We already have an opencv to handle a lot of the annoying implementation of HOG (Histogram of Oriented Gradients), however we need to understand how everything behind it is working.

For a summary of all the steps, look here really quickly:
http://www.pyimagesearch.com/2014/11/10/histogram-oriented-gradients-object-detection/

Since it also has good visuals.

However, i'm going to write it here anyways in an abridged version to ensure I understand it.

1) Given a bunch of images with our class we're detecting (**positive** samples), and a bunch of images without the class we're detecting (**negative** samples), we have our dataset.

2) For each sample, we get the HOG descriptors.

Getting HOG Descriptors

This really is its own multi-step process.

1) Get the image gradients, horizontal and vertical (x and y direction derivatives). If we think about an image as a 3d graph, with the pixel intensities being higher and lower z-axis values, it becomes a lot more obvious how we do this. When we look for areas of high slope in this manner, we are now noticing rapid changes in pixel intensity in the image, which gives us useful features such as edge detection.

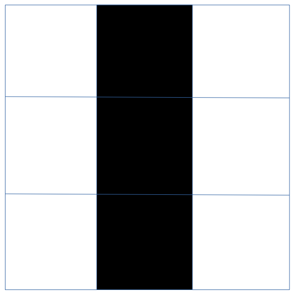
2) We compute these gradients via the following. Since we can think about our image as a matrix (2 dimensional) with z axis values as each entry, we can find the horizontal gradients by convoluting a horizontal kernel across the image, and vice versa for the vertical gradients.

$$Horizontal\ Gradient\ Kernel: \begin{bmatrix}-1, 0, 1\end{bmatrix}$$

$$Vertical\ Gradient\ Kernel: \begin{bmatrix}-1\\0\\1\end{bmatrix}$$

By doing this, for the horizontal gradient, we end up only seeing the vertical edges in our image - since if we have an example 3x3 grid like this (i.e. a small image):

*assume white = 255,
black = 0

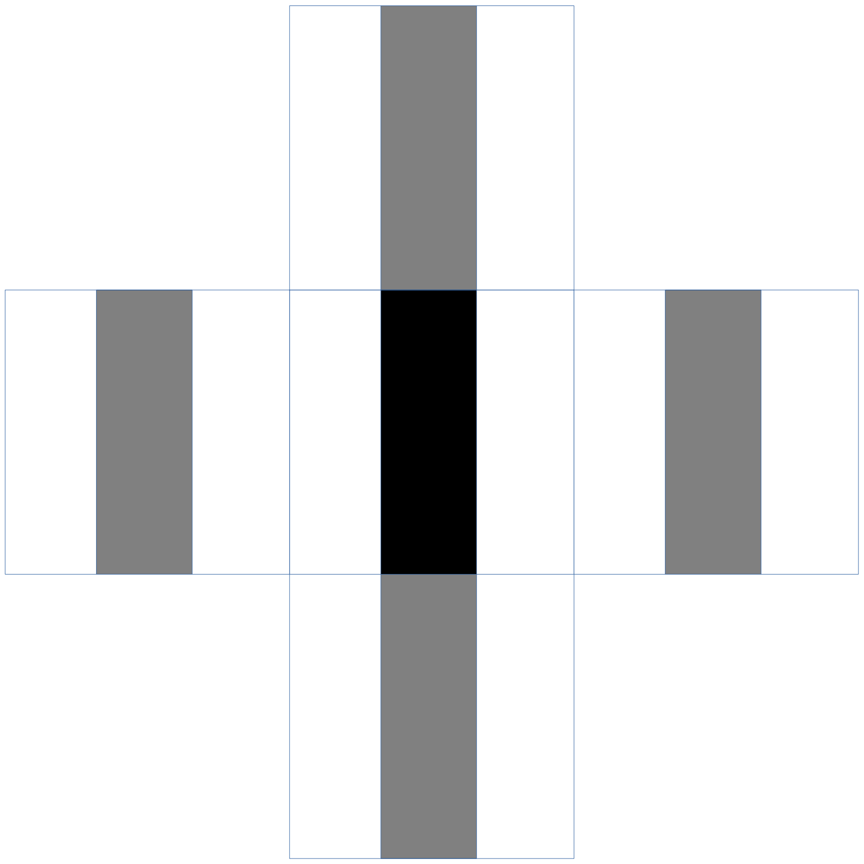


We'd apply our horizontal kernel, and get a large number for the top-left corner, since 1 * large number = large number. We'd then get 0 for the top-middle, since -1 * large number + 1 * large number = 0, and then we'd get a small number for the top-right. We'd repeat this across the rows of the image, since that is how you apply the convolution operator. In doing this, we'd end up with a matrix like this:

$$\begin{bmatrix} 255, 0, -255 \\ 255, 0, -255 \\ 255, 0, -255 \end{bmatrix}$$

As a note, we do this via first reflecting the elements of our kernel horizontally and vertically, then scanning across our image as we normally would do - padding with copies of the outermost image element where needed in order to preserve the original image dimensions. In this case, we'd pad on the left and right sides - in the latter case, we'd pad on top and bottom. As we scan across, we elementwise multiply our kernel by the image elements in that position, and sum them all, putting the result in the result matrix. The position of where to put this result can be determined by the number of strides in the rows, and columns - if we have gone 3 to the right, and 0 down, we'd put the result in index [0, 3]. So that's how to apply the convolutional operator, in a way that works for both matrices and rows.

This process varies depending on if you are padding in a symmetrical manner, if you want to pad so that the result is the same size as the original, and so on. These are usually changeable parameters in whatever library you use to apply the convolution. For an example, see the /documents/convolution_experimentation_script.py script. (This script uses mode="same", which makes the result be the same, but you could also use "full", which would apply the kernel and pad it without regard for the output dimension. It also uses "symm" which means we have ghost elements on the side of our input image like the following:



Where I have the ghost elements as the faded ones. There are of course other types of padding, like constant filling (using one value for all side things, like zero), or wrapping, where it goes around to the other side of the image.)

Anyways, from this we can obviously see our vertical edge made a difference in the resulting horizontal gradient..

However, if we did it for our vertical kernel, we'd end up with:

$$\begin{bmatrix} 0,0,0 \\ 0,0,0 \\ 0,0,0 \end{bmatrix}$$

Not so nice.

3) So we compute the gradients in that manner, and end up with x gradients (horizontal) and y gradients (vertical), which are of the same size as the original image. To reiterate, our x gradients show the vertical edges, and y gradients show horizontal edges. To continue our example, we'd then compute the magnitude and direction of these gradients:

$$For\ a\ horizontal\ gradient\ g_x\ and\ vertical\ gradient\ g_y :$$
$$g = \sqrt{g_x^2 + g_y^2}$$
$$\theta = \arctan g_y/g_x$$

4) By doing this, we now have a matrix for gradient magnitude, and a matrix for gradient direction, on each pixel of the input image. These replace our previous x and y matrices.

5) This is good, but we want to compress it to a more compact representation. Assuming unsigned gradients (sign is irrelevant) in degrees, each value in the direction matrix will be 0-180. This is where the histogram idea comes in. We then break up our image into 8x8 patches (though this number could be different depending on your implementation), in order to ensure that the values obtained in each 8x8 patch are robust to a considerable amount of noise, instead of just using all individual pixel gradient magnitudes and directions.

6) For each 8x8 patch, we have 9 bins: 0, 20, 40, 60, 80, 100, 120, 140, 160. We loop through our magnitudes and directions, and sort them in the way shown in these examples:

*The histogram is a hashmap in this example
ex1) mag = 2, direction = 20
0: 0
20: 2
40: 0
60: 0
80: 0
100: 0
120: 0
140: 0
160: 0

ex2(continued from ex1)) mag = 6, direction = 70
0: 0
20: 2
40: 0
60: 3
80: 3
100: 0
120: 0
140: 0
160: 0

ex3(continued)) mag=85, direction=165
0: 21.25
20: 2
40: 0
60: 3
80: 3
100: 0
120: 0
140: 0
160: 63.75

For a sample direction, we get the nearest bin values (180 = 0 if unsigned), and then find how far off it is from the first and the second. In the case of ex3, this is |165 - 160| = 5, |165-180| = 15. We then divide each of these by 20 (the gap) to get the percent difference, in this case 5/20 = .25, 15/20 = .75 . This percentage will be how much of our magnitude gets put in the bin, and since we want the bin it is closest to to have a larger percentage of the magnitude than the other, we invert these percentages: 1-.25 = .75, 1-.75 = .25. We then multiply our magnitude by these percentages / proportions, and increment the current value of each bin by the result: 85 * .25 = 21.25, 85 * .75 = 63.75.

7) We repeat this process for every pixel in the patch to get 9 values to represent the patch, and repeat this across all patches. By doing this we end up with 9 values for each 8x8 patch in an image - a very compressed representation.

8) While this is almost good enough, we can do better. We want to be invariant to lighting changes, and since lighting changes are just scalar multiples of our image matrix, we can be invariant by normalizing 2x2 of our 8x8 patches, of size 16x16 in total (we want to make sure we allow actual changes in the image, since it is likely to have some differences throughout and we don't want to lose those - this is why we do normalization on 16x16 instead of the whole image). Since each of our 8x8 blocks has 9 values, we have 9*4 = 36 values in our 16x16 normalization block. We then compute the l2 norm by getting the vector length ($\sqrt{x^2 + y^2 + z^2}$) of this 36-element vector, and divide each element in the vector by the resulting length.

*For an idea of why this helps us be invariant, we can easily look at two examples: one on an (x, y, z) vector and one on a c * (x, y, z) vector where c is a scalar. If we repeat our process, we prove that the result of applying this process on both of these vectors is equivalent.*

9) By striding across our 8x8 patches, repeating this normalization on 2x2 (4) patches at a time, we end up with a new, even more compact representation obtained from the number of times we can scan across with this 16x16 "meta-kernel" (word I made up) size * 36.

10). Afterwards, we just flatten our result - for an example image of size 64x128, we would have a result of 7x15 of these 16x16 blocks, and since each block has 36 values (because 9 * 4 = 36), we end up with 7 * 15 * 36 = 3780 length vector as our end result.

This is our HOG descriptor!

3) But as I was saying. We get HOG descriptors for each sample, and these become our new samples. We then train an SVM on these new samples and I swear to god I am not writing about SVMs right now you can view SO MANY TUTORIALS ON THOSE XD.

4) Once we've done so, we apply **hard-negative mining**, a process where we get all the false-positive samples our SVM predicts (in test data), and train the SVM again. This is useful because often times the SVMs generate tons of false positives at first, and we can improve on this by taking the false positives, creating new "hard-negative" samples from them, and training again.

5) At this point, our model is ready so we move on to actual implementation on data:

6) Given an image, we slide a window across the image from the top left to bottom right, padding is optional, stride is a parameter we can change. We repeat this across the entire image, getting new windows for our next step. Once we've gone across the image, we scale the image size down by a resize factor (another parameter), while keeping the window size the same. Once we've done that, we then slide across and repeat the process, until we've reached an image size smaller than our window size.

7) Given a window, we feed it into our model and draw a box on that area in the image if it is classified as positive. We do this for every window.

8) Once done with the image, we may have multiple overlapping bounding boxes for the same detected object. We can fix this using either the mean-shift algorithm, or another non-maxima suppression algorithm. The mean shift algorithm can be found here, but it's an iterative one where we start with an estimate of the new value (to be obtained from multiple other values, in this case coordinates for either our top-left or bottom-right coordinate) "x", and then iterate over the following:

$$m(x) = \frac{\sum_{x_i \in N(x)} K(x_i - x) x_i}{\sum_{x_i \in N(x)} K(x_i - x)}$$

Where we repeatedly do this, setting x = m(x) after each evaluation until convergence.

Oh also this is K:

$$K(x_i - x) = e^{-c||x_i - x||^2}$$

So, yea. We can do most of this using OpenCV's detectMultiScale, if we also train a HOG and SVM using OpenCV, I think. Sources for this down below.

Afterwards that's it.

Sources

Mean Shift
- https://en.wikipedia.org/wiki/Mean_shift

HOG
- http://www.pyimagesearch.com/2014/11/10/histogram-oriented-gradients-object-detection/
- http://answers.opencv.org/question/4351/training-gpu-hogdescriptor-for-multi-scale-detection/
- http://stackoverflow.com/questions/28390614/opencv-hogdescripter-python
- http://stackoverflow.com/questions/6090399/get-hog-image-features-from-opencv-python

detectMultiScale
- http://www.pyimagesearch.com/2015/03/23/sliding-windows-for-object-detection-with-python-and-opencv/
- http://www.pyimagesearch.com/2015/11/16/hog-detectmultiscale-parameters-explained/

More on SVMs in different documents.