May 10, 2019

Vehicle Detection and Tracking

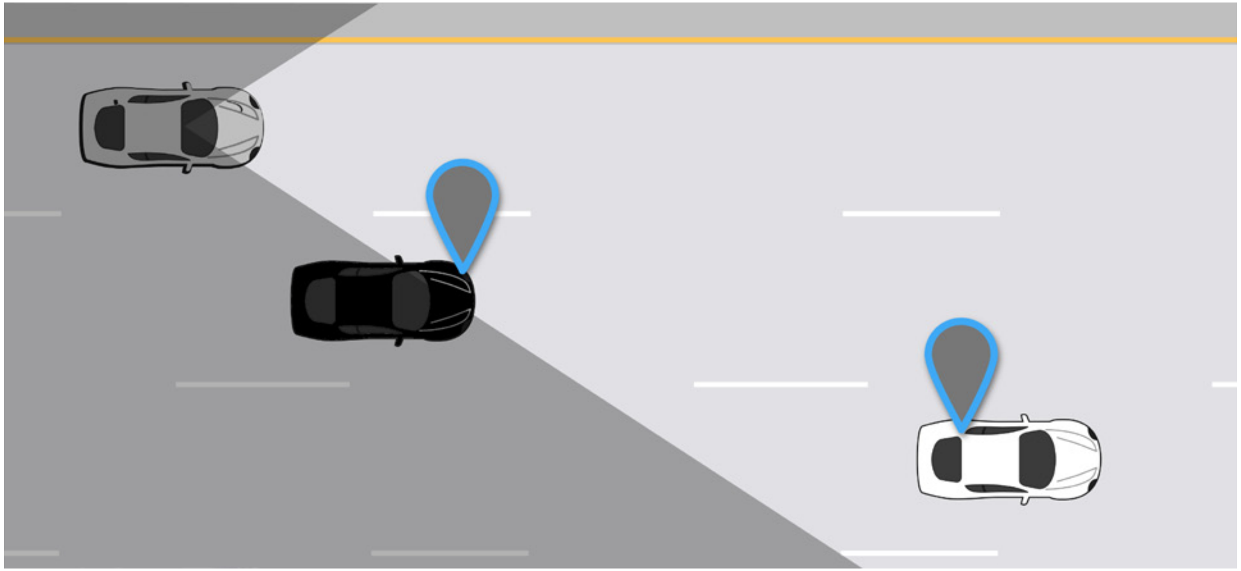
Proximity Algorithm research by James W. Dunn

Goal: Annotate a
video sequence with identified
vehicles in adjacent lanes.

Contents

Analysis

Traffic is a reality. To warrant safe travel from point A to point B, obstacles must be identified and avoided. The supplied project video from a forward-facing dashcam contains 50 seconds



Field of view, illustrating the aim to identify vehicles as they transit the camera viewport.

of highway driving in which two vehicles pass through the camera field in adjacent lanes on the right. Though objects are easily seen and understood by humans, the goal here is to deconstruct the scene with a set of computer vision techniques and annotate each frame with an approximate bounding box around the vehicle location.

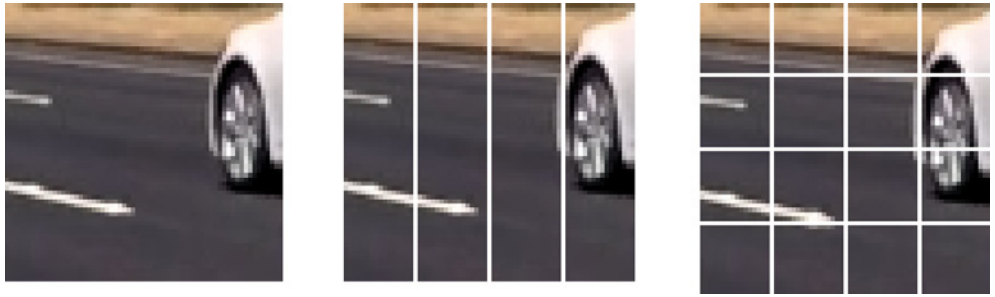
While the source video contains additional ‘safe’ vehicles in the distance and across the median, the project scope is limited primarily to identifying the ‘cautionary’ elements in the two adjacent lanes (a white Lexus LS460 and a black Lexus IS250).

Aligning with the goal of safe travel, recurring themes include accuracy, precision, and performance. In addition to locating obstacles correctly and exactly, an efficient solution can be implemented in real-time as one of the many components in a self-driving car. Peripheral early-detection can help with accuracy and inform the navigator of potentially hazardous conditions.

Input data quality

Reasoning that accurate input leads to improved output, an inspection to determine quality of the supplied training data revealed both ‘good’ and ‘bad’ data. For example, on the plus side, 1) the majority of vehicles contain the features of passenger cars, as opposed to trucks, busses, and other transport; 2) the angle of view aligns primarily with that of the supplied video; 3) mostly consistent spatial scaling [64×64 images]; 4) mostly consistent lighting [daylight]. On the negative side, 1) ambiguous objects; 2) signage unrelated to US roads; 3) skewed vehicle aspect ratios; 4) primary focus on the back of vehicles with some negative reinforcement of the leading edge. As a result, the input data was filtered for unlikely extraneous elements such as signage, odd skews, sky, median, and oncoming vehicles. New data was added to complete the vehicle angles along the lane trajectory.

Training a classifier with whole images of vehicles may lead to “all-or-nothing” situations and/or less accuracy predicting images with partial vehicles. This concept led to experiments with 16 pixel vertical strips to identify incoming vehicles on the periphery. To illustrate, the following 64×64 pixel image is categorized in the ‘Extras’ dataset as ‘non-vehicle’:



Slicing a 64×64 ‘non-vehicle’ image into vertical strips, then horizontally into cells of 16×16 pixels.

While the majority of the image does not contain a vehicle, if it is quartered into vertical strips, the far right strip can be assigned to the ‘vehicle’ class. If each strip is further extracted into 16×16 pixel squares, then clearly three of the ‘cells’ can be assigned to the vehicle class. Additional data policies were defined as: 1) non-vehicle cells cannot contain any part of a vehicle; 2) over 50% of the pixels in vehicle cells must be recognizable automobile features.

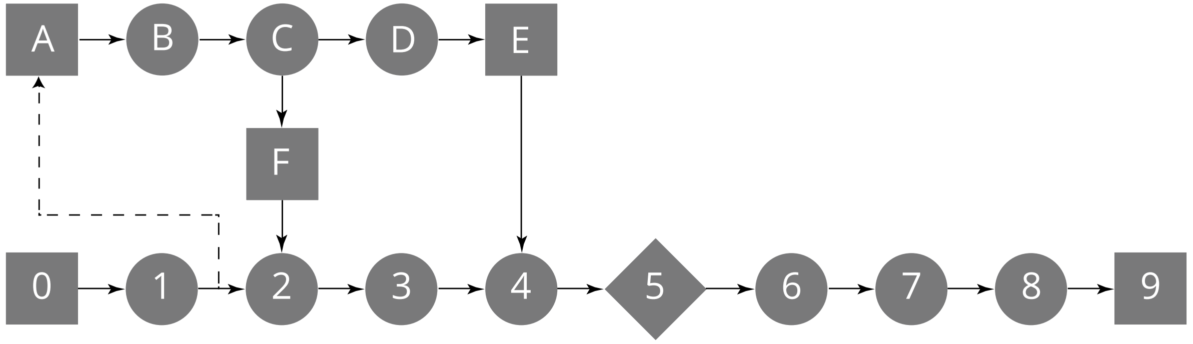
With this system, an improvement in matching incoming lower vehicle segments such as wheels/hood can be expected. Initial training accuracy results were encouraging (approx 98%). Adding spatial binning of color features increased accuracy to just over 99%.

Software pipeline design

A chain of processing elements is arranged to accomplish three tasks: load a set of training images (which have been labeled as either ‘vehicle’ or ‘non-vehicle’); extract identifying features; and train a classifier model to ‘recognize’ the difference between the two classes.

A second chain of functions is arranged to examine a video stream with three objectives: windowing on targeted regions of expected traffic, then making a prediction of what class is present based on the features perceived, and visualizing the filtered results.

Referring to the illustration on the next page, the design consists of two inter-related pipelines, A->E for training of the classifier and 0->9 for annotating video.



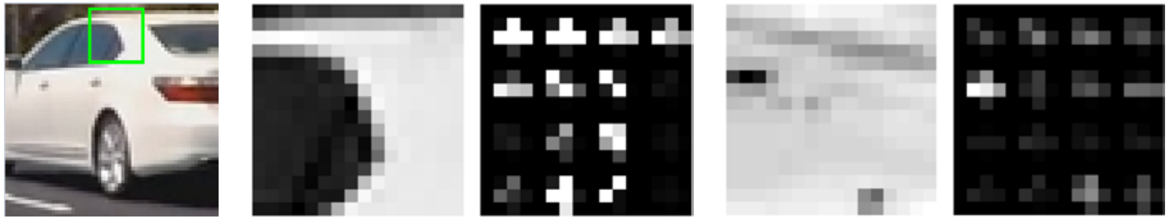
Pipeline architecture

Training images (A) are read and sliced into cells (B) and then preprocessed (C). The complete training set produces a scaler data file (F) for use by the second pipeline. The preprocessed data is used for training (D) of the classifier model (E).

Source video (0) is windowed and scaled spatially (1). After preprocessing (2), data is collected into batches (3) and sent for prediction (4) by the classifier. The results are filtered (5) where positive cells are heat-mapped (6), bounding boxes determined (7), frame annotated (8), and finally output (9). Cells can be extracted after (1) and utilized as additional training data (A) to improve the classifier.

Histogram of Oriented Gradients (HOG)

A customized high-performance implementation of the HOG method calculates the first order Sobel derivative in both x and y. The resulting coordinate pairs are converted from Cartesian coordinates to polar (angle and magnitude). Next, angles are binned and, along with magnitude, grouped into 4×4 sets (also called *cells* in HOG terminology). The magnitudes are then multiplied by the number of angle occurrences (technically, summed multiple times). Finally, the resulting vector is normalized as a block to unit length to enhance invariance to changes in illumination. [The 'L1-sqrt' method is also implemented as of Jan'17 in *skimage.feature.hog*() at the block level.]


Example of visualized Histogram of Oriented Gradients of both 'vehicle' and 'non-vehicle' class cells.

The L1-norm of a non-negative vector v is defined as the sum of its elements:

$$\|\vec{v}\|_1 = \sum_{i=1}^{n} v_i$$

The HOG feature vector is computed as the element-wise square-root of the histogram vector scaled by the L1-norm. Epsilon (ε) is a small constant to avoid division by zero:

$$\vec{f} = \sqrt{\frac{\vec{v}}{\|\vec{v}\|_1 + \varepsilon}}$$

Determined through iterative experimentation to minimize the classifier loss function, the final selection of HOG parameters include color space: YCrCb; orientation bins:12; *cell* size: (4,4); color channels: all three. Each channel produces a 192-element feature vector.

Code reference: The implementation is defined as function *histoGrad*() in p5extract.py at line 13.

Spatial binning of color

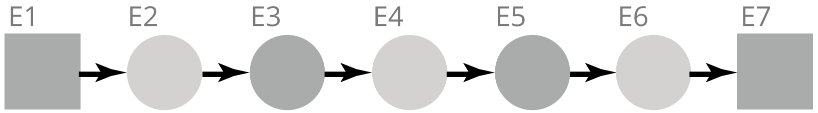
While HOG features inform the classifier of orientation, no color information is present. Coarse spatial binning provides additional features in the scene, such as color and position of black, white, brown, gray, blue, etc. Each input image in the YCrCB color space is resized to a 4×4×3 array and flattened to a 48-element feature vector.

Example of visualized coarse spatial binning of color.

Code reference: The implementation is defined as function *get_spatial*() in p5extract.py at line 38.

Classifier

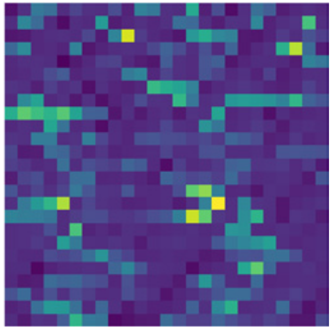
A Keras non-linear binary classifier running on a discrete GPU is defined as follows:



Classifier model

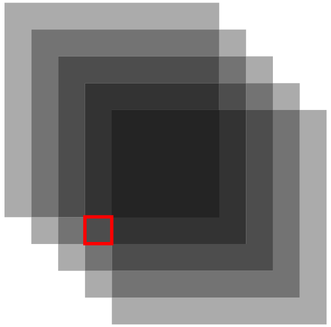
The input (E1) consists of a one-dimension 624-element feature vector. A dropout of 0.5 (50%) is applied (E2) followed by a 255-node fully-connected layer (E3) with ReLU activation. Another dropout of 0.5 is applied (E4) and followed by a 127-node fc-layer (E5) with ReLU. A final dropout of 0.5 is applied (E6) before a final 1-node fc-layer (E7) with sigmoid activation to produce a probability output in the range of 0 to 1. The model is trained with an Adam optimizer and a logarithmic loss function (binary_crossentropy).

The input vector is composed of the three appended 192-element HOG feature vectors (one for each color channel) and further appended with the 48-element spatial binned color vector. The collective column for each feature is scaled to zero mean with unit variance using sklearn StandardScaler. (The feature values are centered at zero, equally scaled, with standard deviation of one.)


Visually stacked input vector

Code reference: The classifier model is defined at line 152 in p5train.py and trained at line 174. The function to prepare the training data is defined as *extract_features*() at line 33 in the same file.

Sliding window search

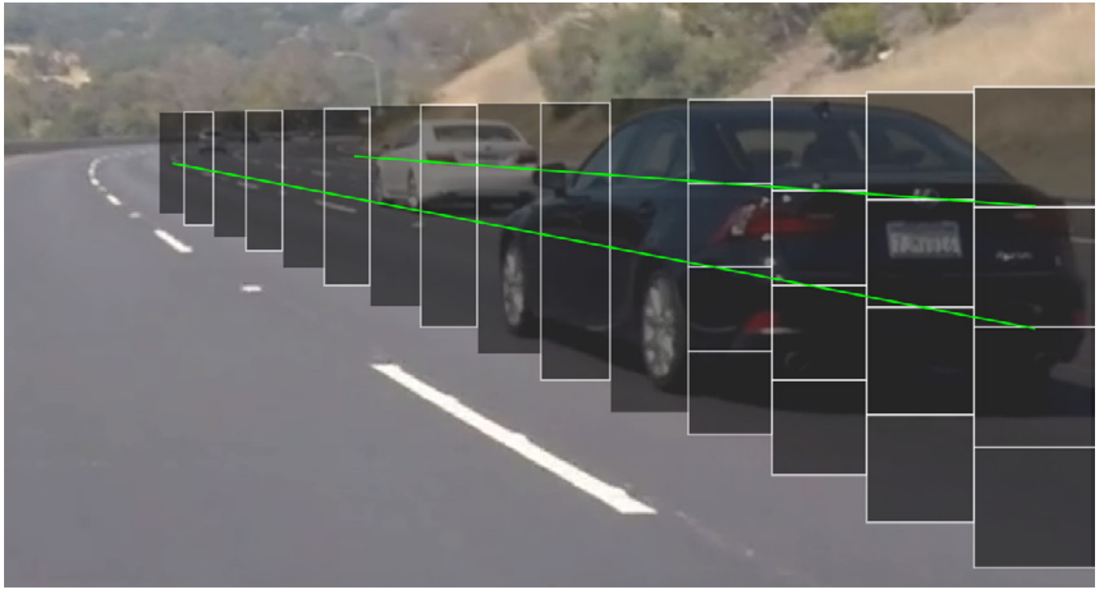

Sliding windows reduced to cells

Experiments with designing several sliding window arrangements reveal a cellular grid structure (the underlying lowest common denominator of overlapping windows).

A framework of cells is constructed along the highest probable travel paths of vehicles in the adjacent lanes.

The lanes are overlaid with 15 contiguous strips having an aspect ratio of 4-to-1, fitted along the lines leading to the vanishing point. The height of the ‘closest’ strip on the right side of the screen is sized to the height of a standard passenger car, similar to the training data.

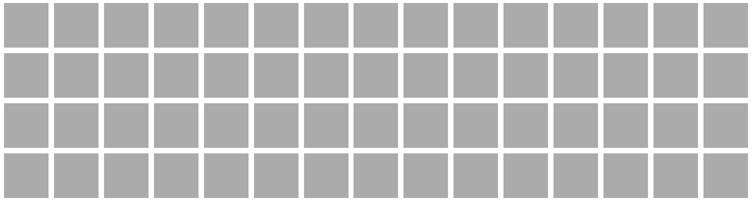
Approximations of the windows are diagrammed below. The trajectory path of lane 1 aligns with the center line of the middle two rows of the window grid, while the trajectory of lane 2 aligns with the center line of the upper two rows. This arrangement creates two effective spatial scales for the classifier to learn: 1X and ½X.


Region of interest consists of sliding windows along probable vehicle trajectories.

In line with performance goals, resampling is avoided. Due to the cellular architecture, adjacent cells group into a multitude of overlapping window combinations which can be bounded at the maximum extents.

Normalized Lane Space (NLS) grid

Sampling from this manifest of regions and scaling each image to 16×16 pixel produces what can be termed ‘normalized lane space’ or NLS, which is represented by a two-dimensional matrix of 4 rows and 15 columns.



Normalized lane space matrix representing window regions of interest.

The number of possible sub-rectangles in an $m \times n$ array of cells is given by the formula:

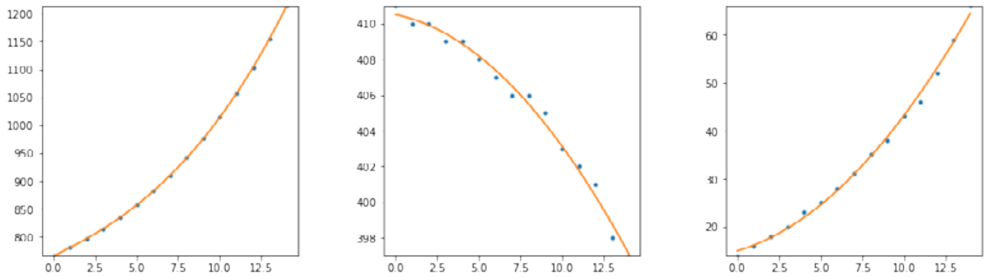
$$\text{total rectangles} = \frac{m(m+1) \times n(n+1)}{4}$$

Thus, the NLS grid structure provides a total of 1200 sub-rectangles (including squares).

Sample points collected from the overlay illustration were fitted to polynomials.

Let X and Y be screen coordinates and S be the size of a screen cell.

$$X = 0.050x^3 + 0.567x^2 + 14.164x + 765.433$$
$$Y = -0.0549x^2 - 0.188x + 410.5$$
$$S = 0.179x^2 + 1.020x + 14.985$$



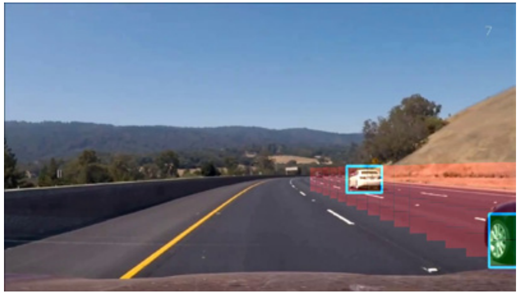
Plot of X, Y, and S

In the implementation, a video frame is sampled in strips determined by inserting the NLS matrix column index (denoted by lower-case x) into the above polynomials: The the resulting screen coordinates are the top-left point of a 4 row × 1 column (4S × S) strip. A square of size S is sampled and scaled to 16×16 pixels. The S value is repeatedly added to Y to move down the screen to sample the next 3 cells. This strip locating process is repeated 15 times to sample a total of 60 screen cells along the path.

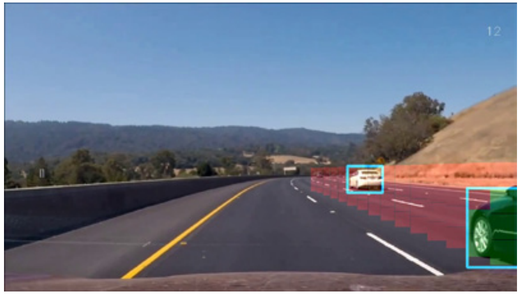
Code reference: The sliding window search system is defined as *processStrip*() at line 183 in p5pipeline.py with mathematical support from *transformIdxToScreen*() defined at line 79.

Operation of the pipeline

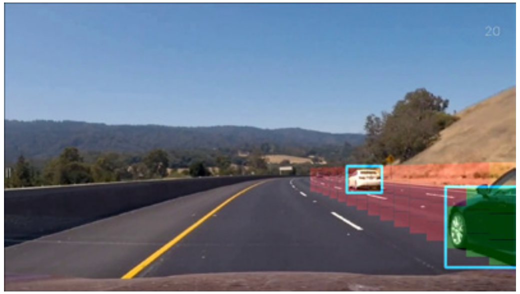
Below are sample images demonstrating the pipeline in action.



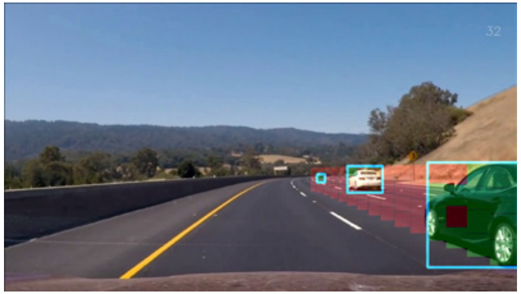
Note the early detection of the black vehicle



Red cells are negative, green cells are positive



Amber cells are predominantly white



Bounding boxes are indicated in azure blue

Classifier reliability improvements: 1) eliminate ambiguous images such as cells containing less than 50% vehicle features; 2) capture negative cells between two positives for further manual assessment with potential inclusion in the training data; 3) add spatial binning of color.

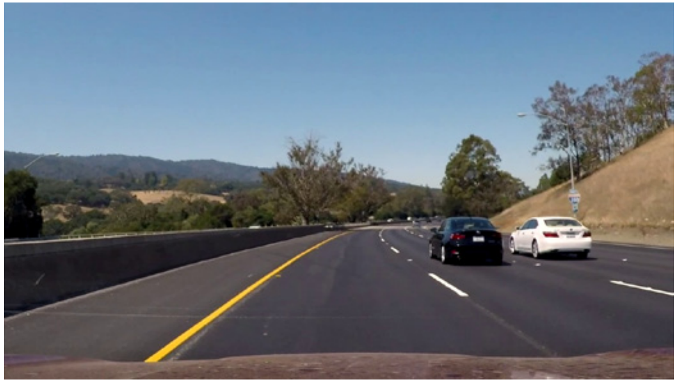
Classifier performance improvements: 1) add batching layer to improve prediction response time by reducing the overhead of making multiple calls to the GPU per frame processing cycle. 2) alternate video frame processing.

Video implementation

The following example images were produced at the output of each stage of the pipeline (steps 0 through 9 in the design diagram on page 7). For consistency, all visualizations are from video frame 922.

Step 0

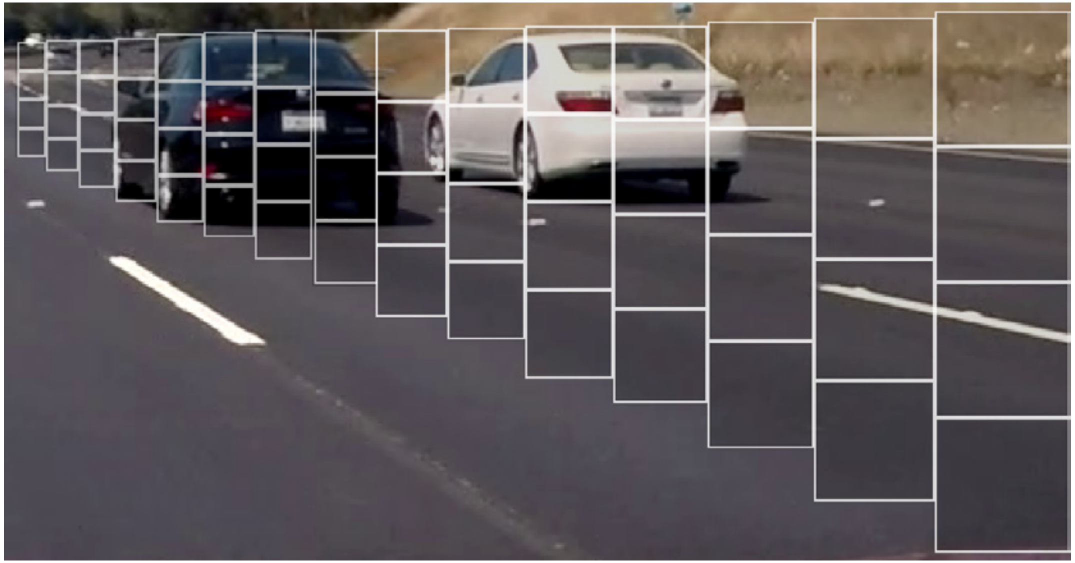
The initial step in the pipeline reads a source frame from the video file.



Source video frame 922

Step 1 (part A)

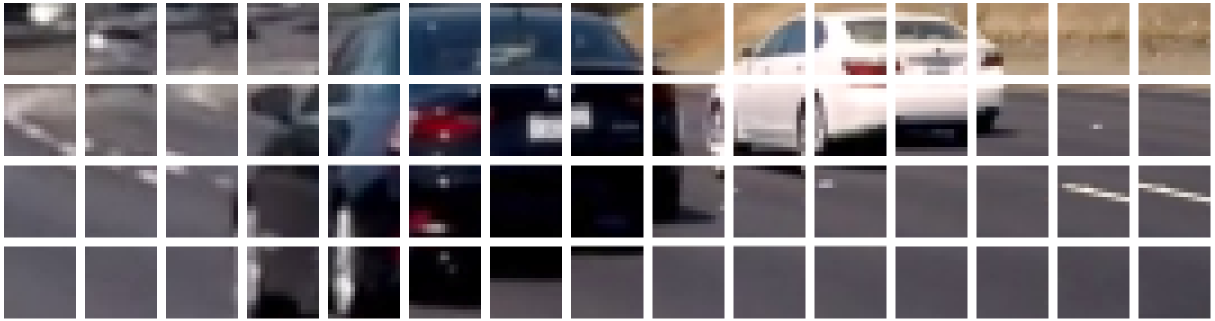
Windowed snapshots are taken from the frame along the probable vehicle trajectories.



Sliding windows cell placement along trajectories

Step 1 (part B)

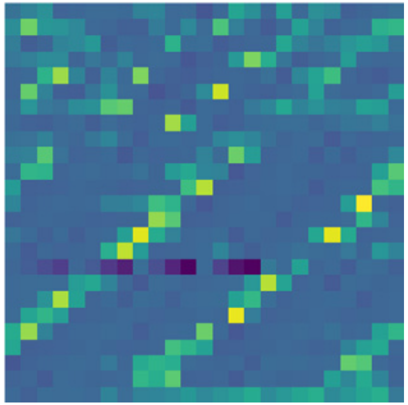
Each snapshot is spatially scaled to 16×16 pixels.



Rescaled into NLS grid cells

Step 2

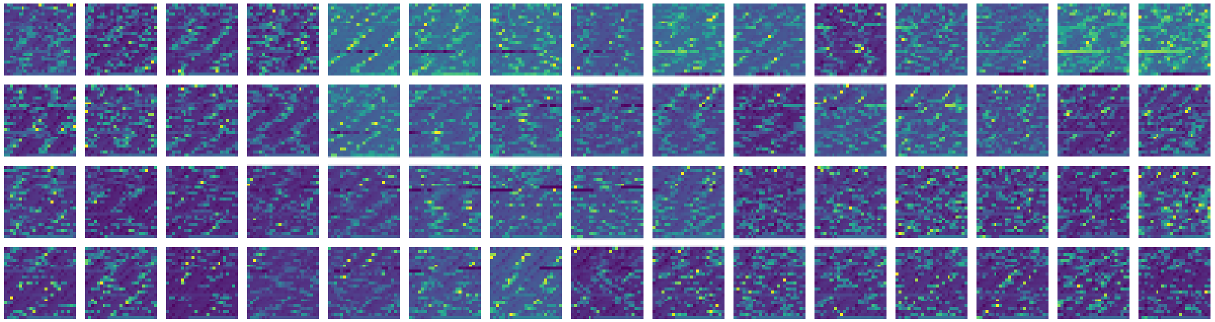
Each cell is preprocessed by 1) converting to YCrCb color space; 2) extracting histogram of gradient orientation features; 3) extracting spatial binning of color features; and 4) transforming with the zero-mean unit variance scaler determined by the training data. Below is a 624 element preprocessed vector (squared into a 25×25 array and output as a PNG file).



Preprocessed vector from x:834, y:408

Step 3

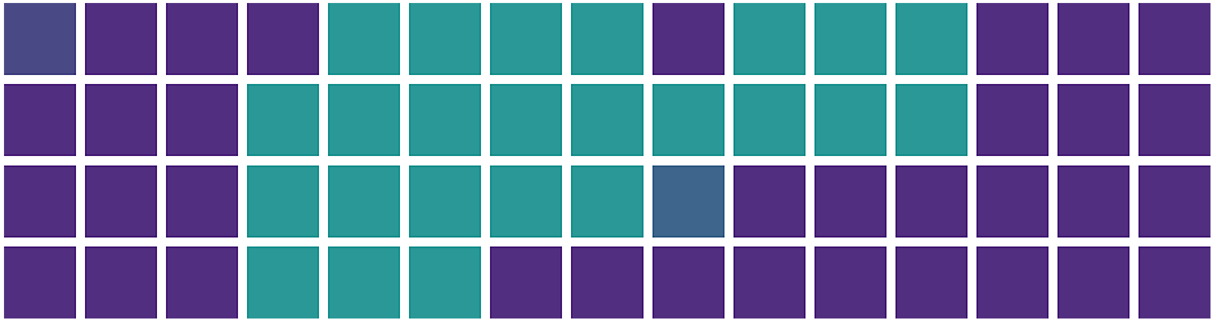
The 60 cells are collected into a batch. In the following data capture, the vectors are ready for presenting to the classifier.



Preprocessed batch of feature vectors

Step 4

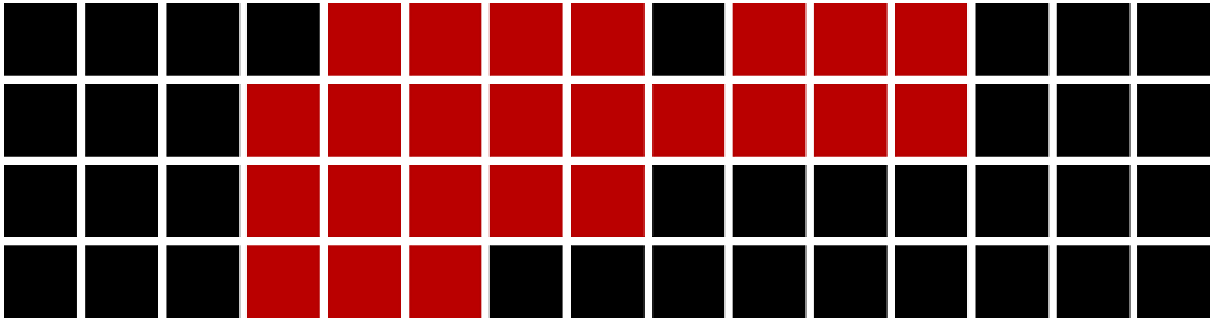
The output of the classifier ranges from 0 to 1 which translates to a prediction ranking captured below (in 5 discrete levels) where the color *violet* through *aqua* indicate the prediction value.



Prediction results

Step 5

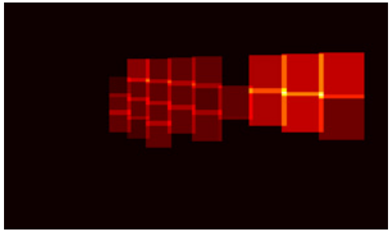
Cells with a confidence value over 0.9 are considered a probable vehicle.

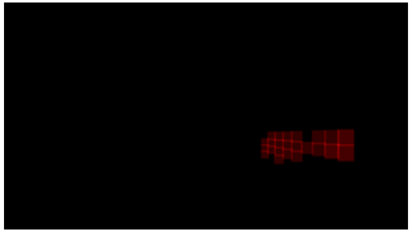


Red cells indicate a positive match to the vehicle class

Step 6

Cells are added to a queue of length 50, enough to span several frames. Each cell in the queue represents a potential on-screen vehicle-class cell, and therefore added to a 'heat' map. Cells are rendered in the heat map with an additional 5 pixels added to the side measure in a screen-sized array. This slight overlapping fuses the cell into neighboring structures. False positives are reduced by filtering areas from the heat map below a threshold of 1.


Heat map (close up)

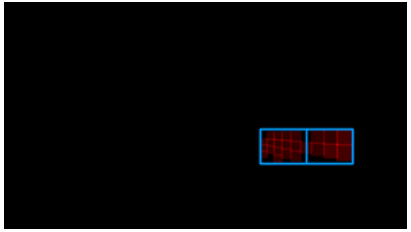

Position in frame

Reference frame

Code reference: False positives are filtered by the function *apply_threshold*() defined at line 98 in p5pipeline.py and invoked at line 139. The key function *addBoxToList*() is defined at line 128.

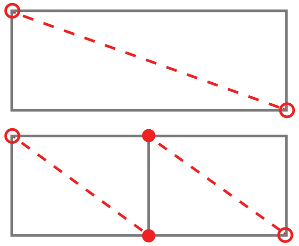
Step 7

Bounding boxes are determined from the heat map using the *label* function from SciPy.ndimage.



Heat map yields bounding boxes

Reference frame



Box bisection

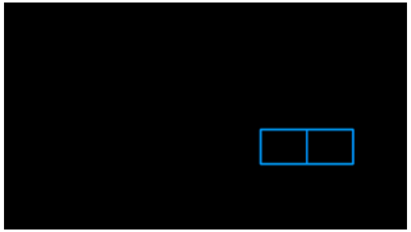
Code reference: Two rules are applied during box construction in the function *draw_labeled_ bboxes*() defined at line 103 in p5pipeline.py:

1) Abnormal aspect ratios are created where two vehicles are near each other. Any bounding box greater than 100 pixels in height with a width over 2.68 times its height is bisected vertically into two adjacent boxes.

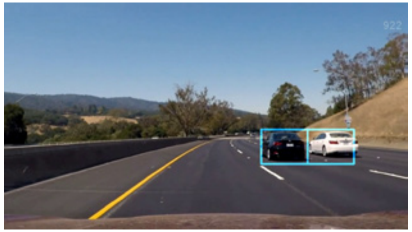
2) To assist the false-positive filter in cases where adjacent cells are rejoined, filtered out, yet a remaining common joint exists, only bounding boxes of width and height greater than or equal to the minimum on-screen window size [14 pixels] are rendered.

Step 8

The filtered bounding box list is rendered on a diagram overlay...


Overlay complete and ready

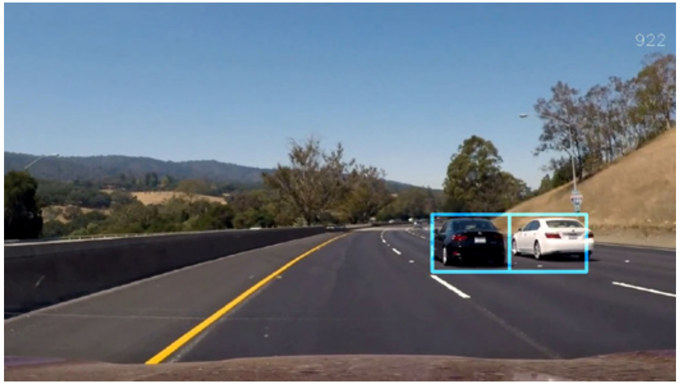
Reference frame


Blended result

...and subsequently blended with the video frame using the OpenCV *addWeighted*( ) function.

Step 9

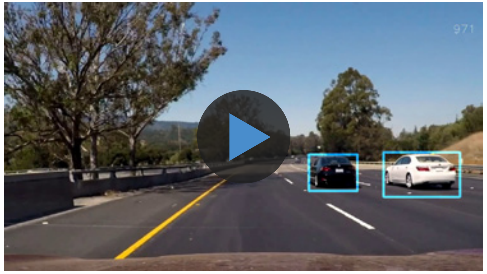
The final step in the pipeline is appending the processed frame to the new video sequence file.



Output from the pipeline

Final video

The final annotated video can be found in the *output_images* folder or on Vimeo through the link below. Additional videos in the folder include diagnostic tests. The folder also contains a subfolder of images of captured at the output from each stage of the pipeline.



http://vimeo.com/208120673

Discussion

Approach

The previous project inspired placement of sliding windows in perspective. The normalized cellular grid structure was discovered as a result of experimenting with scaling along the trajectories and searching for a lowest common element that would match the training data size and shape.

Iterative improvements made during development: Generalized the classifier further by training with additional data from the P4 challenge video and regularizing with dropout; Cycled outputting cells over a threshold and then sorting these for retraining the classifier; Added spatial binning of color to the feature extraction stage.

Performance

The targeted region of interest helps reduce search time. The pipeline processes the video stream at 10.11 fps on a Xeon W3520 (equivalent to i7 920 first generation) at 2.8GHz with 12GB memory (Windows 10). The system includes a discrete GPU (GTX 1070). Other performance enhancements include alternating frame processing. Frame rates using scikit-learn.svc function did not meet the performance prerequisite. Scikit-image *hog*() function performance was not adequate to meet the performance prerequisite. A customized implementation of the HOG method was derived from the open source code found in the OpenCV examples and informed by references [1], [2], [3] and through online forum discussion.

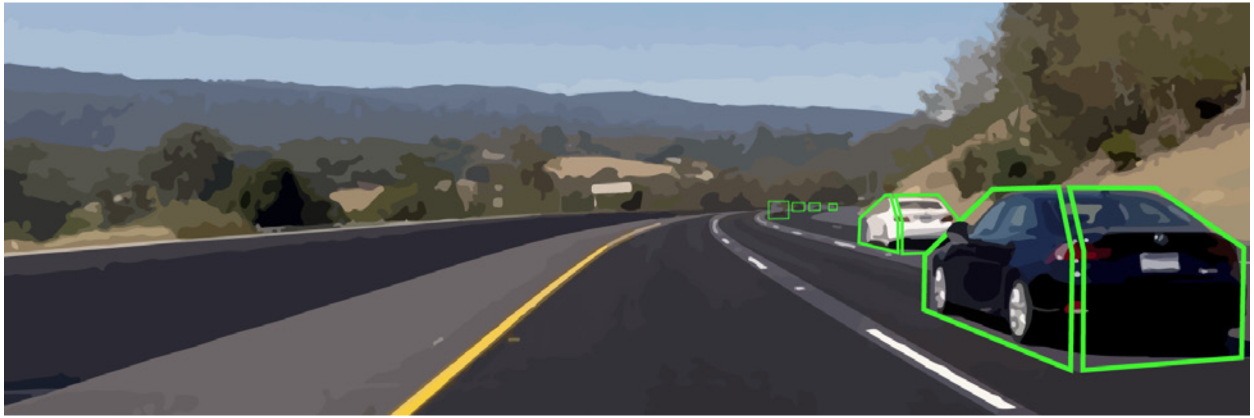
Challenges

If a positive cell has no immediate neighbors, it may be rendered as an orphan bounding box, or sparse cells across a larger vehicle, as evidenced in the p4 challenge video. If subjected to new conditions, the system needs to be retrained with additional data. Occasionally, *cv2.cartToPolar*() returns a value of 2π which over-bins the angles (expecting the range 0 through 11 and never 12). To resolve this, the angle is multiplied by (1-ε) which shifts the binning negligibly. A large and refined training dataset would extend this proof-of-concept to a wider range of vehicles.

Further research

The current solution is tolerant to the curvature of the road in the supplied video; however, higher rates of curvature would misalign the lanes. An area to explore is modifying the trajectory polynomials with data derived from the overhead view of the advanced lane-finding project. The NLS method can be extended to overlapping cells for increased location accuracy.

Find predominant color of vehicle to determine a threshold to produce a binary image of the vehicle using the color range filter. Additional locator regions can be positioned to find the edges of the identified vehicles. With the leading, trailing, and top edge positions, a more accurate target location can be rendered…ideally a rig transformed to fit the vehicle as perspective changes. The classifier can be trained with a label to identify a tail light class to determine an additional point for the rig transform (see visualization below).



Proposed rig registered to accurate vehicle position

References

Analytical tools
- Microsoft Excel
- Adobe Illustrator

Scikit-image
- scikit-image.org

Open Source Computer Vision Library
- opencv.org

NumPy Library
- numpy.org

Histogram of Oriented Gradients
- en.wikipedia.org/wiki/Histogram_of_oriented_gradients [1]
- pascal.inrialpes.fr/soft/olt
- github.com/scikit-image/scikit-image/blob/master/skimage/feature/_hog.py [2]
- robots.ox.ac.uk/~vgg/publications/2012/Arandjelovic12/arandjelovic12.pdf [3]
- github.com/opencv/opencv/blob/master/samples/python/digits.py