Craig Vargas

CarND Project 5: Vehicle Detection

Feature Vector

(car_cv.py lines: 227-313)

In order to train the image classifier in this project you have to extract features from the image that the classifier can use to learn the different classes. The features used in this project were: histogram of gradients, spatial binning, and pixel binning.

Histogram of Gradients

(car_cv.py lines: 114-124)

Histogram of gradients is a procedure that computes the gradient for each pixel in an image, breaks up the image into smaller tiles, computes a histogram of the gradient information within each tile, then normalizes the information with info from neighboring tiles. This information effectively becomes a signature for the image. The parameters needed for this procedure are: tile sizes, number tiles for normalization, and number of bins in each histogram. Our training images in this project were 64x64 in size. For these images I ending up selecting parameters: tile_size = 8x8, num_tiles_norm = 2, num_bins = 9. There is one other parameter used in this procedure which is to decide which image channels to use. I used HOG information from all three image channels because it gave me a significant increase in accuracy versus any one channel alone. I used skimage's "hog" function to perform the HOG feature extractions.

Spatial Binning

(car_cv.py lines: 49-53)

Spatial binning is simply the process of using the raw image pixels as a feature. The parameter to choose at this stage is the size of the image to use. By downsizing the image, you can reduce the size of the feature set and still retain valuable information of spatial pixels. The original images were 64x64 and during the trial and error phase of parameter tuning it didn't seem that the results differed much between images of size 64x64, 32x32, or 16x16. I used 16x16 as my spatial feature set in an effort to reduce the size of the feature vector. I used openCV's "resize" function to perform the spatial binning.

Pixel Binning

(car_cv.py lines: 198-205)

Pixel binning is the process of grouping the pixel intensities into bins and using that information as a signature for the image. The parameter to choose at this stage is the number of buckets to bin the pixel intensities into. I chose to group the pixel intensities into 16 buckets because the results were comparable to bucket sizes of 32 and 64 so I chose the smallest value to reduce the size of the feature vector. This binning was done for each image channel separately and the information was concatenated to make one signature to use as a part of the feature vector. I used numpy's "histogram" function to perform the pixel binning.

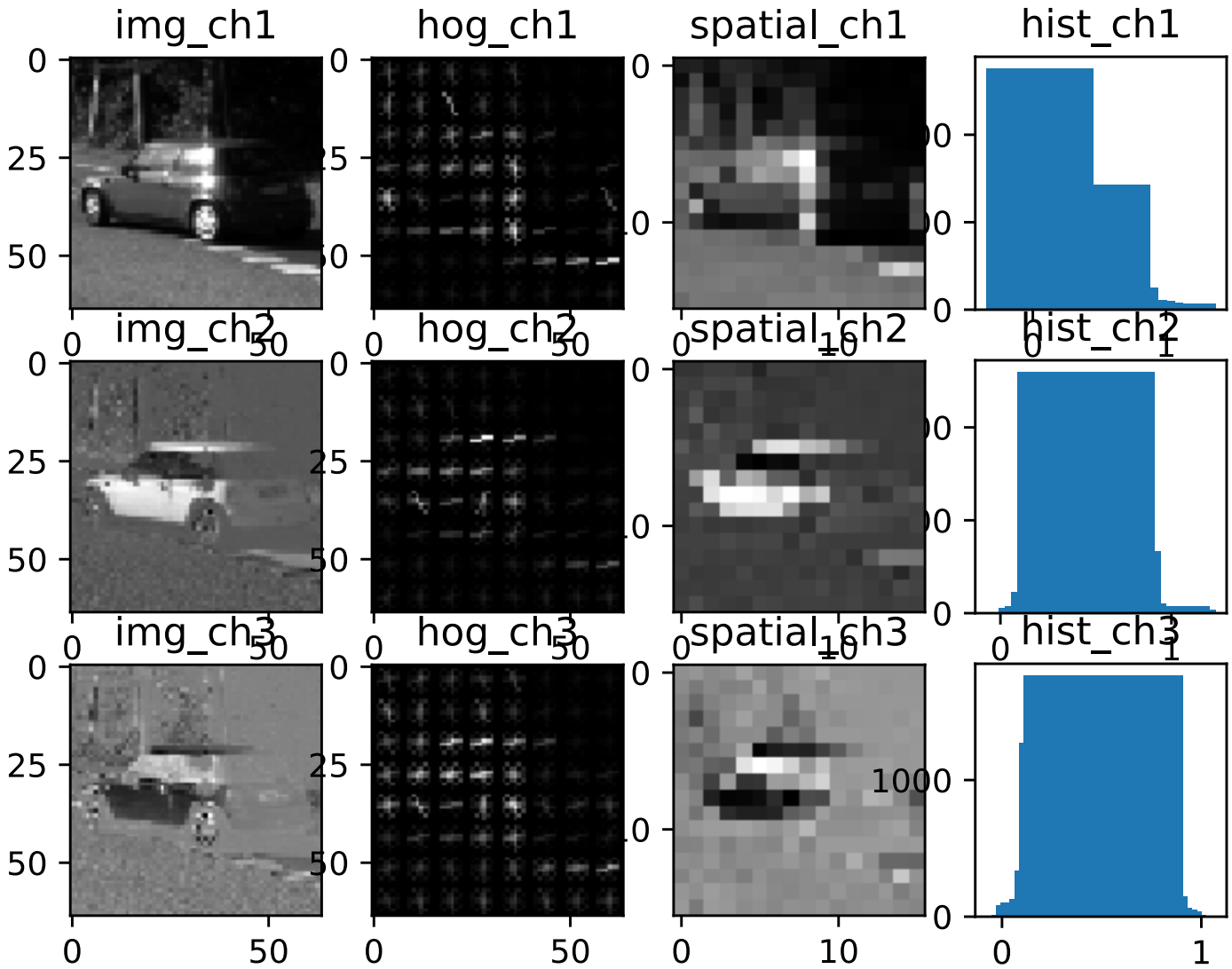
Parameter Selection

(find_cars.py lines: 37-117)

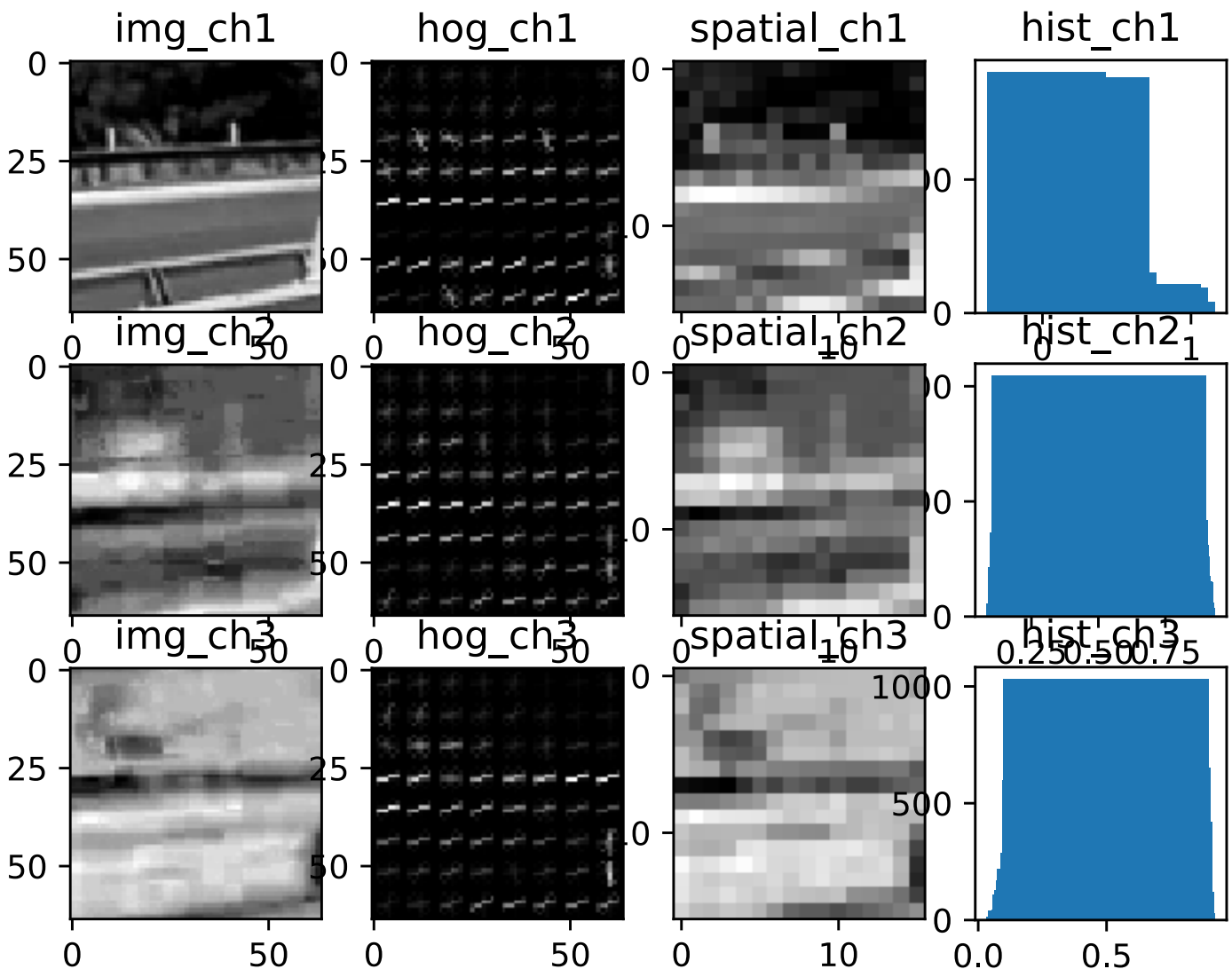
These parameters were selected via a process of trial and error where at each trial I used a small subset of the data (1000 class balanced images), trained on 800 images, and tested the classifier on 200 images. This process was repeated 10 times for each trial and the test accuracies were averaged to compare each trial.

One parameter that had to be decided on that was not covered above was the color space parameter. All of the features listed above work on pixel information in an image. That data can be represented in different color spaces and the signatures derived from each image will be different based on the color space used. Via the same trial and error process explained above, I chose to use the YCrCb color space because it gave slightly better results than some of the other spaces like HLS and LUV for example. A lot of the color spaces gave good results with exception given to the RGB color space.

Example of Car Features:



Example of Non Car Features:



Training the classifier

(car_cv.py lines: 208-294)

I chose to use a linear support vector machine classifier to detect cars in images. I chose a linear SVC because it was already proven to have good results in image classification. I trained the classifier on 80% of the Udacity dataset and tested its results on the remaining 20%. The feature vector for each image was a concatenation of the flattened arrays returned from the spatial binning, histogram binning, and HOG feature extraction procedures. I then scaled the training set using sklearn's "StandardScaler" module and created a set of labels that consisted of 1's for all car features and 0's for all non-car features. The data set was then shuffled, split into a training and test set (80%/20%), and fed into scikit-learn's "SVC" module. The scaler and the SVC were then saved for use in processing the video file later.

Sliding Window Search

(car_cv.py lines: 372-449)

Reusing HOG Info

(car_cv.py lines: 401-413)

In order to speed up the process of the sliding window search I implemented a version of the search where the HOG features were calculated for the entire image once and the feature vector for each window image that is fed into the classifier got its hog features by sampling sections of that large HOG matrix rather than calculating new HOG features for every window which gets redundant because there is so much overlap in the window searching procedure.

Window Sizes

(car_cv.py lines: 381-383)

Searching different window sizes was implemented by adding a scaling factor as a parameter. The default window size used was 64x64 and if a different window size was desired then a scaling factor could be passed into the search function which scaled the photo down by the scaling factor and searched the same 64x64 window size. The end effect was the same as changing the window size but with this implementation I could still implement the HOG sub sampling and have feature vectors conveniently be in the size necessary for our classifier. The thing to remember with the scaling is to adjust for your scaling factor when drawing boxes back on to the original image *(car_cv.py lines: 431-433).*

I experimented with window sizes of 16x16, 32x32, 48x48, 64x64, 96x96, and 128x128. I got the best results with dimension sizes 48, 96, and 128. The 48x48 windows were able to find small cars as well as locate portions of medium sized cars. The 96x96 windows were able to locate medium sized cars. Finally, the 128x128 was able to locate large vehicles usually cars that were just entering the image and were closest to the camera.

Window search regions

(find_cars.py lines: 139-194)

The search regions for each window size started at pixel 400 in the 'Y' direction on the image, because this was roughly where the horizon of the road ended in the image, and swept across the entire 'X' direction of the image. The 48x48 windows introduced a lot of information for vehicles in the image but slowed down the algorithm the most because the number of windows that need to be searched is much greater since each window sweeps a small amount of space. I chose to limit the 'Y' direction of the 48x48 windows to [400:492]. The 96x96 windows swept from [400:544] in the 'Y' direction. And the 128x128 windows swept from [400:656] in the 'Y' direction. I chose to make the bigger windows search deeper into the image in the 'Y' direction because they were locating bigger cars which often were located closer to the camera.

Window overlap

(car_cv.py lines: 396-398)

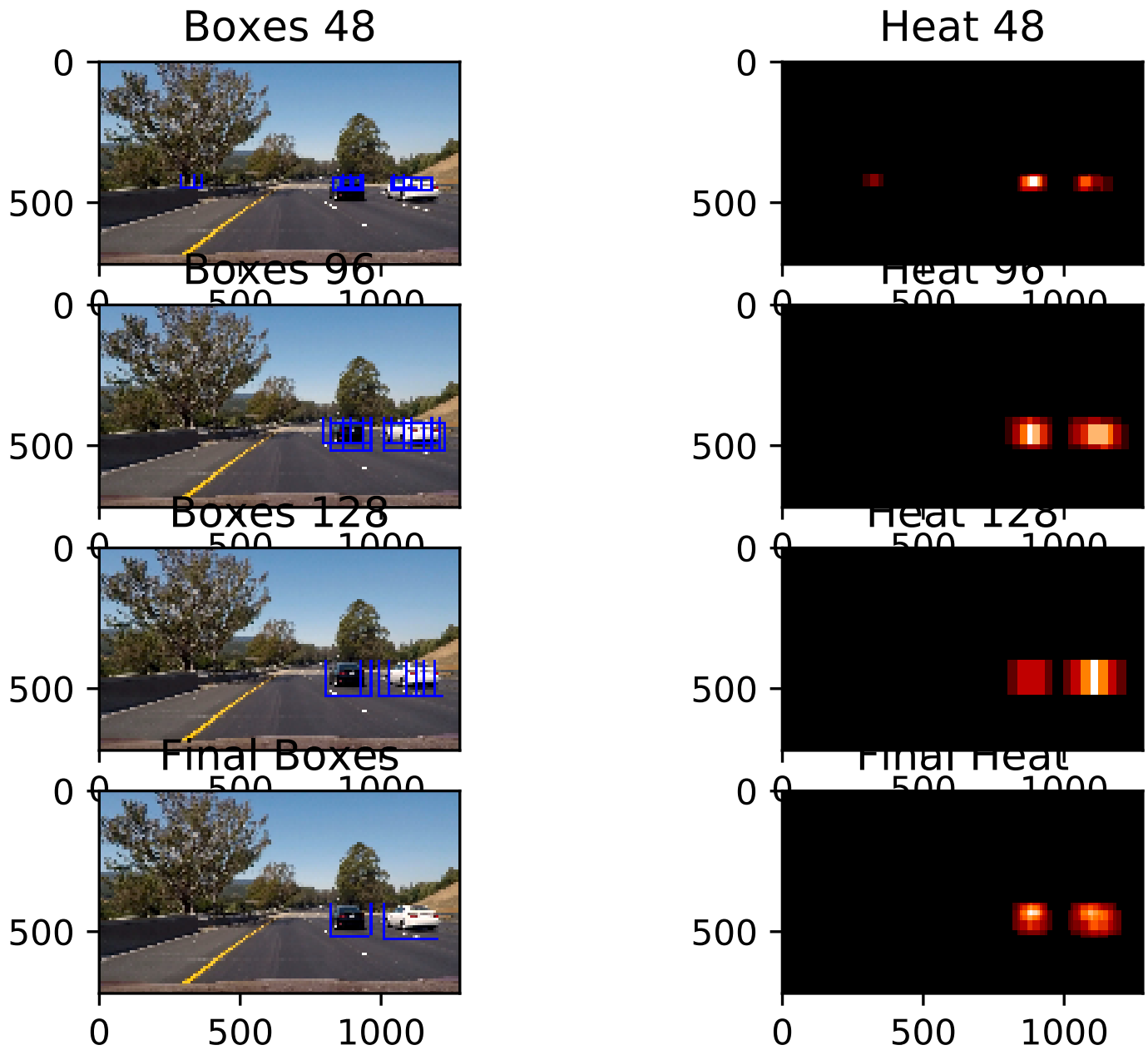
I chose to overlap windows by 75% and implemented it by stepping the default 64x64 window search 8 pixels per step. This type of overlap requires your algorithm to search more windows and thus slows performance, but it helps to not miss vehicle detection. If no overlap was used, then it would be very likely for windows to cut off important car image information by landing in positions on the photo that were awkward relative to the position of actual vehicles.

Classification

(car_cv.py lines: 407-436)

At each window the feature vector was calculated by sampling the HOG data, calculating the spatial and histogram binning, scaling the data using the same scaler that scaled the training data, and feeding the data into the SVC that was trained. If the SVC classified the window of interest as a car then the window information was used to draw boxes over the original image for visualization purposes, and create a heat map corresponding to pixels that were bounded by that box. Heat maps from each window size were then added together and subjected to some min thresholding to help remove heat from false detections. That final aggregated and thresholded heat map was then used to draw boxes around contiguous areas of heat. Scipy's "label" function was helpful in labeling the areas of contiguous heat.

Examples of Window Search results:



Video Implementation

Video Link:
https://youtu.be/drQrDs_G8gI

False Positive Filtering
(find_cars.py lines: 222-237)

I filtered for false positives by saving the resulted heat map from the last 30 frames and requiring that any new heat also existed in those last 30 frames. Although 30 frames of memory is much higher than the suggested advice of 10 frames from the instructor, I found that requiring that heat exists in all of the last 30 frames really helped to reduce false positives from bushes, signs, and other traffic structures. The only drawback is that at 30 frames per second it takes about a second for a new vehicle on the screen to get detected.

Discussion

One of the biggest problems I encountered in the project is performance. Searching the windows of each frame was costly in terms of time and required 10+ minutes of processing time to process a 50 second video. Obviously the current state of my algorithm is not suited for real-time information gathering. I know that in practice we would have the aid of beefy GPUs to speed up the window searching portion of the algorithm but I also think that some software optimizations can be implemented to speed up the process. Once easy optimization would be to shrink the image of each video frame down. This would require me to also change the "default" window search size in my window search algorithm, but I would still be able to perform similar window searches.

The other major problem encountered in this project was trying to limit the false positive detections. As stated earlier, I used a long memory horizon to help filter false positives and it worked well. If for some reason it is deemed unsafe to require a vehicle to be on screen for a full second before detection, then I would shorten my memory horizon back to 10 frames and try to gain an edge in the training phase of building the classifier. The test accuracy on the classifier I used for the project was 99%. This sounds really awesome but because we are searching so many windows the amount of false detections that appear on screen, at that level of accuracy, is not acceptable. I think accuracy can be increased in this phase of the model by possibly exploring other models and combining the results in an ensemble methods manner. Also I would like to spend more time learning about the feature space to ensure I am using all of the best features.