Advanced Vehicle Detection & Tracking

By David Rose

6/22/17

**Vehicle Detection and Tracking**

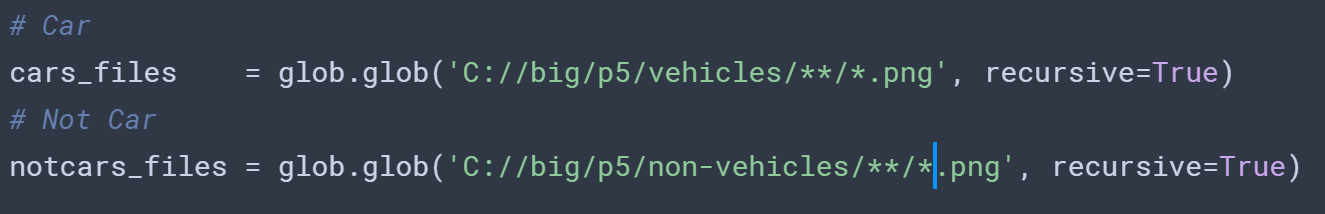
**Following topics include:**

* Extracting Features
  + Histogram Oriented Gradients (HOG)
  + Converting color space
  + Choose from various parameters to perform ideal detection of vehicles
* Scaling within features and among all features using normalization
* Implementing a Sliding Window approach of analyzing the image using multiple small crops across the image, at various positions and sizes
* Training a classifier, in this case a Support Vector Machine (SVM), on a pre-labeled dataset and then use it to detect on new images and video
* Combining bounding boxes (detected car sections) and adding them up to create heat maps.

**Steps**

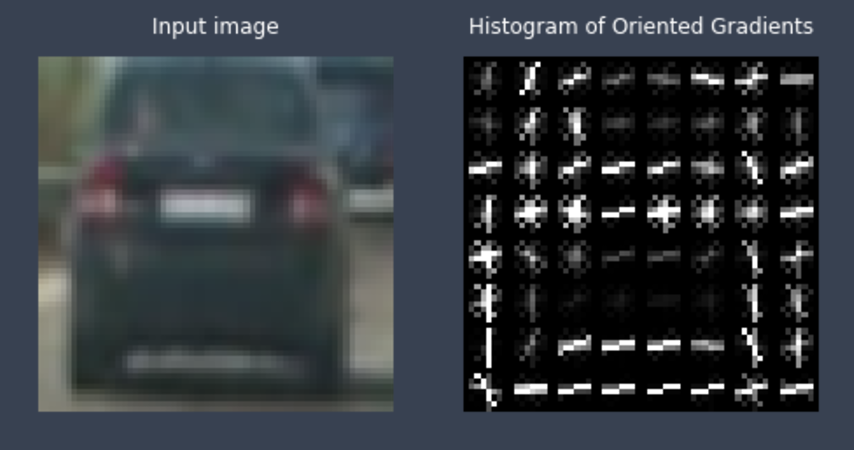
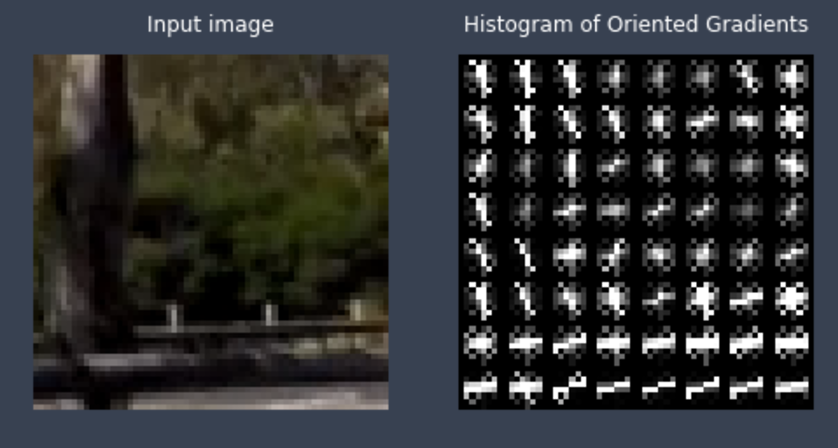
## Read in Training Data

* Using the Glob module, I searched all subfolders from with a cars and not-cars set, appending their locations to a list for both cars and not-cars. Using the newly added argument for recursive=True enables the searching of any/all subfolders for the images.



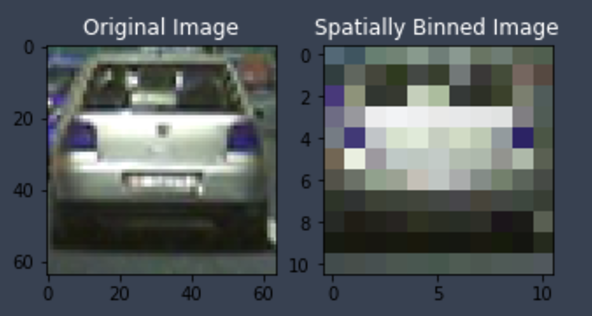
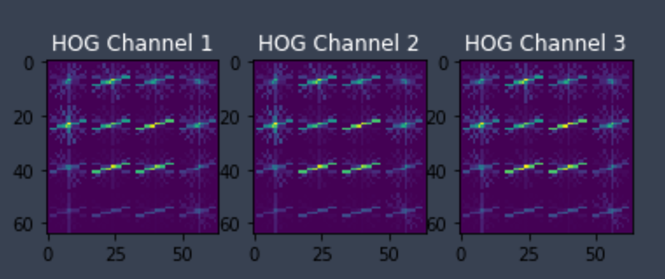
## Histogram of Oriented Gradients (HOG)

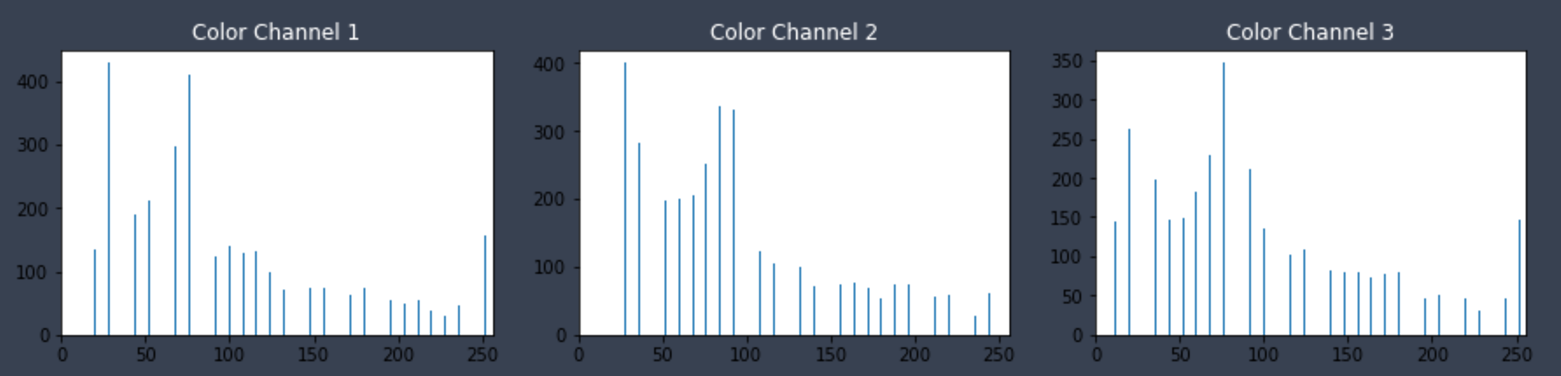
* Using the HOG algorithm from the SKImage module, I was able to convert the standard images of cars/not-cars to sections of pixel orientations, in effect compressing the image and extracting certain features that may be representative in pictures of cars.
* There are a few parameters to tweak, such as converting a specific color space, the amount of possible orientations, the pixels per cell, and the amount of cells per block. I ended up with using the following:
  + Orient = 9
  + Pix\_per\_cell = 8
  + Cell\_per\_block = 8
  + Hog\_channel = 0



## Other Features

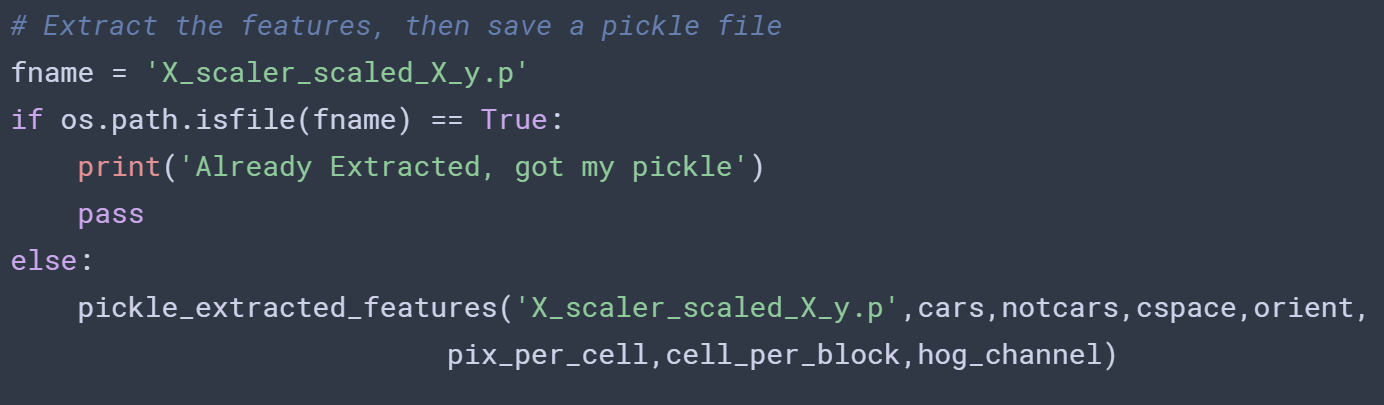
* I also performed a **spatial binning**, taking three different color channels from the last dimension of the images and in essence **re-sizing the overall dimensions**, along with taking a **histogram** of the color dimensions.
* These other two features were then placed along a single 1D array, and concatenated together.
* **Normalization:** Each individual feature was scaled through normalization, then once combined were all scaled as well. These in effect became the training dataset, to help the classifier learn which images are cars and which are not-cars.





## Saving Extractions to Pickle

* To prevent having to perform this over-and-over I saved all extracted features as a binary pickle file.
* The script will check to see if the file exists first, and loads the file if it does. If not, it will run the feature extraction.



## Training the Classifier (SVM)

* Once I have all the image data ready I train an SVM classifier on the data, learning the features of cars and not-cars based on the features that were extracted earlier (spatial binning, color histogram, HOG)
* The code is fairly straightforward, and performs well out of the box, pulling in around a ~96-98% accuracy from the training set.
  + I use a 70/30% training/test split on the data

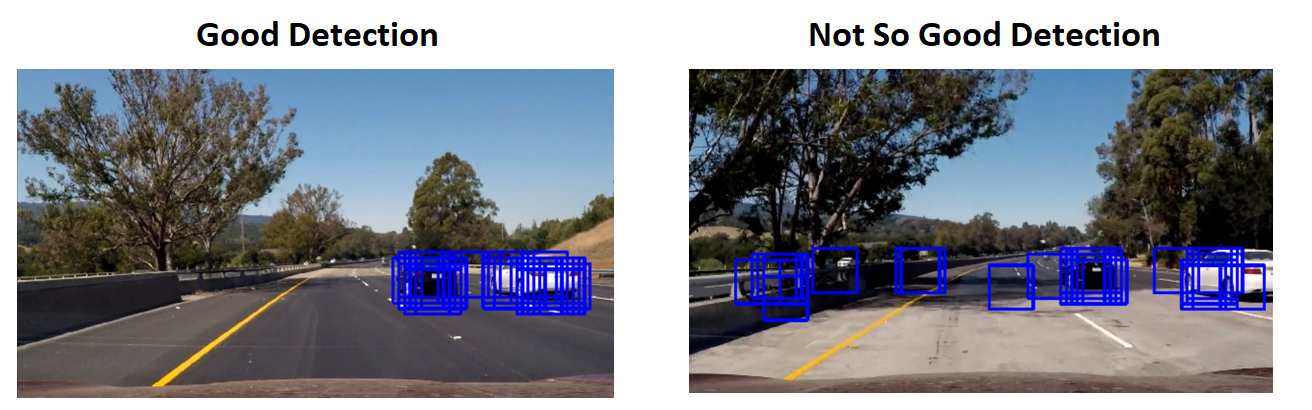
## Find the Cars!

### Sliding Windows

* 1. Taking multiple windows (crops) of the image, the function goes one-by-one and performs the same feature extraction as performed earlier in the training section.
  2. In this case I can specify a specific upper/lower bound for the y-axis to analyze as I don’t (normally) assume there a possibility of a car being in the sky, or on top of my car hood.
     + Here I use **400,700** as the pixel y-limits.
  3. Once the images are have been brought in, the windows begin sliding around, performs the extractions and is then ready to start predicting the windows that may or may not contain a car by using the SVM trained model that was set up earlier.
  4. The code here gets pretty long and complex but basically ends with **svc.predict(features)** that returns a yes or no regarding a detection of a car.

### If a car is detected?

* 1. When a car is detected, it will draw a box using OpenCV around the perimeter of that window.
  2. At the end of the image detection window sequence I will (hopefully) have a set of boxes surrounding any car in the overall image.



### False Positives?

* 1. Given that an actual car will usually have a large amount of windows on top, and various false positives usually just have one or two errant windows, I can use a stacking effect to let windows build upon themselves, effectively throwing away a lot of the bad data
  2. To be more specific, the function is fairly straightforward:
     + Add 1 to pixel value for each pixel contained in a box that detected a car.
     + When multiple windows overlap, the pixels will begin to add up and get brighter.
     + 



## Time For Some Video

* It is surprisingly simple to switch from detection in a single frame to a video clip. The video is broken down in to individual frames and just processes as before. Depending on the size and complexity of your pipeline it can take a while to run. With my setup it currently processes in around ~100 minutes. I have a brand new 8-core AMD processor, but it seems this only runs single-threaded which is a huge bottleneck. Theoretically I could run this in 10 minutes or less if allowed to use the whole processor.

### Steps:

* 1. Begin the window search on first frame.
  2. If there are one or more windows detected

####2. Show some examples of test images to demonstrate how your pipeline is working. What did you do to optimize the performance of your classifier?

Ultimately I searched on two scales using YCrCb 3-channel HOG features plus spatially binned color and histograms of color in the feature vector, which provided a nice result. Here are some example images:

**Video Implementation**

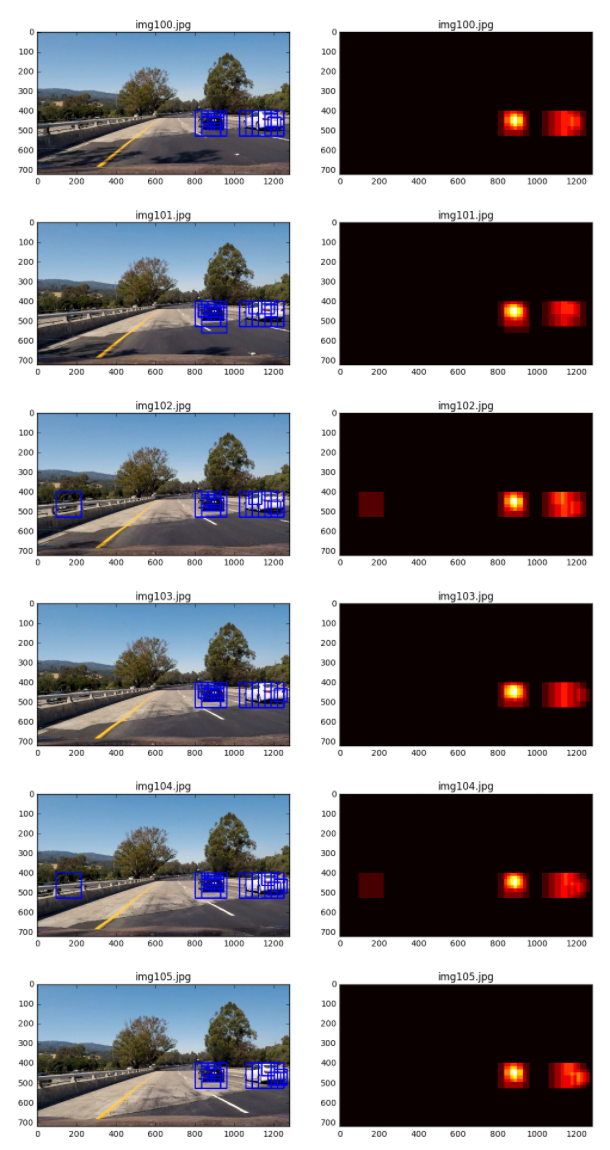
####1. Provide a link to your final video output. Your pipeline should perform reasonably well on the entire project video (somewhat wobbly or unstable bounding boxes are ok as long as you are identifying the vehicles most of the time with minimal false positives.) Here's a [link to my video result](https://github.com/udacity/CarND-Vehicle-Detection/blob/master/project_video.mp4)

####2. Describe how (and identify where in your code) you implemented some kind of filter for false positives and some method for combining overlapping bounding boxes.

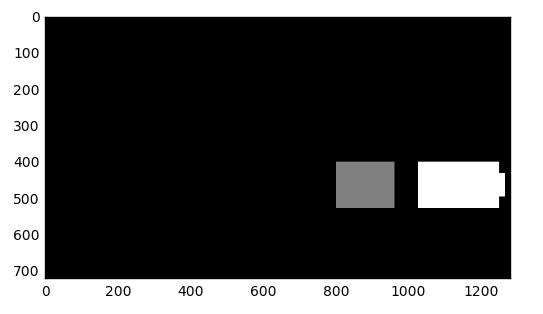
I recorded the positions of positive detections in each frame of the video. From the positive detections I created a heatmap and then thresholded that map to identify vehicle positions. I then used scipy.ndimage.measurements.label() to identify individual blobs in the heatmap. I then assumed each blob corresponded to a vehicle. I constructed bounding boxes to cover the area of each blob detected.

Here's an example result showing the heatmap from a series of frames of video, the result of scipy.ndimage.measurements.label() and the bounding boxes then overlaid on the last frame of video:

**Here are six frames and their corresponding heatmaps:**

[](https://github.com/udacity/CarND-Vehicle-Detection/blob/master/examples/bboxes_and_heat.png)

**Here is the output of scipy.ndimage.measurements.label() on the integrated heatmap from all six frames:**

[](https://github.com/udacity/CarND-Vehicle-Detection/blob/master/examples/labels_map.png)

**Here the resulting bounding boxes are drawn onto the last frame in the series:**

[](https://github.com/udacity/CarND-Vehicle-Detection/blob/master/examples/output_bboxes.png)

###Discussion

####1. Briefly discuss any problems / issues you faced in your implementation of this project. Where will your pipeline likely fail? What could you do to make it more robust?

Here I'll talk about the approach I took, what techniques I used, what worked and why, where the pipeline might fail and how I might improve it if I were going to pursue this project further.