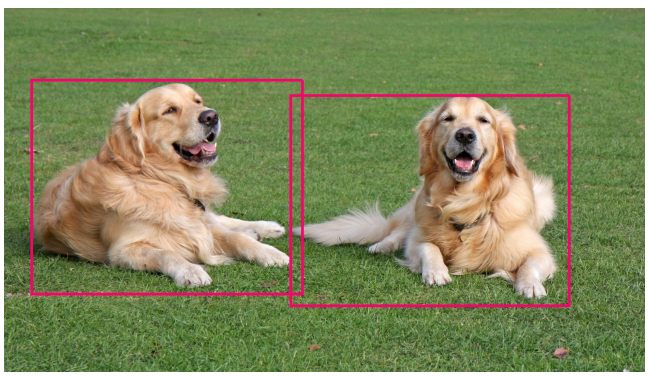
Object Detection vs. Object Recognition

An object recognition algorithm identifies which objects are present in an image. It takes the entire image as an input and outputs class labels and class probabilities of objects present in that image. For example, a class label could be “dog” and the associated class probability could be 97%.

On the other hand, an object detection algorithm not only tells you which objects are present in the image, it also outputs bounding boxes (x, y, width, height) to indicate the location of the objects inside the image.

At the heart of all object detection algorithms is an object recognition algorithm. Suppose we trained an object recognition model which identifies dogs in image patches. This model will tell whether an image has a dog in it or not. It does not tell where the object is located.

To localize the object, we have to select sub-regions (patches) of the image and then apply the object recognition algorithm to these image patches. The location of the objects is given by the location of the image patches where the class probability returned by the object recognition algorithm is high.



The most straightforward way to generate smaller sub-regions (patches) is called the Sliding Window approach. However, the sliding window approach has several limitations. These limitations are overcome by a class of algorithms called the “Region Proposal” algorithms. Selective Search is one of the most popular Region Proposal algorithms.

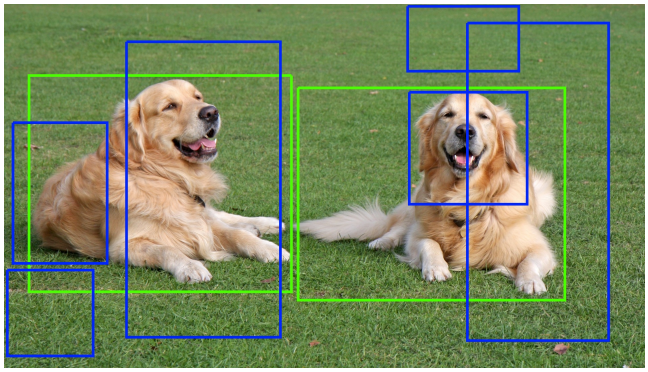
Sliding Window Algorithm

In the sliding window approach, we slide a box or window over an image to select a patch and classify each image patch covered by the window using the object recognition model. It is an exhaustive search for objects over the entire image. Not only do we need to search all possible locations in the image, we have to search at different scales. This is because object recognition models are generally trained at a specific scale (or range of scales). This results into classifying tens of thousands of image patches.

The problem doesn’t end here. Sliding window approach is good for fixed aspect ratio objects such as faces or pedestrians. Images are 2D projections of 3D objects. Object features such as aspect ratio and shape vary significantly based on the angle at which image is taken. The sliding window approach because computationally very expensive when we search for multiple aspect ratios.

Region Proposal Algorithms

The problems we have discussed so far can be solved using region proposal algorithms. These methods take an image as the input and output bounding boxes corresponding to all patches in an image that are most likely to be objects. These region proposals can be noisy, overlapping and may not contain the object perfectly but amongst these region proposals, there will be a proposal which will be very close to the actual object in the image. We can then classify these proposals using the object recognition model. The region proposals with the high probability scores are locations of the object.



Blue Boxes: False Positives; Green Boxes: True Positives

Region proposal algorithms identify prospective objects in an image using segmentation. In segmentation, we group adjacent regions which are similar to each other based on some criteria such as color, texture etc. Unlike the sliding window approach where we are looking for the object at all pixel locations and at all scales, region proposal algorithm work by grouping pixels into a smaller number of segments. So the final number of proposals generated are many times less than sliding window approach. This reduces the number of image patches we have to classify. These generated region proposals are of different scales and aspect ratios.

An important property of a region proposal method is to have a very **high recall**. This is just a fancy way of saying that the regions that contain the objects we are looking have to be in our list of region proposals. To accomplish this our list of region proposals may end up having a lot of regions that do not contain any object. In other words, it is ok for the region proposal algorithm to produce a lot of false positives so long as it catches all the true positives. Most of these false positives will be rejected by object recognition algorithm. The time it takes to do the detection goes up when we have more false positives and the accuracy is affected slightly. But having a high recall is still a good idea because the alternative of missing the regions containing the actual objects severely impacts the detection rate.

Several region proposal methods have been proposed such as

1. Objectness
2. Constrained Parametric Min-Cuts for Automatic Object Segmentation
3. Category Independent Object Proposals
4. Randomized Prim
5. Selective Search

Amongst all these region proposal methods Selective Search is most commonly used because it is fast and has a very high recall.

Selective Search for Object Recognition

What is Selective Search?

Selective Search is a region proposal algorithm used in object detection. It is designed to be fast with a very high recall. It is based on computing hierarchical grouping of similar regions based on color, texture, size and shape compatibility.

Selective Search starts by over-segmenting the image based on intensity of the pixels using a graph-based segmentation method by Felzenszwalb and Huttenlocher. The output of the algorithm is shown below. The image on the right contains segmented regions represented using solid colors.


Input Image


Output Image

Can we use segmented parts in this image as region proposals? The answer is no and there are two reasons why we cannot do that:

1. Most of the actual objects in the original image contain 2 or more segmented parts
2. Region proposals for occluded objects such as the plate covered by the cup or the cup filled with coffee cannot be generated using this method

If we try to address the first problem by further merging the adjacent regions similar to each other we will end up with one segmented region covering two objects.

Perfect segmentation is not our goal here. We just want to predict many region proposals such that some of them should have very high overlap with actual objects.

Selective search uses oversegments from Felzenszwalb and Huttenlocher's method as an initial seed. An oversegmented image looks like this.


Oversegmented Image

Selective Search algorithm takes these oversegments as initial input and performs the following steps

1. Add all bounding boxes corresponding to segmented parts to the list of regional proposals
2. Group adjacent segments based on similarity
3. Go to step 1

At each iteration, larger segments are formed and added to the list of region proposals. Hence we create region proposals from smaller segments to larger segments in a bottom-up approach. This is what we mean by computing “hierarchical” segmentations using Felzenszwalb and Huttenlocher’s oversegments.

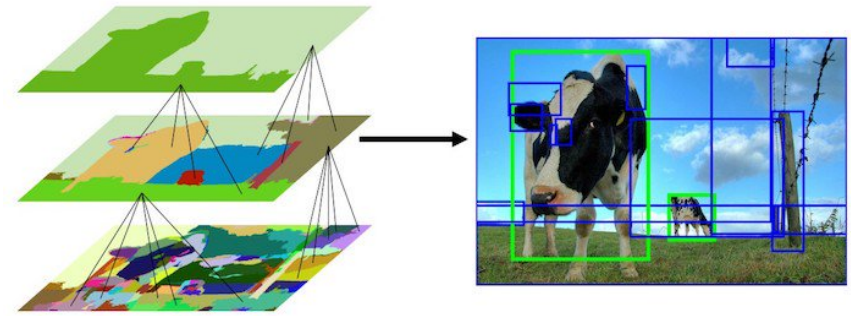


Image credit

This image shows the initial, middle and last step of the hierarchical segmentation process.

Similarity

Let’s dive deeper into how do we calculate the similarity between two regions.

Selective Search uses 4 similarity measures based on **color, texture, size and shape compatibility**.

Color Similarity

A color histogram of 25 bins is calculated for each channel of the image and histograms for all channels are concatenated to obtain a color descriptor resulting into a 25×3 = 75-dimensional color descriptor.

Color similarity of two regions is based on histogram intersection and can be calculated as:

$s_{color}(r_i, r_j) = \sum_{k=1}^{n} min(c_i^k, c_j^k)$

c_i^k is the histogram value for k^{th} bin in color descriptor

Texture Similarity

Texture features are calculated by extracting Gaussian derivatives at 8 orientations for each channel. For each orientation and for each color channel, a 10-bin histogram is computed resulting into a 10x8x3 = 240-dimensional feature descriptor.

Texture similarity of two regions is also calculated using histogram intersections.

$s_{texture}(r_i, r_j) = \sum_{k=1}^{n} min(t_i^k, t_j^k)$

t_i^k is the histogram value for k^{th} bin in texture descriptor

Size Similarity

Size similarity encourages smaller regions to merge early. It ensures that region proposals at all scales are formed at all parts of the image. If this similarity measure is not taken into consideration a single region will keep gobbling up all the smaller adjacent regions one by one and hence region proposals at multiple scales will be generated at this location only. Size similarity is defined as:

$s_{size}(r_i, r_j) = 1 - \frac{size(r_i) + size(r_j)}{size(im)}$

where $size(im)$ is size of image in pixels.

Shape Compatibility

Shape compatibility measures how well two regions r_i and r_j fit into each other. If r_i fits into r_j we would like to merge them in order to fill gaps and if they are not even touching each other they should not be merged.

Shape compatibility is defined as:

$s_{fill}(r_i, r_j) = 1 - \frac{size(BB_{ij}) - size(r_i) - size(r_j)}{size(im)}$

where $size(BBij)$ is a bounding box around $r_i$and r_j.

Final Similarity

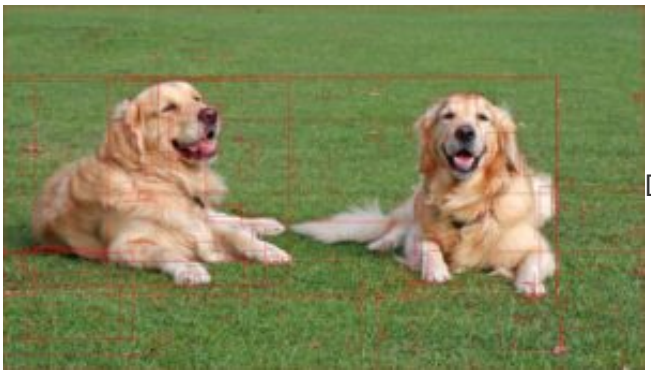
The final similarity between two regions is defined as a linear combination of aforementioned 4 similarities.

$$s(r_i, r_j) = a_1 s_{color}(r_i, r_j) + a_2 s_{texture}(r_i, r_j) + a_3 s_{size}(r_i, r_j) + a_4 s_{fill}(r_i, r_j)$$

where r_i and r_j are two regions or segments in the image and $a_i \in 0, 1]$ denotes if the similarity measure is used or not.

Results

Selective Search implementation in OpenCV gives thousands of region proposals arranged in decreasing order of objectness. For clarity, we are sharing results with top 200-250 boxes drawn over the image. In general 1000-1200 proposals are good enough to get all the correct region proposals.



Dogs: top 250 region proposals

Breakfast Table: top 200 region proposals

Selective Search Code

Let's take a look on how can we use Selective Search based segmentation implemented in OpenCV.

Selective Search: C++(略,查看原文)

Selective Search: Python

The code below is a Python tutorial for Selective Search using OpenCV 3.3. Note the bug alert for OpenCV 3.2 mentioned after the code block. Please read through the comments to understand the code.

```
`#!/usr/bin/env python``'''``Usage:``    ``./ssearch.py input_image (f|q)``    ``f=fast, q=quality``Use "l" to display less rects, 'm' to display more rects, "q" to quit.``'''` `import` `sys``import` `cv2` `if` `__name__ ``=``=` `'__main__'``:``    ``# If image path and f/q is not passed as command``    ``# line arguments, quit and display help message``    ``if` `len``(sys.argv) < ``3``:`` ``print``(__doc__)``        ``sys.exit(``1``)` `    ``# speed-up using multithreads``    ``cv2.setUseOptimized(``True``);`` ``cv2.setNumThreads(``4``);` `    ``# read image``    ``im ``=` `cv2.imread(sys.argv[``1``])``    ``# resize image``    ``newHeight ``=` `200``    ``newWidth ``=` `int``(im.shape[``1``]``*``200``/``im.shape[``0``])``    ``im ``=` `cv2.resize(im, (newWidth, newHeight))    ` `    ``# create Selective Search Segmentation Object using default parameters``    ``ss ``=` `cv2.ximgproc.segmentation.createSelectiveSearchSegmentation()` `    ``# set input image on which we will run segmentation`` ``ss.setBaseImage(im)` `    ``# Switch to fast but low recall Selective Search method``    ``if` `(sys.argv[``2``] ``=``=` `'f'``):``        ``ss.switchToSelectiveSearchFast()` `    ``# Switch to high recall but slow Selective Search method``    ``elif` `(sys.argv[``2``] ``=``=` `'q'``):`` ``ss.switchToSelectiveSearchQuality()``    ``# if argument is neither f nor q print help message``    ``else``:`` ``print``(__doc__)``        ``sys.exit(``1``)` `    ``# run selective search segmentation on input image``    ``rects ``=` `ss.process()`` ``print``(``'Total Number of Region Proposals: {}'``.``format``(``len``(rects)))``    ` `    ``# number of region proposals to show``    ``numShowRects ``=` `100``    ``# increment to increase/decrease total number``    ``# of reason proposals to be shown``    ``increment ``=` `50` `    ``while` `True``:``        ``# create a copy of original image``        ``imOut ``=` `im.copy()` `        ``# itereate over all the region proposals``        ``for` `i, rect ``in` `enumerate``(rects):``            ``# draw rectangle for region proposal till numShowRects``            ``if` `(i < numShowRects):``                ``x, y, w, h ``=` `rect`` ``cv2.rectangle(imOut, (x, y), (x``+``w, y``+``h), (``0``, ``255``, ``0``), ``1``, cv2.LINE_AA)``            ``else``:``                ``break` `        ``# show output`` ``cv2.imshow(``"Output"``, imOut)` `        ``# record key press``        ``k ``=` `cv2.waitKey(``0``) & ``0xFF` `        ``# m is pressed``        ``if` `k ``=``=` `109``:``            ``# increase total number of rectangles to show by increment``
```

```
    ``numShowRects ``+``=` `increment``        ``# l is pressed``
    ``elif` `k ``=``=` `108` `and` `numShowRects > increment:``
    ``# decrease total number of rectangles to show by increment``
        ``numShowRects ``-``=` `increment``        ``# q is pressed``
        ``elif` `k ``=``=` `113``:``            ``break``    ``# close
    image show window``    ``cv2.destroyAllWindows()`
```

Bug Alert: There was a bug in Python bindings of Selective Search which was fixed in this commit. So the Python code will work for OpenCV 3.3.0 but not for OpenCV 3.2.0.

If you don't want to compile OpenCV 3.3.0 and have the build folder for OpenCV 3.2.0 which you compiled earlier, you can fix this bug too. If you look at the Github commit it is just a small change. You have to change line#239 in file

opencv_contrib-3.2.0/modules/ximgproc/include/opencv2/ximgproc/segmentation.hpp

```
1  `// from``CV_WRAP ``virtual` `void` `process(std::vector<Rect>&
   rects) = 0;``// to``CV_WRAP ``virtual` `void` `process(CV_OUT
   std::vector<Rect>& rects) = 0;`
```

Now recompile your OpenCV 3.2.0 again. If you have a build folder in which OpenCV was compiled earlier, running the make command will just compile this module.

Subscribe & Download Code

If you liked this article and would like to download code (C++ and Python) and example images used in this post, please subscribe to our newsletter. You will also receive a free Computer Vision Resource Guide. In our newsletter, we share OpenCV tutorials and examples written in C++/Python, and Computer Vision and Machine Learning algorithms and news.