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

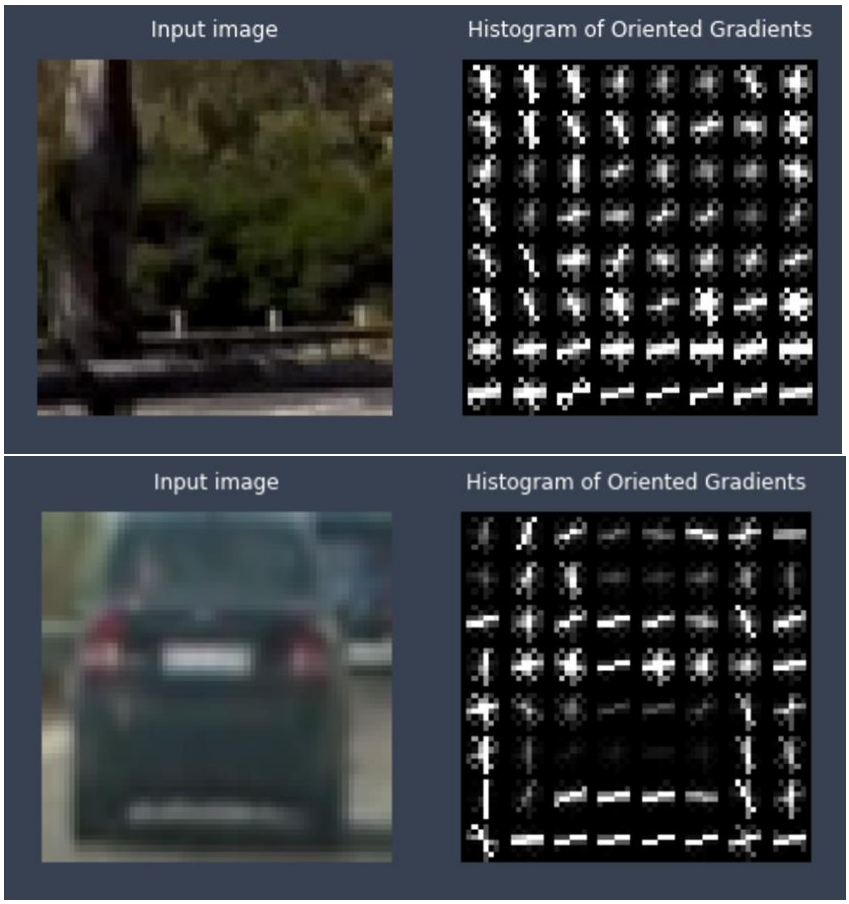
Read in Training Data

- 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.

```
# Car
cars_files    = glob.glob('C://big/p5/vehicles/**/*.png', recursive=True)
# Not Car
notcars_files = glob.glob('C://big/p5/non-vehicles/**/*.png', recursive=True)
```

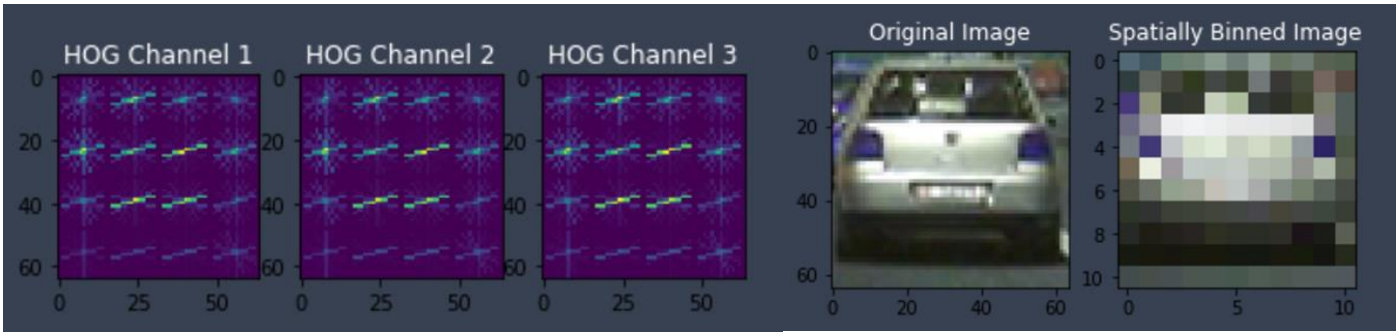
Histogram of Oriented Gradients (HOG)

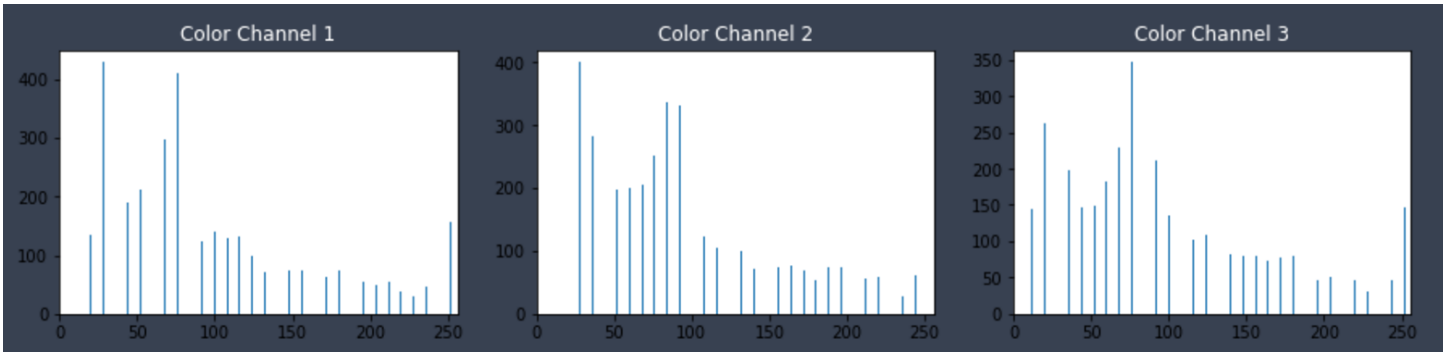
- *The code can be found in code block 4 in main.ipynb, and lines 423-441 in helperfunctions.py*
- Using the HOG algorithm from the SKImage module, I was able to convert the standard images of cars/not-cars to sections of pixel orientations, in effect compressing the image and extracting certain features 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:
    - Color space = 'YCrBc'
    - Orient = 11
    - Pix_per_cell = 16
    - Cell_per_block = 2
    - Hog_channel = ALL
- Initially I had trouble using lower values, I consistently got some false positives, but I eventually moved up orient by 2, and doubled pix_per_cell. Adding all channels was also changed in the next revision, though it did add some more time to processing speed. The color space took a bit of playing around with, initially I used HLS as in the lane finding project, but it got way too many false positives. Some on the forums mentioned to try YCrBc and that's what I have found best since.



Other Features

- *Helperfunctions.py lines 236-252*
- 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.
- **<u>Normalization</u>:** 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

- *helperfunctions.py lines 445-481*
- 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.

```
# Extract the features, then save a pickle file
fname = 'X_scaler_scaled_X_y.p'
if os.path.isfile(fname) == True:
    print('Already Extracted, got my pickle')
    pass
else:
    pickle_extracted_features('X_scaler_scaled_X_y.p',cars,notcars,cspace,orient,
                              pix_per_cell,cell_per_block,hog_channel)
```

Training the Classifier (SVM)

- *In main.ipynb code block 4*
- 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

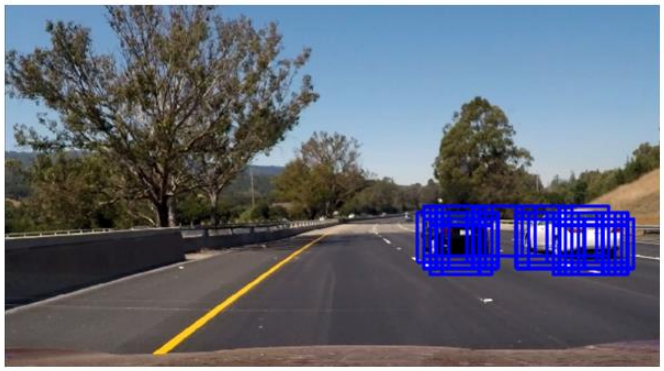
Find the Cars!

- **Sliding Windows**
    - *In main.ipynb code block 5 and helperfunctions.py in lines 105-155*
    - 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.
    - 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.
    - 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.
    - 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.

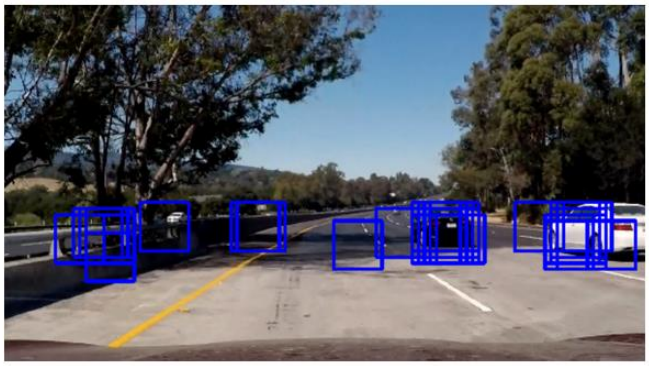
y_start	y_stop	scale
400	464	1.0
416	480	1.0
400	496	1.5
432	528	1.5
400	528	2.0
432	560	2.0
400	596	3.5
464	660	3.5

- **If a car is detected?**
  - *In helperfunctions.py lines 159-173*
  - When a car is detected, it will draw a box using OpenCV around the perimeter of that window.
  - At the end of the image detection window sequence I will (hopefully) have a set of boxes surrounding any car in the overall image.

Good Detection



Not So Good Detection

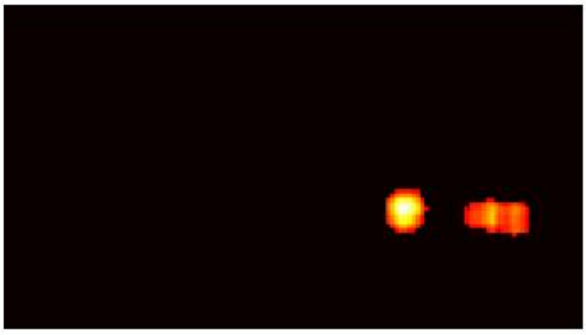


- **False Positives? (use heatmaps!)**
  - *Code can be found in process_frame in code block 9 in main,ipynb and lines 176-190*
  - Given that an actual car will usually have a large amount of windows on top, and various false positives usually just have one or two errant windows, I can use a stacking effect to let windows build upon themselves, effectively throwing away a lot of the bad data.
  - 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.
    - ```
      heatmap[win[0][1]:win[1][1],win[0][0]:win[1][0]] += 1
      ```

- **Bounding Boxes**
  - *In main,ipynb code block 9 and helperfunctions.py in lines 602-617*
  - Next method is to add up those windows/heatmaps and draw a singular rectangle around them, signifying a vehicle.

Final Boxes

Heatmaps


      ```

Time For Some Video

- *In main.ipynb code block 11 and helperfunctions.ipynb lines 669-728*
- 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:**
    - Begin the window search on first frame.
    - If there are one or more windows detected, add a pixel for heat
    - Average out the last 15 frames
    - Use the draw_labeled_windows function from earlier to translate the heat map back to boxes
    - Return the image overlayed with boxes for the cars

Discussion

Issues:

- **How to tune all the parameters?**
    - With the multitude of functions used and the various methods of feature extraction, there ends up being a LOT of various parameters that I can adjust. And given that the model takes a bit of time to run on video it is impracticable to try out every single one.
    - I started out with the provided values from the lessons and moved a few around based on recommendations from the forums, but I know it is not perfect.
    - Ideally I could get a function that passes through various values over time, and continually puts out single frames from the video. I could compare those and see what works best.
- **Why are we not using deep learning?**
    - Seems like an obvious solution to some of this, as object recognition is best performed by a model that can learn more complex and non-linear relationships between features.
    - Google has recently begun releasing pre-trained weights for models in general object recognition within their TensorFlow API, it could be possible to integrate this by removing the last layer and re-training on the vehicle dataset to better focus on vehicles.
- **Need to go faster!**
    - It takes my high-end computer over an hour for just the provided test image. For use in a car we will need to process at least 10 frames a second, and likely more. So there will need to be a LARGE increase in processing speed, likely done by trying a completely different approach than this one.