cv.HOGDescriptor/detectMultiScale

Detects objects of different sizes in the input image. The detected objects are returned as a list of rectangles.

`SvmDetector` should be set before calling this method.

The image is repeatedly scaled down by the specified `Scale` factor as long as it remains larger than `hog.WinSize` or until a maximum of `hog.NLevels` levels is built. The resized images are then searched with a sliding window to detect objects similar to the cv.HOGDescriptor.detect method (this method is parallelized). Finally the found rectangles are grouped and clipped against the image size.

작동 순서

(1) 임의의 크기의 사각형을 정의해서 부분 영상을 추출합니다. (Crop)

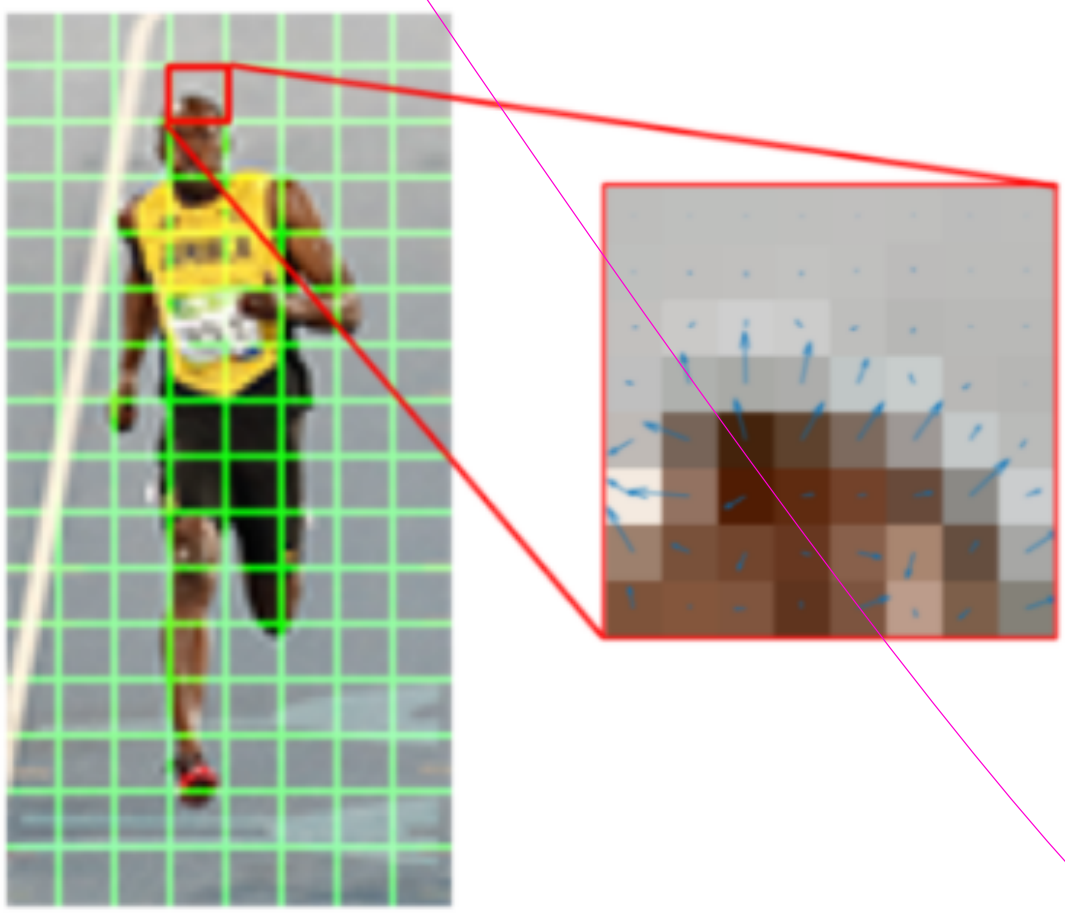
(2) 추출한 부분 영상의 크기를 정규화 합니다. (64X128)

(3) 64X128 영상의 그래디언트를 계산하여 방향 성분과 크기 성분을 파악합니다.

(4) 64X128 영상을 8X8 크기의 셀(cell)로 분할합니다.

(5) 각 셀마다 방향과 크기 성분을 이용하여 방향 히스토그램을 계산합니다.

(6) 각각의 셀에서 방향 성분을 9개로 구분하여 9가지 방향에 대한 히스토그램을 생성합니다.
(180도를 20도씩 9가지 방향, 대칭하면 360도)



2	3	4	4	3	4	2	2
5	11	17	13	7	9	3	4
11	21	23	27	22	17	4	6
23	99	165	135	85	32	26	2
91	155	133	136	144	152	57	28
98	196	76	38	26	60	170	51
165	60	60	27	77	85	43	136
71	13	34	23	108	27	48	110

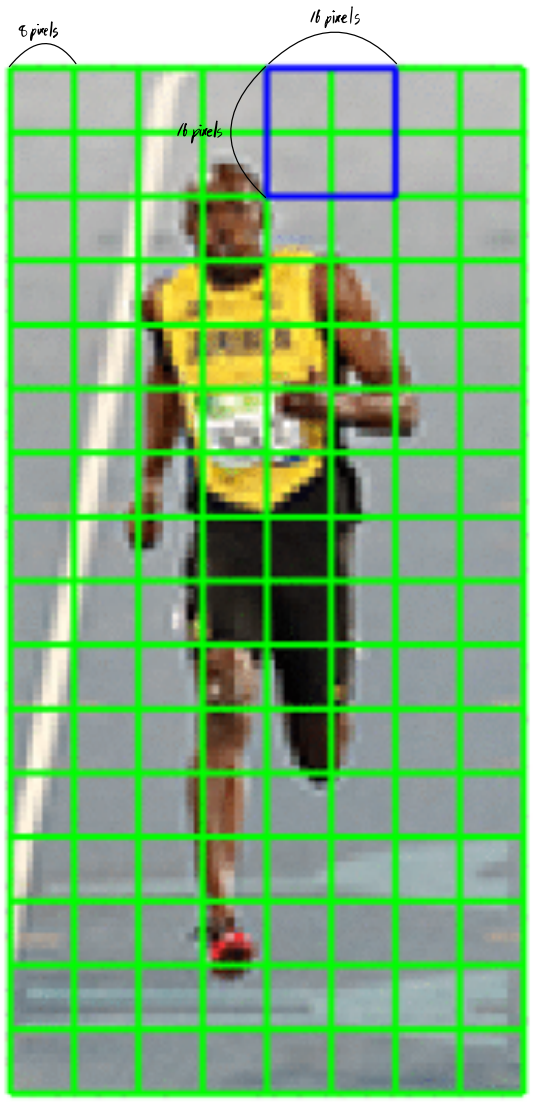
Gradient Magnitude

80	36	5	10	0	64	90	73
37	9	9	179	78	27	169	166
87	136	173	39	102	163	152	176
76	13	1	168	159	22	125	143
120	70	14	150	145	144	145	143
58	86	119	98	100	101	133	113
30	65	157	75	78	165	145	124
11	170	91	4	110	17	133	110

Gradient Direction

블록 히스토그램 구하기

8X8 셀 4개를 하나의 블록으로 지정합니다.



즉, 블록 하나의 크기는 16X16입니다.

8픽셀 단위로 이동합니다. (stride = 8) (블록 반칸씩 겹쳐서 이동)

각 블록의 히스토그램 빈(bin) 개수는 4X9 = 36개 입니다. (방향 성분 조합 36가지)

하나의 부분 영상 패치에서의 특징 백터 크기는 7 X 15 X 36 = 3780이 됩니다.

여기에 특징 벡터이므로 또 4를 곱하게 됩니다.

용량이 엄청나지만 지금도 많이 사용하는 방법입니다.

window

해당 윈도우의 feature 크기

window의 가로에 존재하는 block 개수

window의 세로에 존재하는 block 개수

한 개의 block 내 gradient 개수


Cells
Cell

winStride (optional)

The `winStride` parameter is a 2-tuple that dictates the "step size" in both the *x* and *y* location of the sliding window.

Both `winStride` and `scale` are ***extremely important parameters*** that need to be set properly. These parameter have ***tremendous implications*** on not only the *accuracy* of your detector, but also the *speed* in which your detector runs.

In the context of object detection, a sliding window is a rectangular region of fixed width and height that "slides" across an image, just like in the following figure:

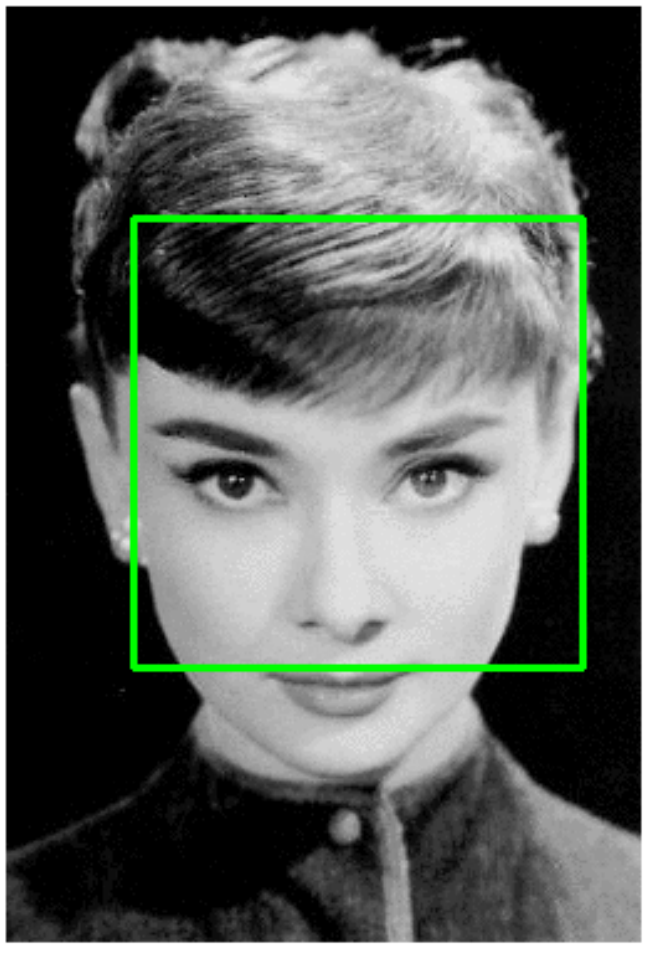


Figure 3: An example of applying a sliding window to an image for face detection.

At each stop of the sliding window (and for each level of the image pyramid, discussed in the `scale` section below), we (1) extract HOG features and (2) pass these features on to our Linear SVM for classification. The process of feature extraction and classifier decision is an expensive one, so we would prefer to evaluate *as little windows as possible* if our intention is to run our Python script in near real-time.

The *smaller* `winStride` is, the more windows need to be evaluated (which can quickly turn into quite the computational burden):

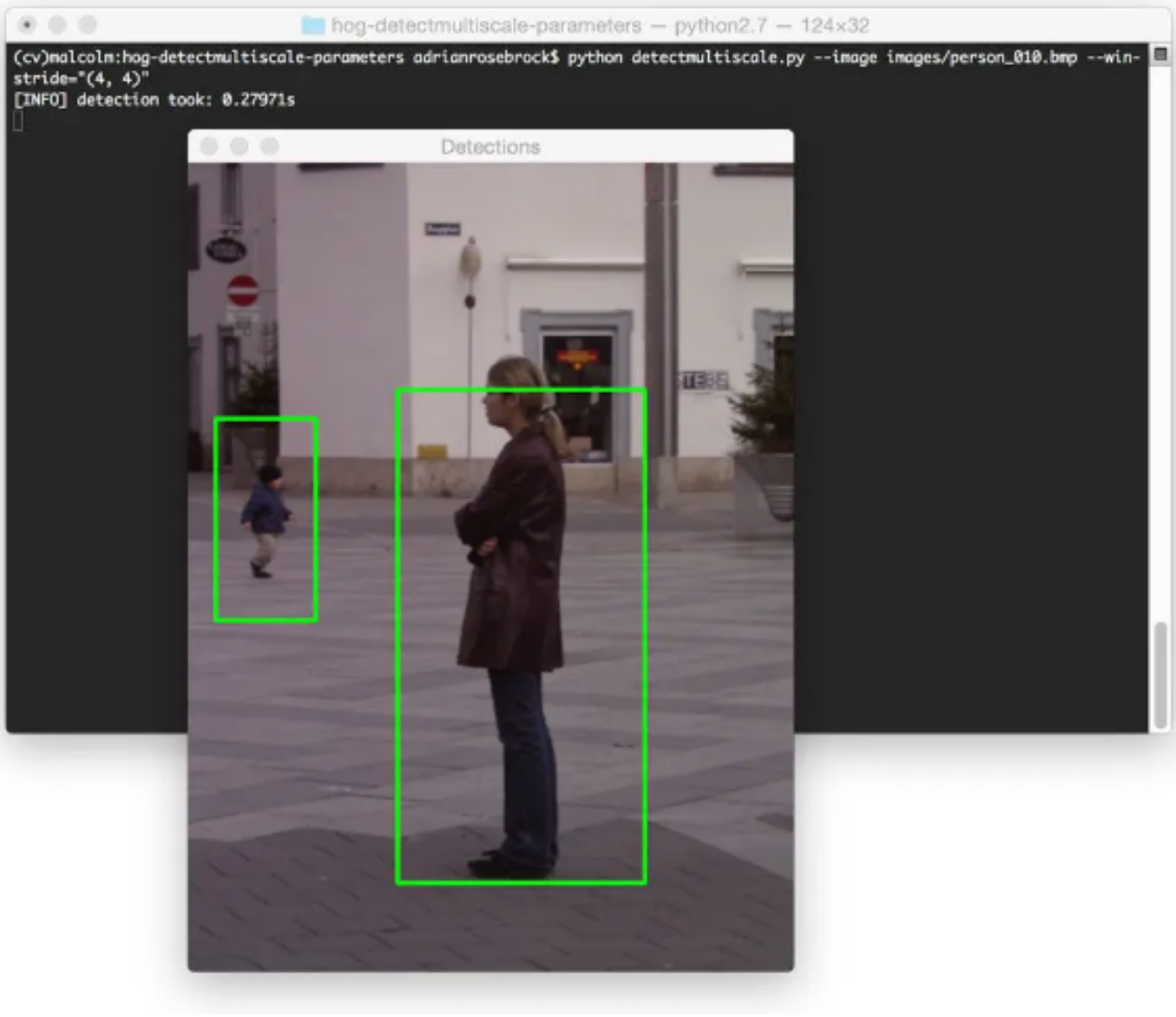


Figure 4: Decreasing the *winStride* increases the amount of time it takes it process each each.

Here we can see that decreasing the `winStride` to *(4, 4)* has actually increased our detection time substantially to 0.27s.

Similarly, the *larger* `winStride` is the *less windows* need to be evaluated (allowing us to dramatically speed up our detector). However, if `winStride` gets too large, then we can easily miss out on detections entirely:

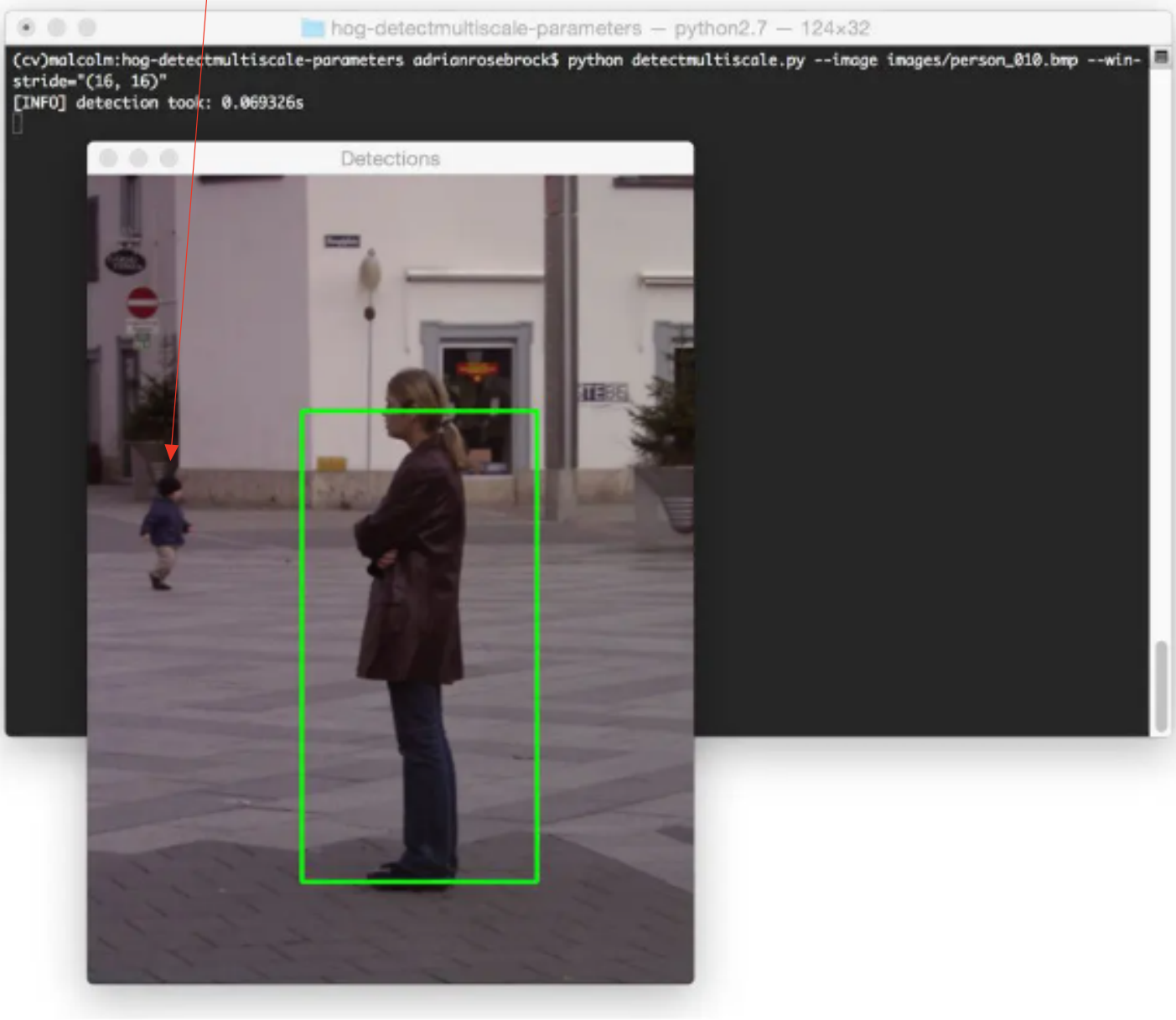


Figure 5: Increasing the *winStride* can reduce our pedestrian detection time (0.09s down to 0.06s, respectively), but as you can see, we miss out on detecting the boy in the background.

I tend to start off using a `winStride` value of *(4, 4)* and increase the value until I obtain a reasonable trade-off between speed and detection accuracy.

padding (optional)

The `padding` parameter is a tuple which indicates the number of pixels in both the *x* and *y* direction in which the sliding window ROI is "padded" (prior to HOG feature extraction).

Region Of Interest.

As suggested by Dalal and Triggs in their 2005 CVPR paper, ***Histogram of Oriented Gradients for Human Detection***, adding a bit of padding (surrounding the image ROI) (*prior* to HOG feature extraction and classification) can actually increase the accuracy of your detector.

Typical values for padding include *(8, 8)*, *(16, 16)*, *(24, 24)*, and *(32, 32)*.

scale (optional)

An image pyramid is a *multi-scale representation* of an image:

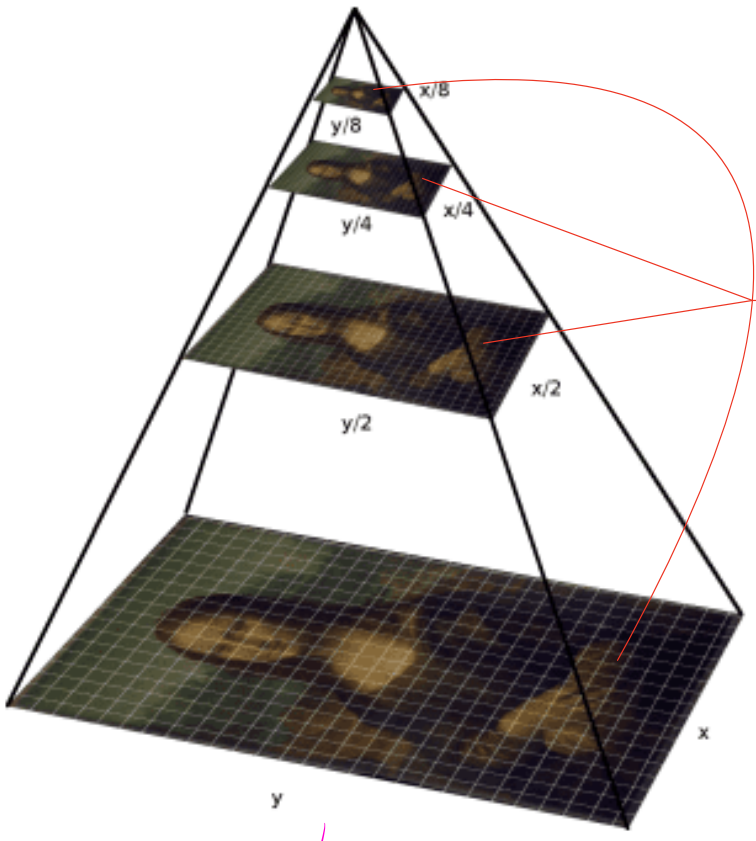


Figure 6: An example image pyramid.

At each layer of the image pyramid the image is downsized and (optionally) smoothed via a Gaussian filter.

This `scale` parameter controls the factor in which our image is resized at each layer of the image pyramid, ultimately influencing the *number* of levels in the image pyramid.

A *smaller* `scale` will ***increase*** the number of layers in the image pyramid *and* ***increase*** the amount of time it takes to process your image.

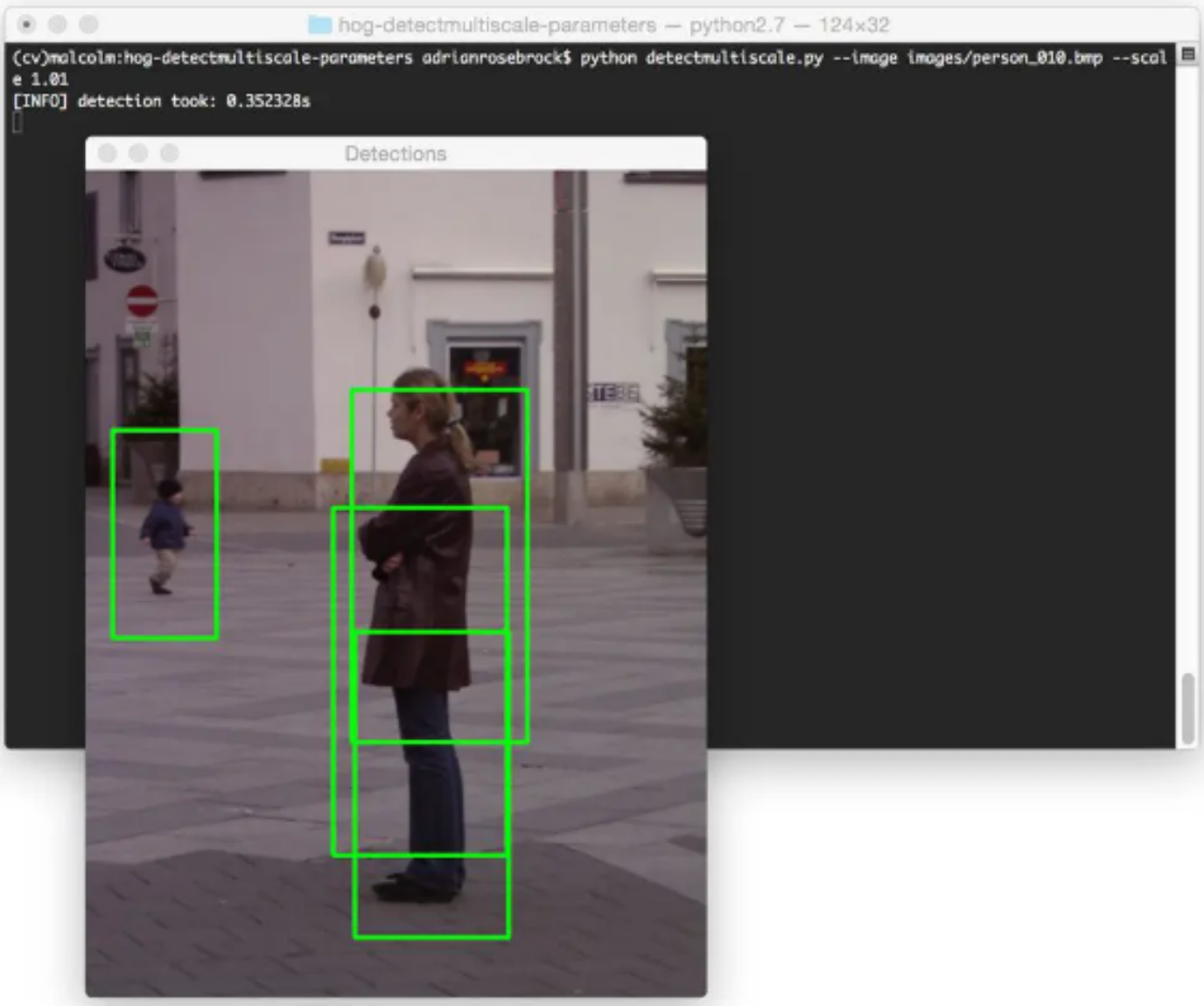


Figure 7: Decreasing the *scale* to 1.01

The amount of time it takes to process our image has significantly jumped to 0.3s. We also now have an issue of overlapping bounding boxes. However, that issue can be easily remedied using **non-maxima suppression**.

maximum의 복수 억제, 진압

Meanwhile a *larger* scale will ***decrease*** the number of layers in the pyramid as well as ***decrease*** the amount of time it takes to detect objects in an image:

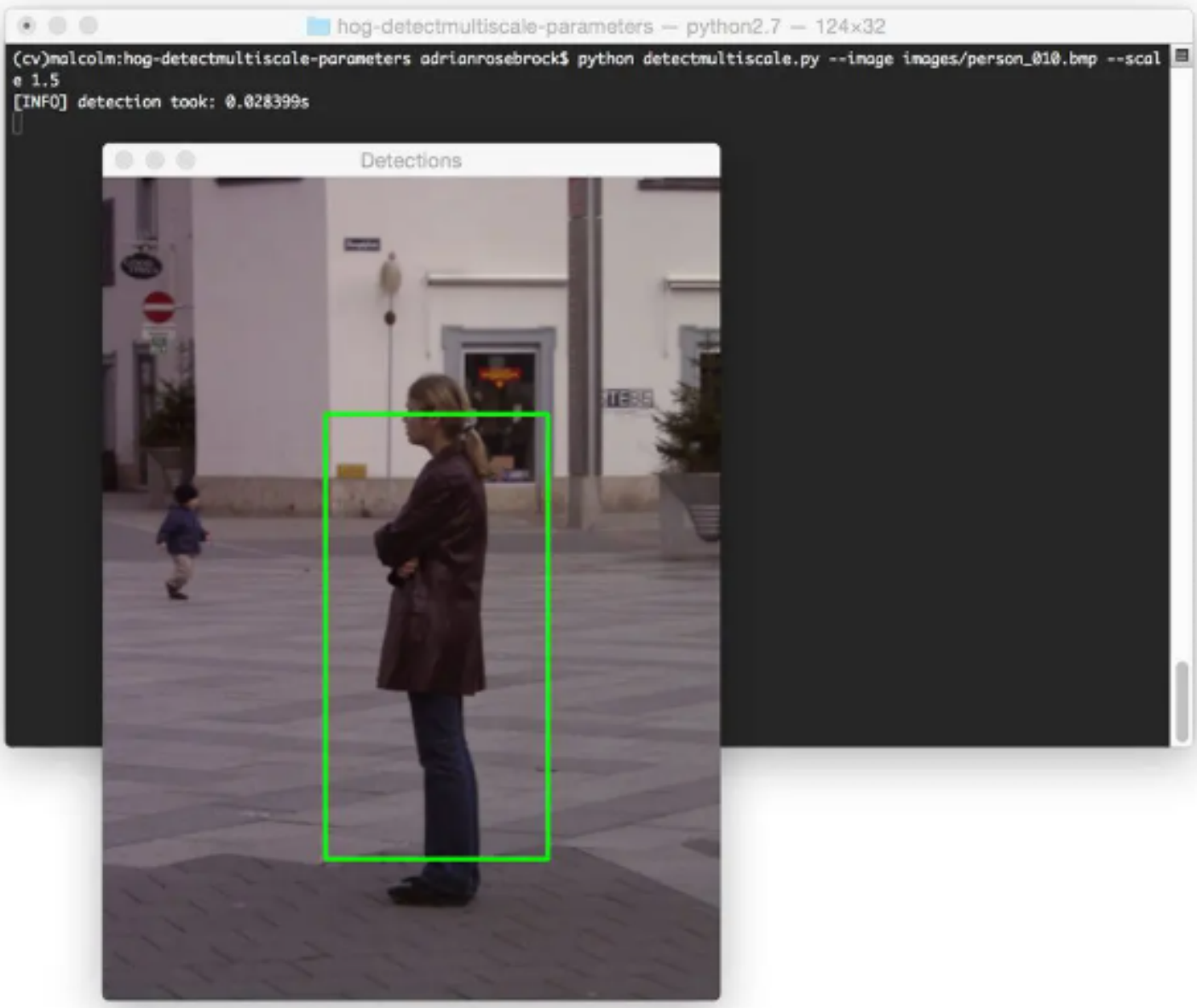


Figure 8: Increasing our *scale* allows us to process nearly 20 images per second — at the expense of missing some detections.

Here we can see that we performed pedestrian detection in only 0.02s, implying that we can process nearly *50 images per second*. However, this comes at the expense of missing some detections, as evidenced by the figure above.

Finally, if you decrease *both* `winStride` and `scale` at the same time, you'll *dramatically* increase the amount of time it takes to perform object detection:

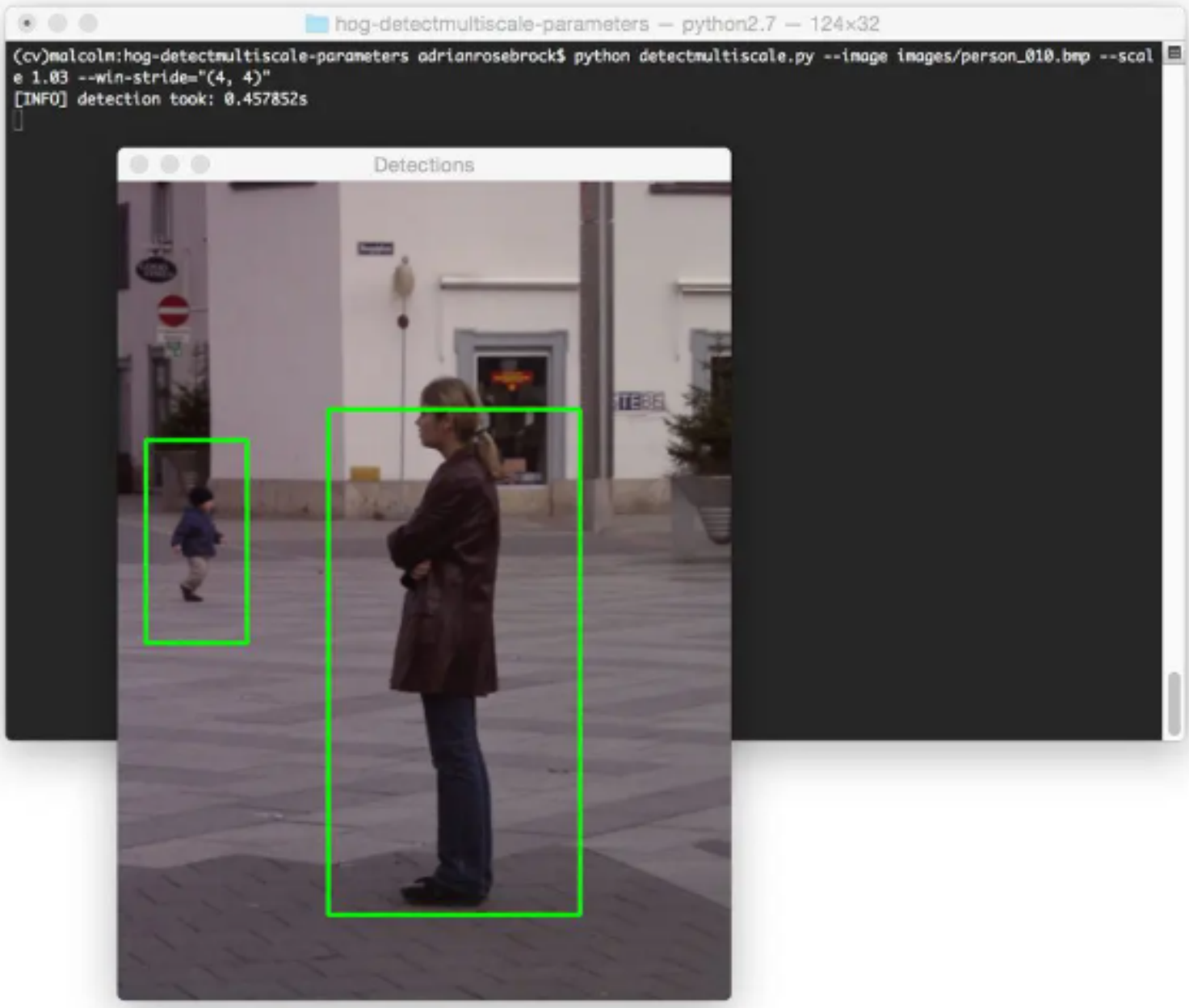


Figure 9: Decreasing both the scale and window stride.

We are able to detect both people in the image — but it's taken almost half a second to perform this detection, which is absolutely not suitable for real-time applications.

Keep in mind that for *each layer of the pyramid* a sliding window with `winStride` steps is moved across the *entire layer*. While it's important to evaluate multiple layers of the image pyramid, allowing us to **find objects in our image at different scales**, it also adds a significant computational burden since each layer also implies a series of sliding windows, HOG feature extractions, and decisions by our SVM must be performed.

Typical values for `scale` are normally in the range *[1.01, 1.5]*. If you intend on running `detectMultiScale` in real-time, this value should be as large as possible without significantly sacrificing detection accuracy.

Again, along with the `winStride` , the `scale` is the most important parameter for you to tune in terms of detection speed.

hitThreshold (optional)

The `hitThreshold` parameter is *optional* and is not used by default in the `detectMultiScale` function.

When I looked at the **OpenCV documentation** for this function and the only description for the parameter is: *"Threshold for the distance between features and SVM classifying plane"*.

Given the sparse documentation of the parameter (and the strange behavior of it when I was playing around with it for pedestrian detection), I believe that this parameter controls the maximum Euclidean distance between the input HOG features and the classifying plane of the SVM. If the Euclidean distance *exceeds* this threshold, the detection is rejected. However, if the distance is *below* this threshold, the detection is accepted.

My personal opinion is that you shouldn't bother playing around this parameter *unless* you are seeing an extremely high rate of false-positive detections in your image. In that case, it might be worth trying to set this parameter. Otherwise, just let **non-maxima suppression** take care of any overlapping bounding boxes, **as we did in the previous lesson**.