# Vehicle Detection

*Fabrizio Frigeni, 2017.10.20*

## Specifications

The goal of this project is to detect vehicles in images taken by a car's front camera. The proposed solution is to train a linear SVM classifier with the HOG features of vehicle's images and use it for detection by sliding windows of different sizes on the original image.

## Implementation

All the required functions are implemented in the library file *mylib.py*. The program *main.py* is then used to read each frame of an existing video, process each frame with a pipeline of functions from mylib.py, and then save the frames back into a new video.

The following frame processing functions are described in the rest of this document:

- HOG features extraction
- Classifier training
- Sliding windows
- Windows check
- Heat map
- Bounding boxes

### *HOG features extraction*

In order to detect cars in an image we use the Histogram of Oriented Gradients (HOG) features to differentiate between areas with and without cars. It is a powerful algorithm, often used for face detections applications.

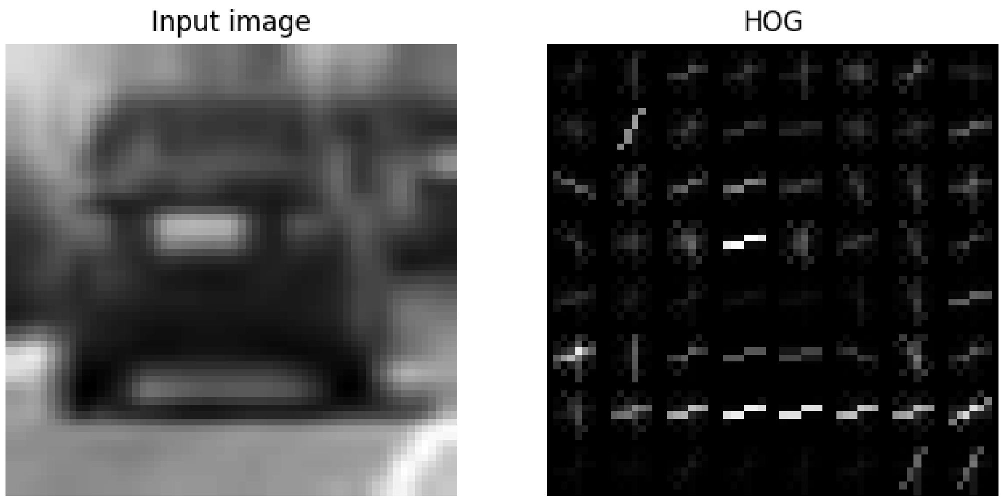
The scikit-image package provides a function to directly extract HOG features from an image:

```
from skimage.feature import hog
```

This function subdivides the image in several blocks, each block is made up of NxN cells and each cell contains MxM pixels. The gradients of all pixels in a block are averaged and classified according to their orientation. The number of possible gradient orientations must be parameterized, together with the size of each block and cell:

```
orient = 9          #number of possible orientations of the gradients
cell_per_block = 2  #number of cells per block (2x2)
pix_per_cell = 8    #number of pixels per cell (8x8)
```

The result of HOG feature extraction is shown in the next image:



## *Classifier training*

As classifier we use an SVM linear model, which is extremely compact, therefore fast, but is also accurate enough for our needs. The input to the classifier is an array of HOG features extracted from a 64x64 image; the output is binary value: 1 (car), 0 (not a car).

The classifier is trained on a large and well-balanced dataset: 17760 images, of which 8792 are cars and 8968 are not cars.

The HOG features are extracted from the LUV channels of each image, which gave the best classification error after experimenting with different color spaces. The final number of features for each train sample is: 3 (L,U,V channels) * 7x7 (grid points) * 2x2 (cells) * 9 (orientations) = 5292.

The input features are also normalized to have zero mean and unit variance:

```
scaler = StandardScaler().fit(X)
scaled_X = scaler.transform(X)
```

Finally, the dataset is randomly shuffled and 20% of it (3552 images) is put aside as a test set.

```
X_train, X_test, y_train, y_test = train_test_split(
    scaled_X, y, test_size=0.2, random_state=rand_state)
```

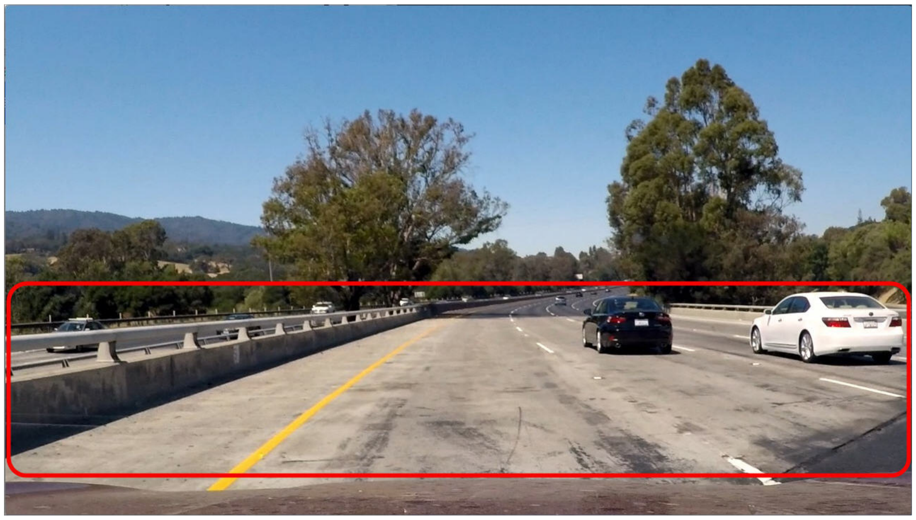
We trained the classifier with different values for the C parameter, but we found the default C=1.0 to be the best fit. We achieve a *0.988 accuracy* on the test set.

Since the feature extraction and the training take quite some time, both the classifier model and the dataset features are saved in pickle files for quick access during the debugging phase.

## *Sliding Windows*

Once the classifier is trained and appears to work well on test images, we can apply it to the incoming frames from the car's camera.

To reduce computational time on each frame we only concentrate on a specific region of interest (ROI) and extract the HOG features from that part of the image. After all, cars will very unlikely appear on the upper half of the frame:



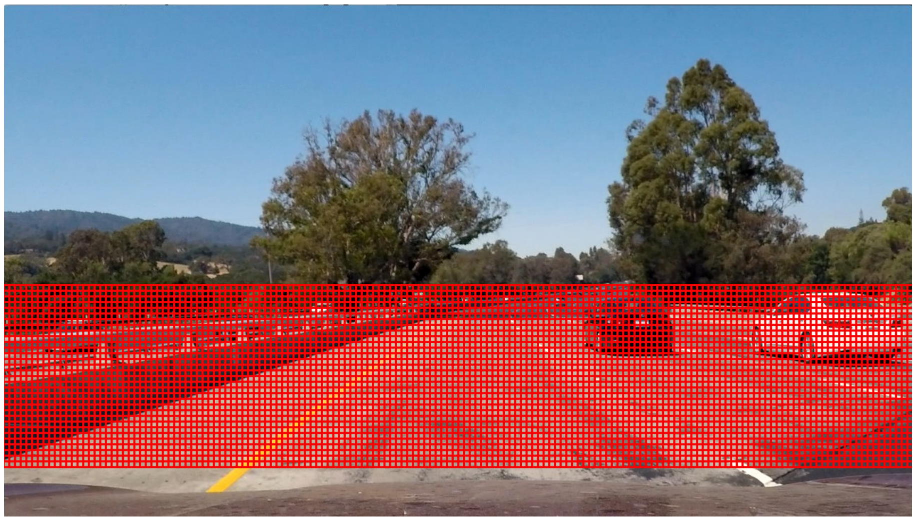
The size of this ROI is 1280x290 and is hard-coded in the program.

```
#extract HOGfeatures for the whole ROI in the image
Ymin = 360
Ymax = 650
start_corner = [0,Ymin]
end_corner = [img.shape[1],Ymax]
img_features = HOGfeatures(img, start_corner, end_corner, draw=False)
```

The extracted HOG features for the highlighted ROI above clearly show the two cars on the right side:

Next, we slide the classifier around the ROI, using windows of different sizes because cars can appear at different distances from the observer.

We use small (64x64 px), medium (128x128 px) and large (192x192 px) windows and steps of 8 pixels in both horizontal and vertical directions. The result is a dense grid of overlapping test windows of different sizes:



## *Windows check*

The next step consists in running the classifier on each of the window to detect which one contains a car.

Since the classifier was trained on HOG features extracted from 64x64 images we need to rescale all windows to this same size. We subsample the HOG features array extracted from the ROI with numpy slicing techniques:

```
        subsample = (window[1][1]-window[0][1])//64
        features = np.array(img_features)[:, (window[0][1]-Ymin)//8:(window[1][1]-
Ymin)//8-1:subsample,window[0][0]//8:window[1][0]//8-1:subsample,:,:,:]
```

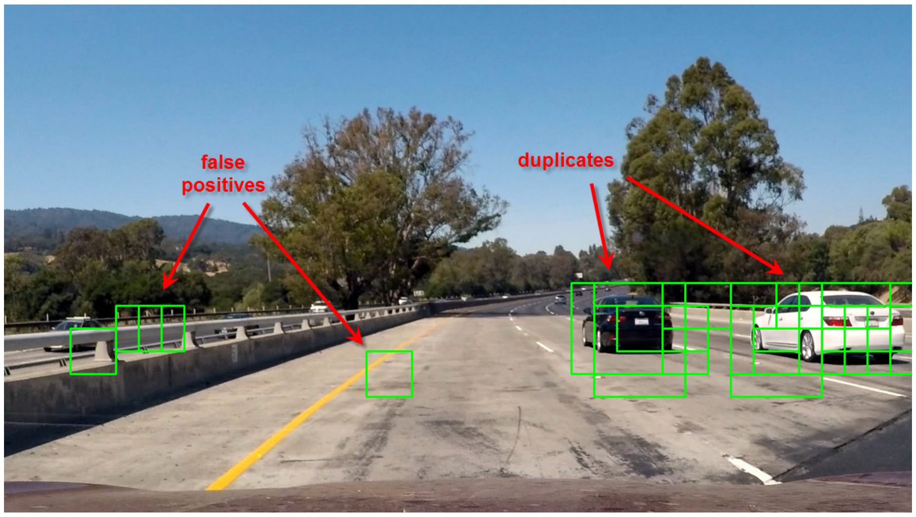
Then we scale the features and test the classifier on them:

```
        features_scaled = scaler.transform(features.ravel().reshape(1,-1))
        prediction = model.predict(features_scaled)
```

If the prediction is positive a car was detected in that window.

However, since we are scanning several overlapping windows, it is likely that multiple windows detect the same car. These are called “duplicates”.

Also, since the classifier is not 100% accurate, it might detect cars where there are actually none. These are called “false positives”. Some examples are shown in the image below:



### *Heat map*

An easy way to remove both false positives and duplicates is the use of a heat map.

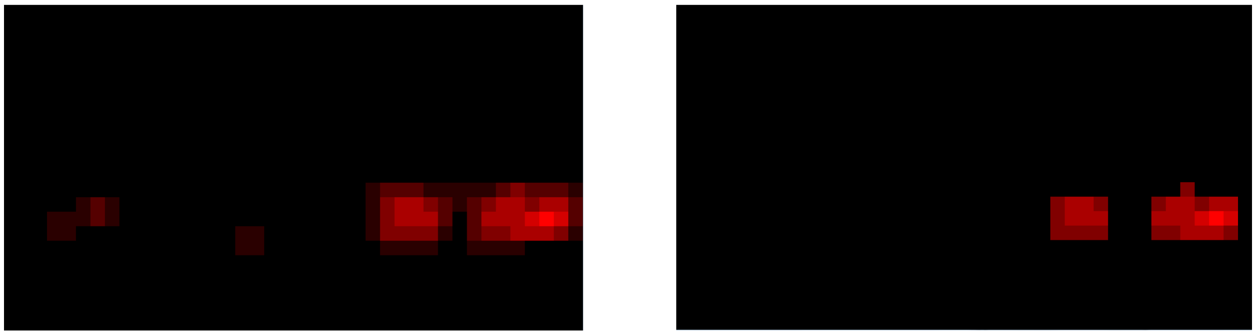
We start from a black image and then color all pixels belonging to all the windows that detected cars. Pixels included in more than one window will have higher intensity than others.

```
heat_map = np.empty_like(img)
for window in windows_with_car:
    heat_map[window[0][1]:window[1][1], window[0][0]:window[1][0],2] += 1
```

Pixels with false positives will usually have a low number of detections and can be filtered out by a threshold.

```
heat_map[heat_map <= threshold] = 0
```

If we build a heat map for the image above we get the results shown below: on the left for all windows that detected cars; on the right after applying a threshold to filter out false positives:



The two cars are now clearly detected and separated from the background.

### *Bounding boxes*

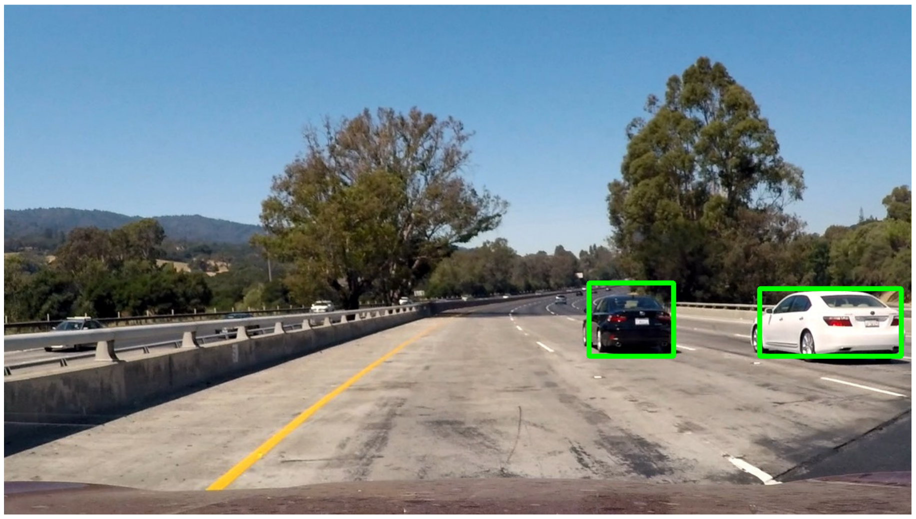
In order to highlight the detected cars in the original image we can use the label function from the scipy library:

```
from scipy.ndimage.measurements import label
```

This function returns the number of separated detected objects and the non-zero pixels belonging to each of the objects.

```
labels = label(heat_map)
print("found {0} cars!".format(labels[1]))
```

We loop over the labels array (if at least one car was detected) and draw rectangles around each non-zero region. The result is shown below:



## Pipeline optimization

We now have all the functions we need to detect cars in a single frame image. In principle, it is enough to loop over all frames of the video and apply the pipeline to each of them. In practice, however, this solution is extremely slow and will definitely not run in real-time.

By running separate parts of the pipeline independently we discovered that the computationally most expensive function is the extraction of the HOG features. We need to reduce the size of the ROI in order to speed up the process.

We observe that:

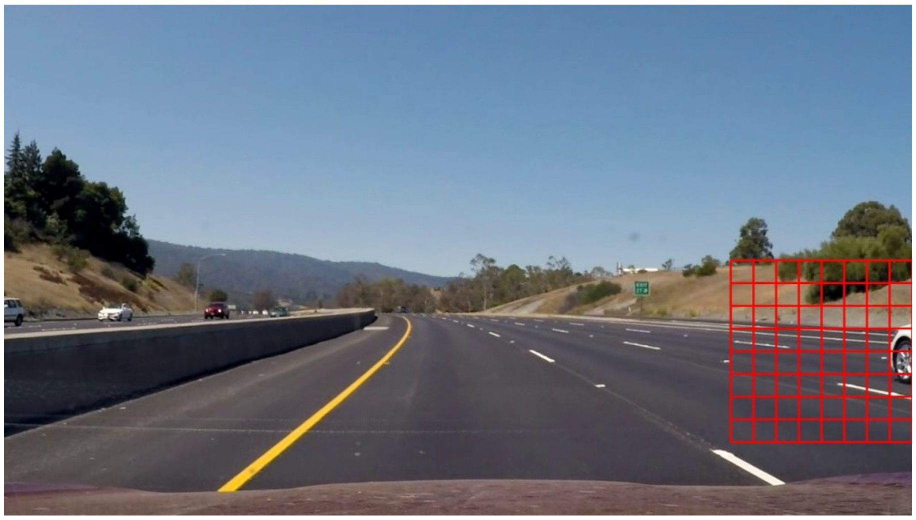
1. Existing cars do not suddenly jump around in the image, so we can predict their current position based on their past position in the previous frame.
2. New cars normally enter the frame from the sides, either on the bottom-left or bottom-right corner.

Based on these intuitions we can:

1. Minimize the area where we extract the HOG features to include only the sides of the image and a neighborhood of the positions of existing cars in the previous frame (*oldcars*). Specifically for the project video we only consider the right edge (X>1024px):

```
if (len(oldcars)>0):
    Xmin = min(1024,np.min(np.array(oldcars)[:,0]-offset))
else:
    Xmin = 1024
```

2. Do a coarse window sliding around the edges to look for new cars. Here we use only a few large windows and no filters.

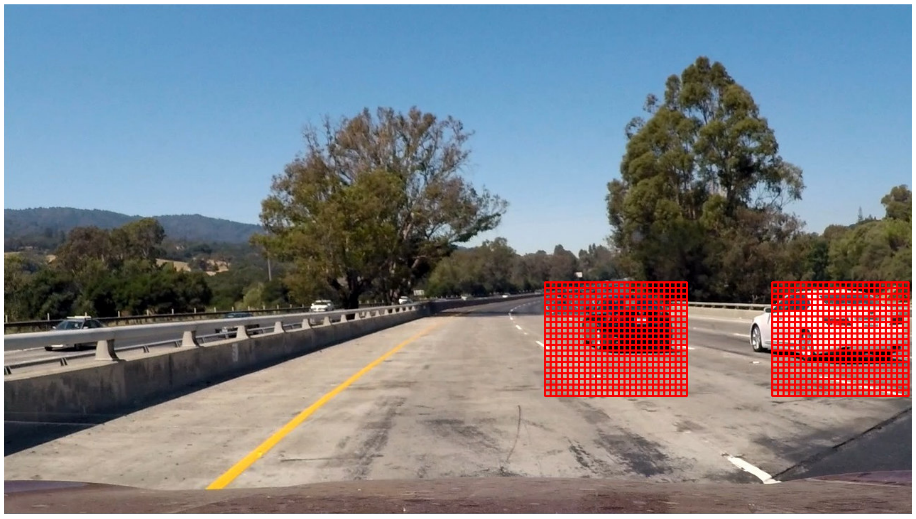


We add whatever positive result we find to the oldcar list, making sure to exclude duplicates based on their relative positions:

```
#add new found cars to old ones
newcars = Centers(new_windows_with_car)
for newcar in newcars:
    #make sure that this car is not present already
    exists = 0
    too_close = 64
    for oldcar in oldcars:
        if abs(newcar[0]-oldcar[0])<too_close
        and abs(newcar[1]-oldcar[1])<too_close:
            exists = 1
    if not exists:
        oldcars.append(newcar)
```

3. Do a much finer window sliding around the positions in the oldcar list, which now includes the “old” cars from the previous frame and the “new” cars detected by the coarse search:

```
start_corner = [max(Xmin,oldcar[0]-offset),max(Ymin,oldcar[1]-offset)]
end_corner = [min(Xmax,oldcar[0]+offset),min(Ymax,oldcar[1]+offset)]
```



This finer analysis includes filtering with a heat map and reduces the number of false positives possibly introduced by the coarse search for new cars.

The final video was saved in the *project_video_SVM.mp4* file.

## Comments

The optimization of the pipeline reduces the computational time by an impressive 6-times factor. However, at about 0.5 frames per second on a regular CPU, the result is still far from a real-time implementation.

There are still some cases of false positives, which repeat for several frames, usually from the background trees. They could be eliminated with a finer grid of sliding windows and a higher threshold in the heat map filter, at the cost of additional computational time.

Alternatively, we could add color-related features to the classifier and probably achieve a better differentiation between cars and trees than what the HOG features alone can do.

## Alternative solution

SVM classifiers are quickly being replaced by deep convolutional neural networks in many practical applications, so we decided to give them a try as well and see how they compare in term of speed/accuracy performance.

In particular, we use the freely available Tensorflow object detection API, which includes several different pre-trained models. We chose the ssd-mobilenet trained on the COCO dataset because it is supposed to be one of the fastest and we hope to achieve higher speed than the SVM trained on HOG features.

Tensorflow provides a template program to test the model on images, so there is not much to be added in terms of programming.

We load each frame of the video, transform into RGB space, crop to the desired ROI and pass the image to the model. The API internally scales down the image to the network's required input and normalizes it for us.

```
with detection_graph.as_default():
    with tf.Session(graph=detection_graph) as sess:
        while(cap.isOpened()):
            ret, img = cap.read()
            img_RGB = cv2.cvtColor(img, cv2.COLOR_BGR2RGB)
            img_ROI = img_RGB[Ymin:Ymax,Xmin:Xmax,:]
```

The actual detection happens here:

```
(boxes, scores, classes, num_detections) = sess.run(
 [boxes, scores, classes, num_detections],
 feed_dict={image_tensor: image_np_expanded})
```

The results are arrays of detected *classes*, together with their *scores* and their bounding *boxes*.

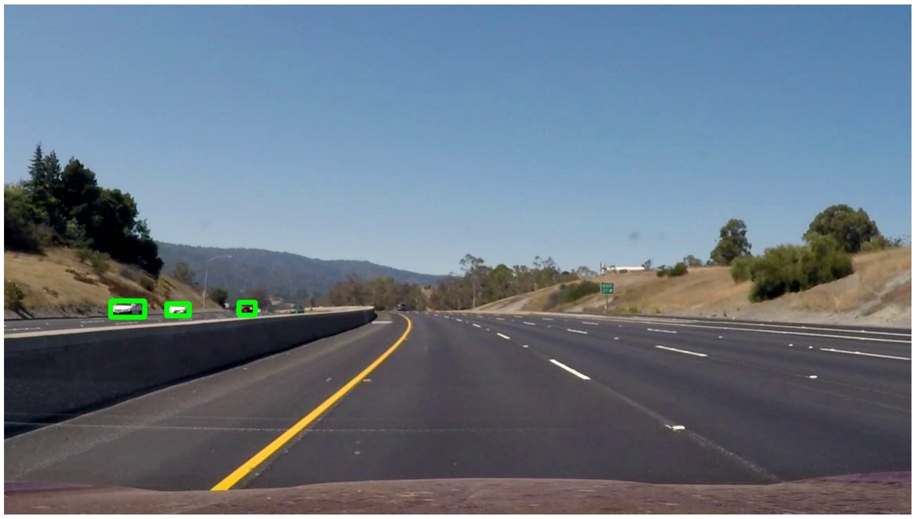
All we need to do is extract the cars from all objects detected in the frame. We consider only objects with class 3 (that is the class of cars) and with score higher than a threshold (we chose a minimum confidence of 10%):

```
idx = np.intersect1d(np.where(classes[0]==3),np.where(scores[0]>0.1))
boxes = boxes[0,idx]
```

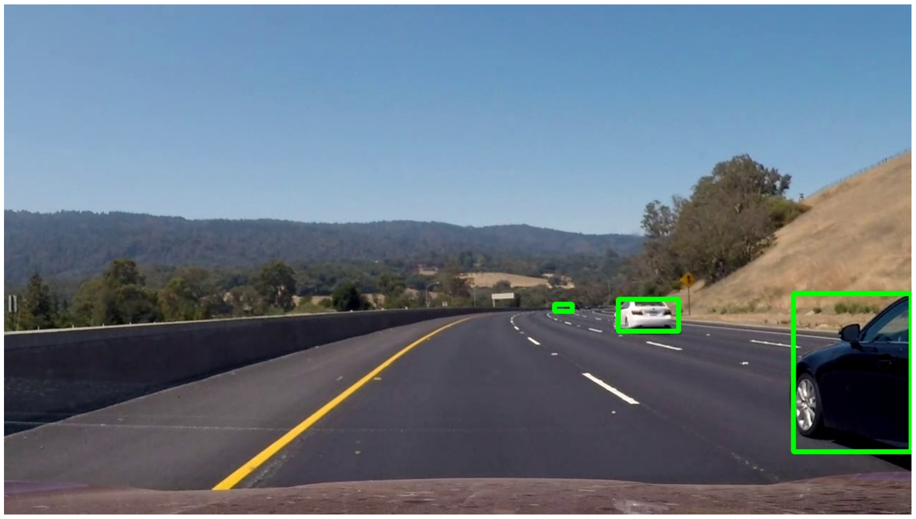
As in the case of SVM we also build a heat map to merge duplicates, then draw the bounding boxes on the image.

The results are quite nice. Although not as fast as we hoped, the sdd-mobilenet surely outperforms the HOG features classification in terms of speed: we get about 1.5 frames per second, which is a 3-fold increase in speed when compared to the already optimized SVM pipeline.

Detection accuracy is also quite good. In particular, bounding boxes are much more stable around the cars, false positives detections are rare, and cars are detected across the whole frame, which the SVM solution did not do because of the much smaller ROI.



Another strength of the CNN solution is that no sliding windows of fixed discretized sized are used in its algorithm, so cars of any size, from very large to very small, can be easily detected.



The final video was saved in the *project_video_CNN.mp4* file.

## Adding line detection

At this point it is relatively straightforward to merge the vehicle detection and the line detection functions on the same frame processing pipeline.

The final video was saved in the *project_video_FINAL.mp4* file.