PROJECT : IMAGE AND SPEECH RECOGNITION

Vehicle Detection with HOG and Linear SVM

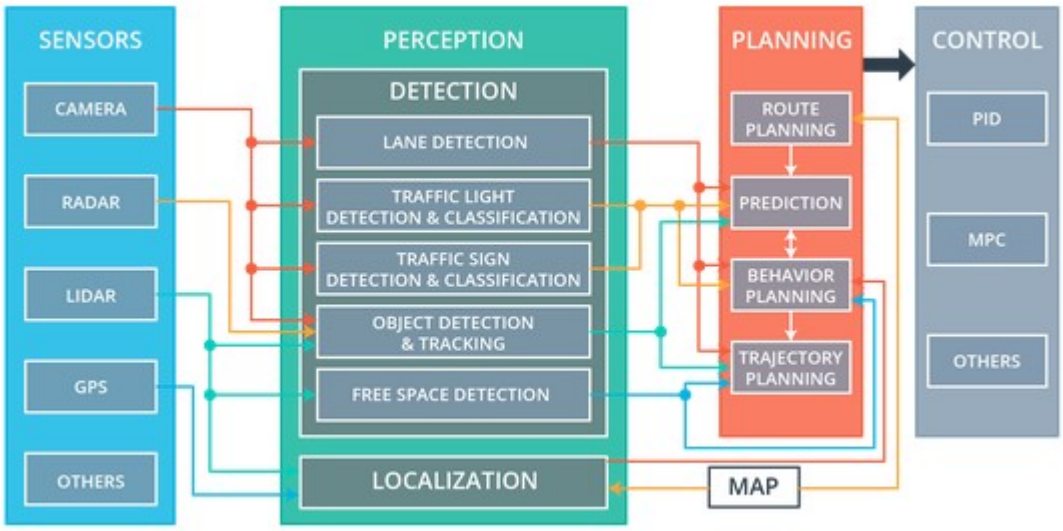
Supervisor : Maciej Stefańczyk

By : Jayant Nehra

Objective

The main objective of the project is to implement a vehicle detection Pipeline for autonomous driving assisting system. Input for the system is an image from the camera attached in front of the user's vehicle. And the goal is to train the system to detect the nearby vehicle with reference to the user's vehicle.

Overall Proposed System



In this first phase of implementation, I am focusing on Monocular Camera Sensors. And in the perception module Lane Detection, object detection and tracking are implemented currently.

Dataset

In the project, used vehicle database provided by Universidad Politécnica de Madrid (UPM), and is available for free for the researchers.

The Image Processing Group is currently researching on the vision-based vehicle classification task. In order to evaluate our methods, we have created a new Database of

images that we have extracted from our video sequences (acquired with a forward looking camera mounted on a vehicle). The database comprises 3425 images of vehicle rears taken from different points of view, and 3900 images extracted from road sequences not containing vehicles. Images are selected to maximize the representativity of the vehicle class, which involves a naturally high variability. In our opinion one important feature affecting the appearance of the vehicle rear is the position of the vehicle relative to the camera. Therefore, the database separates images in four different regions according to the pose: middle/close range in front of the camera, middle/close range in the left, close/middle range in the right, and far range.

In addition, the images are extracted in such a way that they do not perfectly fit the contour of the vehicle in order to make the classifier more robust to offsets in the hypothesis generation stage. Instead, some images contain the vehicle loosely (some background is also included in the image), while others only contain the vehicle partially. Several instances of a same vehicle are included with different bounding hypotheses. The images have 64x64 and are cropped from sequences of 360x256 pixels recorded in highways of Madrid, Brussels and Turin.

Apart from the images of our own collection, we are also using a small set of images of other databases in order to round the number of images to 4000 thousand vehicles images and 4000 non-vehicle images.

The complete set of images is selected so that it covers many different driving conditions, especially relating to weather. In fact, from the 2000 images devoted to each of the image regions (1000 vehicles instances and 1000 non-vehicle images), 20% of them are taken in sunny weather, 20% in cloudy days, 20% in medium conditions (neither very sunny nor cloudy), 20% with poor illumination (down/dusk), 10% with light rain, 5% with bad resolution cameras, and 2,5% in tunnels (with artificial light).

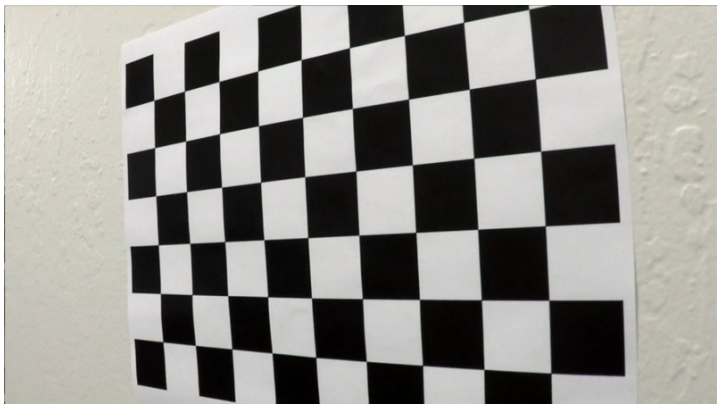
Implementation

Phase 1 Implementation Details : Lane Detection Pipeline (Perception Module)

- Compute the camera calibration matrix and distortion coefficients given a set of chessboard images.
- Apply a distortion correction to raw images.
- Use color transforms, gradients, etc., to create a thresholded binary image.
- Apply a perspective transform to rectify binary image ("birdseye view").
- Detect lane pixels and fit to find the lane boundary.
- Determine the curvature of the lane and vehicle position with respect to center.
- Warp the detected lane boundaries back onto the original image.
- Output visual display of the lane boundaries and numerical estimation of lane curvature and vehicle position.

Camera Calibration

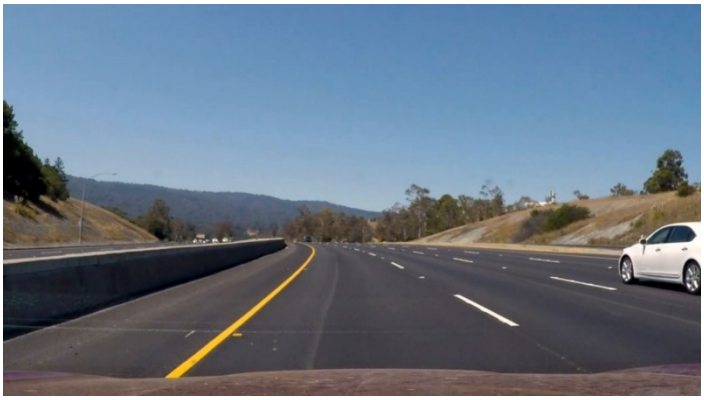
Each image is a view of a chessboard pattern at various angles, as shown below



I use the OpenCV function findChessBoardCorners() to find the x,y coordinates of each corner on a given image. Once I have an array with all these corners, I use the OpenCV calibrateCamera() function to calculate the camera matrix, the vector of distortion coefficients, and the camera vectors of rotation and translation.

Undistort Image

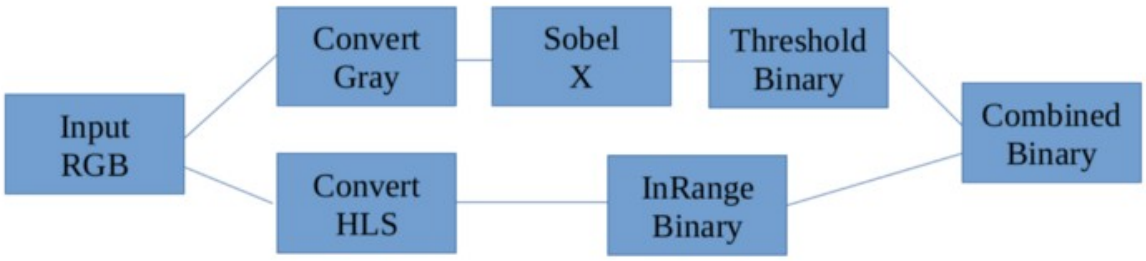
Once we have calibrated the camera as above, we can now apply the camera matrix and distortion coefficients to correct distortion effects on camera input images. This is done using the OpenCV undistort() function.




In the above pictures, if we look near the edges(eg some trees) we can kind of see the effects of this.

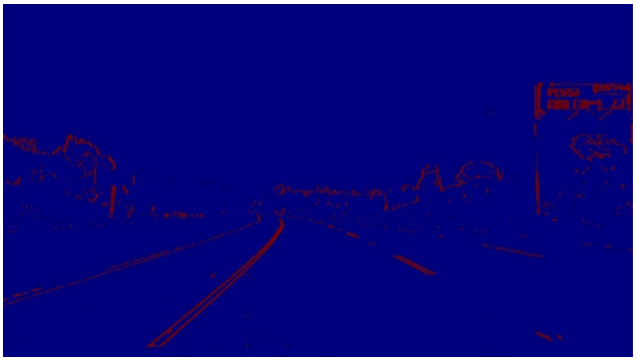
Binary Thresholding

The Thresholding stage is where we process the undistorted image and attempt to mask out which pixels are part of the lanes and remove those that are not.



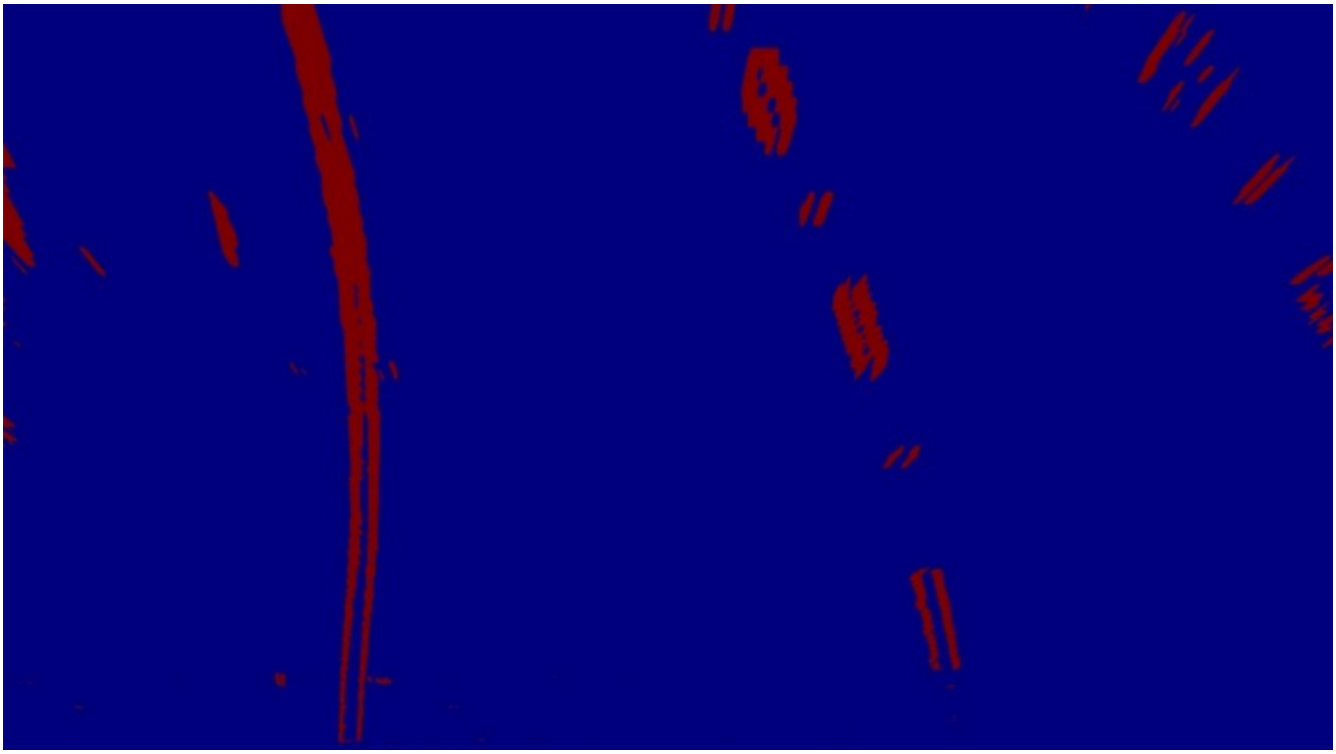
I take the input RGB image, and in the top path, I convert it to grayscale. I then apply a Sobel filter in the X direction to get image edges that match the direction of the lane lines. Then apply a threshold function on this to filter out out pixels that are not of interest. Through experiments, I found that min/max threshold values of 30 and 150 seem to work well. I use this output to generate a binary image of pixels of interest.

On the bottom path, I convert the RGB image to the HLS color space, and then use the S channel from that. The S saturation channel is useful for picking out lane lines under different color and contrast conditions, such as shadows. And pass the S channel into an InRange function, again to filter out pixels that are not of interest. Through experiments, I found values of 175 and 250 to work best here. I also generate a binary image using this output. Finally, I combine both the Threshold Binary and the InRange Binary to generate my final output, the Combined Binary. For reference, it looks like the following



Perspective Transform

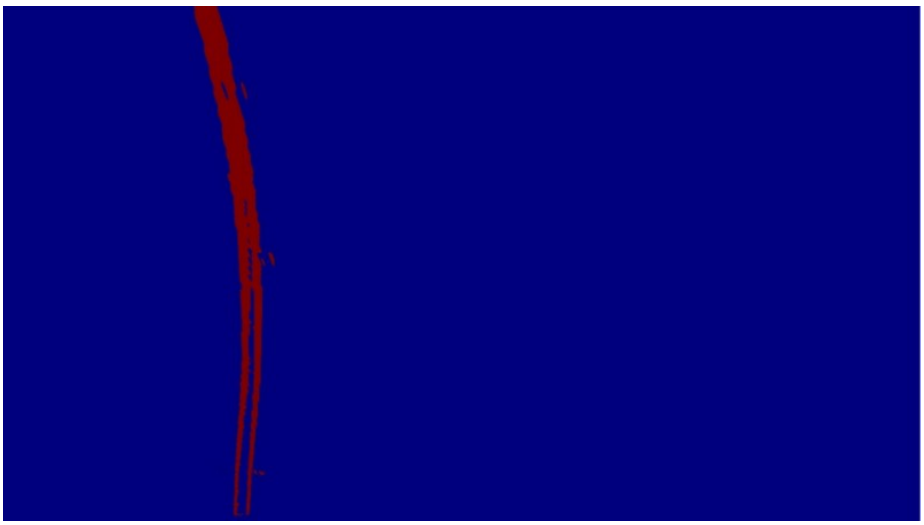
Once I have the binary threshold image above, I apply a perspective transform on the image to generate an image with the effect of looking down on the road from above. (ie : BirdsEyeView using OpenCV warpPerspective() function)



Locating Lanes

This stage is where we will try to extract the actual lane pixels for both the left and right lanes from the above image. For the first frame from a camera, or for cases where we lanes are ‘lost’ (i.e. we have not been able to reliably detect a ‘good’ lane for a number of frames), I generate a histogram of the bottom half of the image.

Then using the two peaks in this histogram, I determine a good starting point to start searching for image pixels at the bottom of the image. Let's call these points x_left and x_right. Once these points are calculated, I divide the image into 10 horizontal strips of equal size. For the bottom strip, I mask out everything outside of a small window around x_left and x_right in order to extract the pixels that belong to the lane, effectively discarded all other 'noise' pixels. I repeat this process for each strip, using histograms on the strips to determine good x_left and x_right values. Once I have processed all strips, I then am left with images for both the left and right lanes. Here's what the left lane image looks like:



Fitting Lanes

Once I have image for both the left and right lanes from above, then I use the Numpy polyfit() function to fit a polynomial equation to each lane. I also calculate the Radius of Curvature for each lane at this stage. Once these have been calculated, I do some checks to determine if the

calculated lane if ‘good’ or not. First I check to see that the Radius of Curvature of the lane is above a minimum threshold. I selected this threshold by looking at the U.S. government specifications for highway curvature. Effectively, this checks to see that the calculated lane is not turning faster than we would expect.

The second check I do is to see that the left_x and right_x coordinates, essentially where the lane intersects the bottom of the image, have not changed too much from frame to frame. (I found that checking for a 15 pixel delta worked well for this check.) Finally, I check that each lanes Radius of Curvature is within 100 times (larger or smaller) than the previous frames values. As the RoC can be very large for vertical lines, I found that check for bounds of 100x seemed to work pretty well.

If any of the above checks fail for a lane, I consider the lane ‘lost’ or undetected for that frame, and I use the values from the previous frame. Again, at 30fps this should be ok for a short number of frames as the lanes will not change too much between successive frames.

If a lane has not been successfully detected for 5 successive frames, then I trigger a full scan for detecting the lanes in the Locate Lanes section.

Draw Lanes

Finally, I draw the detected lanes back on to the undistorted image. We have to do an inverse perspective mapping to do this, as the lanes have been detected in the perspective view. I also annotate the left_x and right_x

coordinates, as well as the Radius of Curvature for both lanes. I calculate a value for how far the car is from the center of the lane and annotate this too.

Object Detection and Tracking (Perception Module)

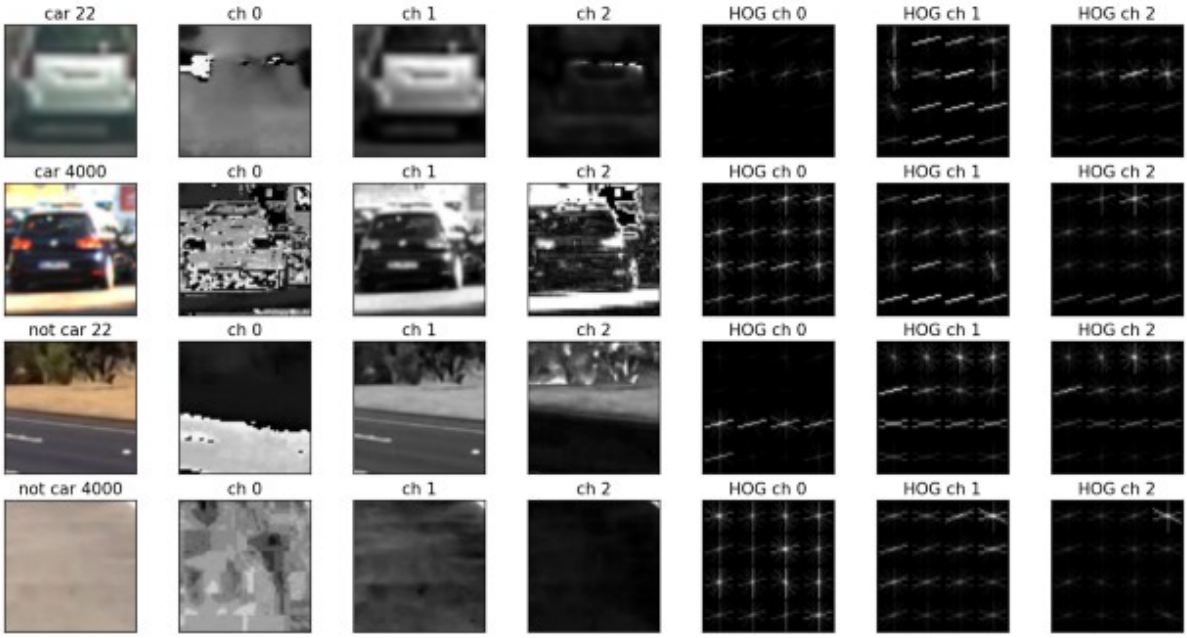
For this purpose i’ve implemented two pipelines using SVM and YOLO. First implementation consists of traditional computer vision techniques, such as Histogram of Oriented Gradients (HOG) and other features combined with sliding windows to track vehicles in a video. Second pipeline is a blazingly fast convolution neural network for object detection.

Feature Extraction

As a feature vector I used a combination of spatial features, which are nothing else but a down sampled copy of the image patch to be checked itself (16x16 pixels) color histogram features that capture the statistical color information of each image patch.

Cars often come in very saturated colors which is captured by this part of the feature vector.

Histogram of oriented gradients (HOG) features, that capture the gradient structure of each image channel and work well under different lighting conditions



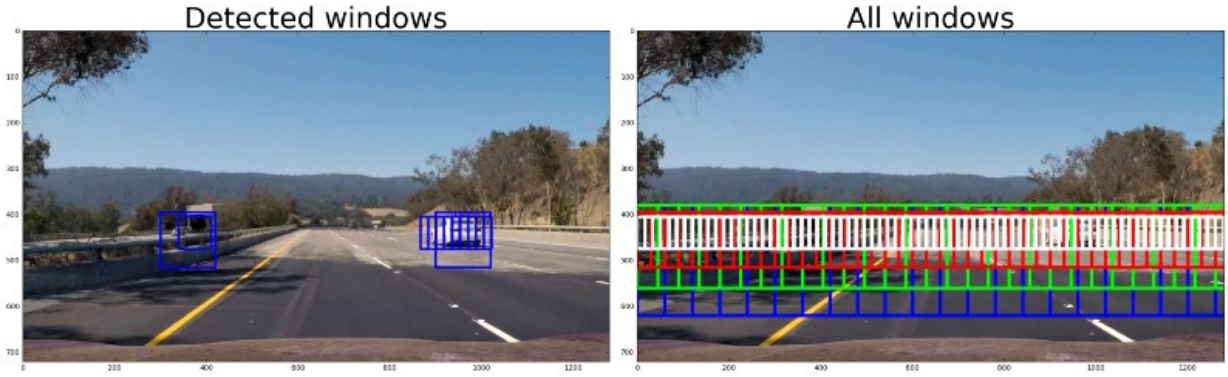
Training a linear support vector machine

Unlike many other classification or detection problems there is a strong real time requirement for detecting cars. So a trade-off between high accuracy and speed is unavoidable. The two main parameters that influence the performance are the length of the feature vector and the algorithm for detecting the vehicles.

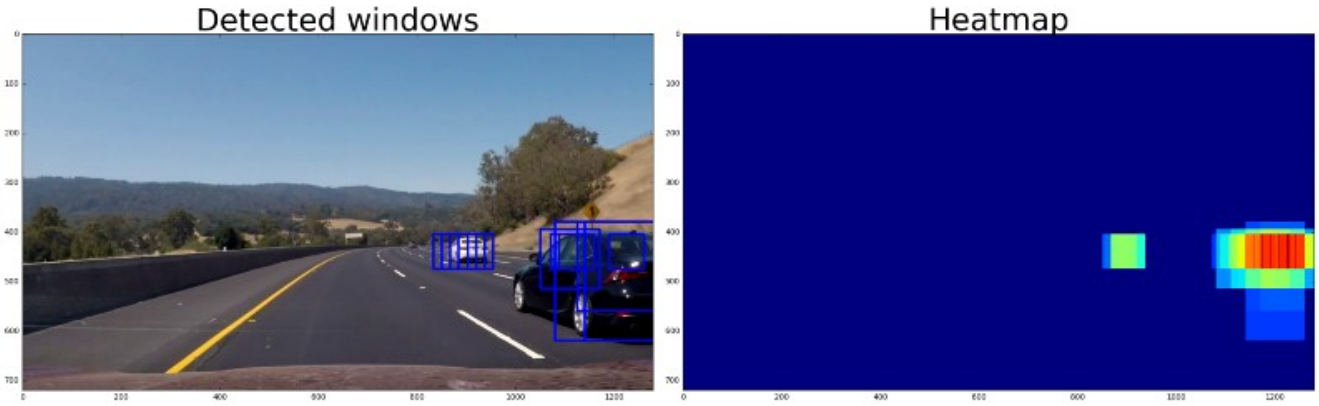
A linear SVM offered the best compromise between speed and accuracy, outperforming random forests (fast, but less accurate) and nonlinear SVMs (rbf kernel, very slow).
The final test accuracy using a feature vector containing 6156 features was above 98.5% which sounds good, but is much less so when you find out that those remaining 1.5% are frequently appearing image patches, particularly lane lines, crash barriers and guard rails.

Sliding Windows

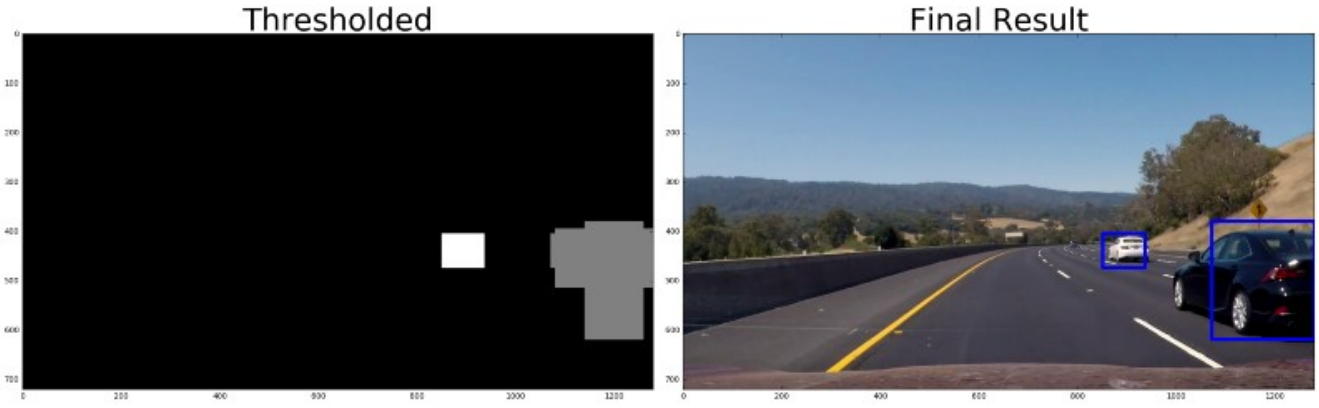
In the standard vehicle detection approach the frames recorded by a video camera the image is scanned using a sliding window. For every window the feature vector is computed and fed into the classifier.
As the cars appear at different distances, it is also necessary to search at several scales. Commonly over a hundred feature vectors need to be extracted and fed into the classifier for every single frame. Fortunately this part can be vectorized (not done). Shown below is a typical example of positive detections together with all ~150 windows that are used for detecting cars. As expected there are some false positives.



For filtering out the false positives, I always kept track of the detected windows of the last 30 frames and only considered those parts of the image as positives where more than 15 detections had been recorded. The result is a heatmap with significantly reduced noise, as shown below :



From the thresholding heatmap the final bounding boxes are determined as the smallest rectangular box that contains all nonzero values of the heatmap :



ALTERNATIVE APPROACH

You Only Look Once

Traditional, computer vision technique based, approaches for object detection systems re purpose classifiers to perform detection. To detect an object, these systems take a classifier for that object and evaluate it at various locations and scales in a test image.

Systems like deformable parts models (DPM) use a sliding window approach where the classifier is run at evenly spaced locations over the entire image. YOLO reframes object detection as a single regression problem, straight from image pixels to bounding box coordinates and class probabilities.

A single convolutional network simultaneously predicts multiple bounding boxes and class probabilities for those boxes. YOLO trains on full images and directly optimizes detection performance.

YOLO Output

YOLO divides the input image into an SxS grid. If the center of an object falls into a grid cell, that grid cell is responsible for detecting that object. Each grid cell predicts B bounding boxes and confidence scores for those boxes. Confidence is defined as (Probability that the grid cell contains an object) multiplied by (Intersection over union of predicted bounding box over the ground truth).
Confidence = Pr(Object) x IOU_truth_pred.

If no object exists in that cell, the confidence scores should be zero. Otherwise we want the confidence score to equal the intersection over union (IOU) between the predicted box and the ground truth.

Bounding Box

Each bounding box consists of 5 predictions: X, y, w, h, confidence

The (x; y) coordinates represent the center of the box relative to the bounds of the grid cell. The width and height are predicted relative to the whole image.

Finally the confidence prediction represents the IOU between the predicted box and any ground truth box.

Each grid cell also predicts C conditional class probabilities, Pr(ClassijObject). These probabilities are conditioned on the grid cell containing an object. We only predict one set of class probabilities per grid cell, regardless of the number of boxes B.

At test time we multiply the conditional class probabilities and the individual box confidence predictions,

Pr(Class|Object) x Pr(Object) x IOU_truth_pred = Pr(Class) x IOU_truth_pred

which gives us classspecific confidence scores for each box. These scores encode both the probability of that class

appearing in the box and how well the predicted box fits the object.
So at test time, the final output vector for each image is a S x S x (B x 5 + C) length vector.

The Model Architecture

The model architecture consists of 9 convolutional layers, followed by 3 fully connected layers. Each convolutional layer is followed by a Leaky RELU activation function, with alpha of 0.1. The first 6 convolutional layers also have a 2x2 max pooling layers.

Implementation : Preprocessing

Input to the model is a batch of 448x448 images. So we first determine the area of interest for each image. We only consider this portion of the image for prediction, since cars won't be present all over the image, just on the roads in the lower portion of the image. Then this cropped image is resized to a 448x448 image.

Normalization : Each image pixel is normalized to have values between 1 and 1. I use simple minmax normalization to achieve this.

Training

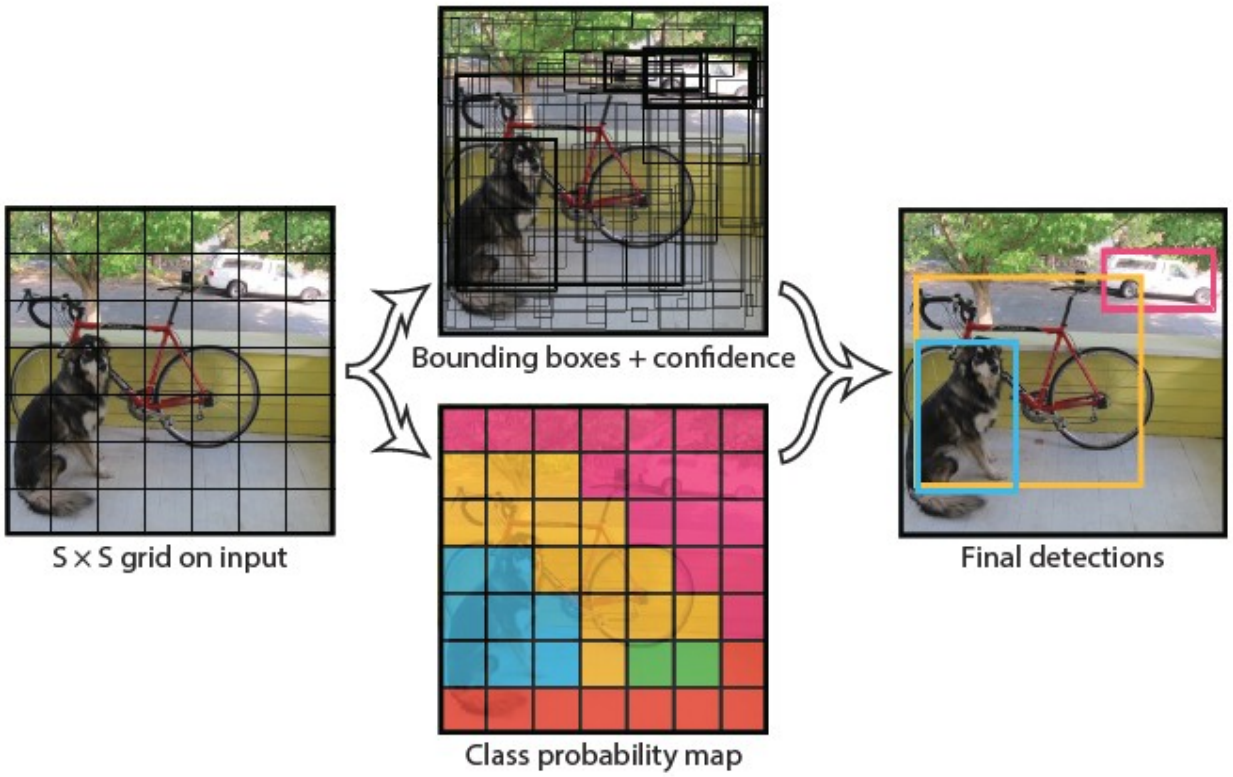
I have used pretrained weights for this project. Training is done in 2 parts

Part 1: Training for classification

This model was trained on ImageNet 1000class classification dataset. For this
we take the first 6 convolutional layers followed by a followed by a fully
connected layer.

Part 2: Training for detection

The model is then converted for detection. This is done by adding 3
convolutional layers and 3 fully connected layers. The modified model is then
trained on PASCAL VOC detection dataset.



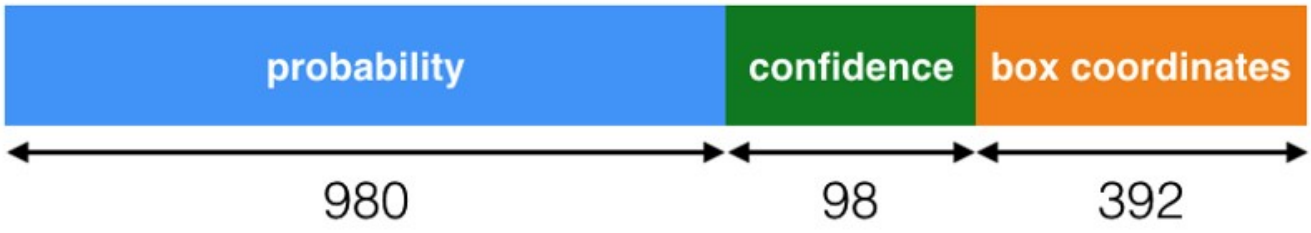
Post Processing

The model was trained on PASCAL VOC dataset. We use S = 7, B = 2. PASCAL VOC has 20 labelled classes so C = 20. So our final prediction, for each input image, is:
output tensor length = S x S x (B x 5 + C)
output tensor length = 7 x 7 x (2x5 + 20)
output tensor length = 1470.

The structure of the 1470 length tensor is as follows:
First 980 values corresponds to probabilities for each of the 20 classes for each grid cell. These probabilities are conditioned on objects being present in each grid cell.

The next 98 values are confidence scores for 2 bounding boxes predicted by each grid cells.
The next 392 values are coordinates (x, y, w, h) for 2 bounding boxes per grid cell.

As you can see in the above image, each input image is divided into an S x S grid and for each grid cell, our model predicts B bounding boxes and C confidence scores.
There is a fair amount of postprocessing involved to arrive at the final bounding boxes based on the model's predictions.

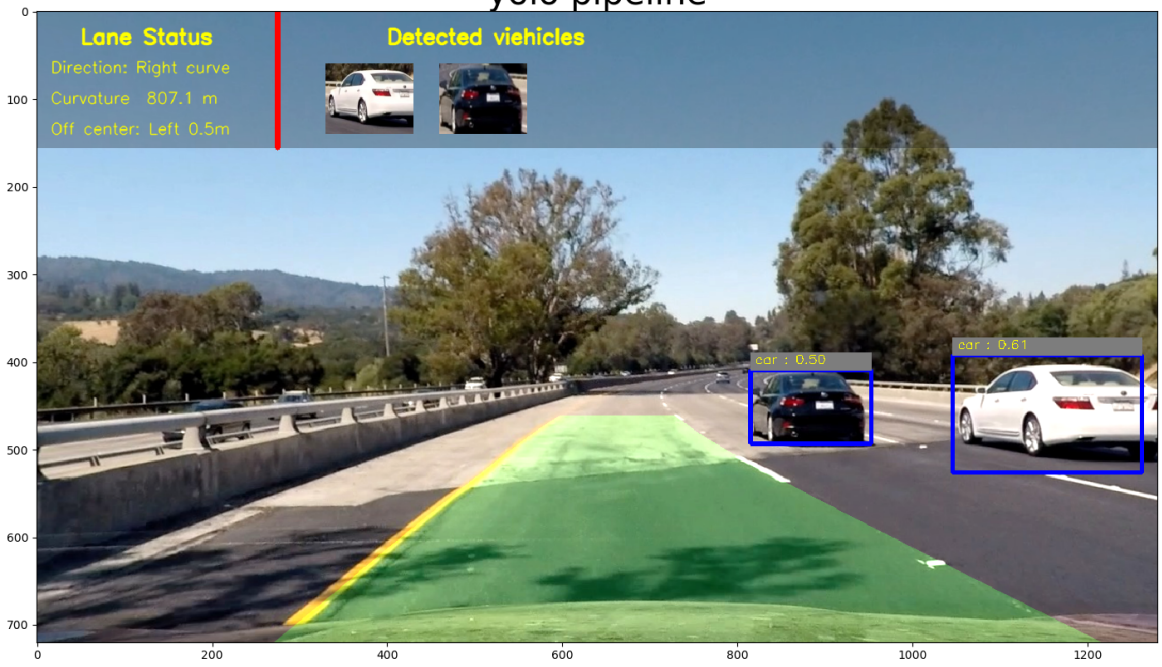


Class score threshold
I reject output from grid cells below a certain threshold (0.2) of class scores (equation 2), computed at test time.

Reject overlapping (duplicate) bounding boxes
If multiple bounding boxes, for each class overlap and have an IOU of more than 0.4 (intersecting area is 40% of union area of boxes), then we keep the box with the highest class score and reject the other box(es).

Drawing the bounding boxes
The predictions (x, y) for each bounding box are relative to the bounds of the grid cell and (w, h) are relative to the whole image. To compute the final bounding box coodinates we have to multiply w & h with the width & height of the portion of the image used as input for the network.

yolo pipeline



Phase 1[Currently Implemented Modules]

- Cameras: Learn the physics of cameras, and how to calibrate, undistort, and transform image perspectives.
- Lane Finding: Study advanced techniques for lane detection with curved roads, adverse weather, and varied lighting.
- Advanced Lane Detection : Detect lane lines in a variety of conditions, including changing road surfaces, curved roads, and variable lighting.
- Use OpenCV to implement camera calibration and transforms, as well as filters, polynomial fits, and splines.
- Support Vector Machines: Implement support vector machines and apply them to image classification.
- Histogram of Oriented Gradients: Implement histogram of oriented gradients and apply it to image classification.
- Deep Neural Networks: Compare the classification performance of support vector machines, decision trees, histogram of oriented gradients, and deep neural networks.
- Vehicle Tracking: Review how to apply image classification techniques to vehicle tracking, along with basic filters to integrate vehicle position over time.

Phase 2 : Sensor Fusion

Phases are broken out into modules, which are in turn comprised of a series of focused sensor implementation.

- Sensors : Sensor Fusion Module covers the physics of two of the most import sensors on an autonomous vehicle — radar and lidar.
- Kalman Filters : Kalman filters are the key mathematical tool for fusing together data. Implement these filters in Python to combine measurements from a single sensor over time.
- Extended Kalman filters are used by autonomous vehicle engineers to combine measurements from multiple sensors into a nonlinear model. Building an EKF is an impressive skill to show an employer.
- Unscented Kalman Filter : The Unscented Kalman filter is a mathematically sophisticated approach for combining sensor data. The UKF performs better than the EKF in many situations. This is the type of project sensor fusion engineers have to build for real selfdriving cars.
- Pedestrian Tracking : Fuse noisy lidar and radar data together to track a pedestrian.