NEW YORK UNIVERSITY

CUSP
CENTER FOR URBAN SCIENCE+PROGRESS

Vehicle Detection for Cities

Machine Learning for Cities

Project Report

Spring 2017

Akshay Penmatcha

Priyanshi Singh

Sunny Kulkarni

1. Introduction:

The task of detecting objects in our physical environment is an interesting one which has a lot of applications in the urban context. New York City's Department of Transportation publishes its traffic camera feeds as part of its Open Data Initiative. The potential applications of building a Vehicle detection model on these cameras could be multifold. It can help us in detecting and tracking congestions is real-time and in turn help in efficient traffic management and planning. Another potential application that is currently being researched at NYU Center for Urban Science and Progress is a Pedestrian tracking and counting using these camera feeds.

Researchers have taken different approaches to solve this for various types of physical objects using the concepts of Image Processing, Machine Learning and Deep Learning. (Dalal & Triggs, 2005) defined a feature descriptor called HOG and used an SVM to detect the presence of a Human in an image. (Viola & Jones, 2004) defined HAAR features motivated to devise a way for recognizing faces. Although it is also used widely for various other object detection tasks. In recent times the use of Convolutional Neural Networks has become widespread when the problem scope includes multiple classes of objects which are complex and when there is a need for reinforcement learning. Training a Neural Network is computationally challenging and to address this (Girshick, Donahue, Darrell, Berkeley, & Malik, 2012) devised a Region based CNN which drastically reduces the computational load because it performs the computations only on a specific region of interest. Later (Girshick, n.d.) and (Ren, He, Girshick, & Sun, 2015) provided enhancements to the R-CNN framework called Fast R-CNN and Faster R-CNN which further improved the computational performance of the CNN framework.

2. Approach:

The task of Vehicle Detection, which is primarily an Object detection problem using Computer Vision techniques could be solved using two approaches. The Shallow Machine Learning techniques and the Deep Learning techniques. The primary difference between the two approaches stems from the way they extract/learn the features in an image. The Shallow Learning methods use Feature Extraction which they input to a classifier to learn the presence of an object. Whereas the portion of Feature Extraction in a Deep Learning method is inbuilt with the first few layers. [6]

During this project we made an attempt to explore the deep learning approach, but due the complexities involved in designing a network and training through back propagation we narrowed down our scope to attempting the novel approaches within the realm of shallow learning techniques. Summarized below are the two different approaches that we have pursued.

I. Object Detection using HOG Features and SVM Classifier. [1]
II. Object Detection using HAAR Wavelets and Cascade Classifier. [5]

3. Datasets Explored:

The primary need for data collection in this project was to find a reliable set of positive training images which contain the object of interest and a set of negative images which contain anything other than the object of interest. Although this process is very similar for both the approaches that we pursued there is a difference in terms of the consistency in images sizes that we use for training. In case of HOG-SVM approach both the positive and negative images need to be in the same dimension. And in case of HAAR-Cascade approach the positive images need to be smaller than the negative images.

The following are the data sources that we explored for the purpose of training the algorithm.

1) ImageNet [7]
2) UIUC Image database for Car Detection [8]

4. Feature Extraction and Classification

A) HOG Features - SVM Classifier

An image is an array of data containing the information about the pixels. To detect the presence of an object with in an image a machine learning classifier needs to understand the localized features of the object rather than just the pixel intensities. Histogram of Oriented Gradients(HOG) is a feature extraction technique which is known to provide excellent performance relative to other feature descriptors [1] available out there.

Histogram of Oriented Gradients (HOG) Features: As (Dalal & Triggs, 2005) defined the

idea behind HOG is "*the local object appearance and shape can be characterized by the distribution of local intensity gradients or edge directions, even without precise knowledge of the corresponding gradient or edge positions." [1]*

Implementation of HOG: As (Dalal & Triggs, 2005) mentioned that implementation of HOG features is accomplished by calculating the aggregates of a 1-D histograms within a cell (group of pixels) and then to account for any illumination differences it within a normalized within a local block of cells. A visual representation of the HOG features is depicted in Fig-1 and Fig-2.

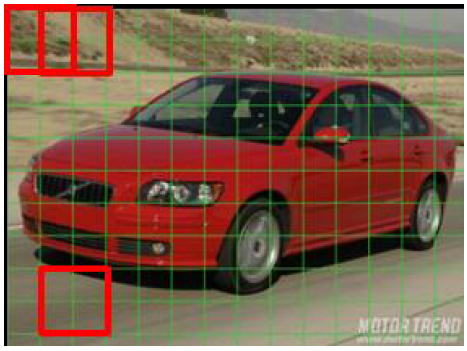


Fig-1: Representation of cells, blocks, stride [7]

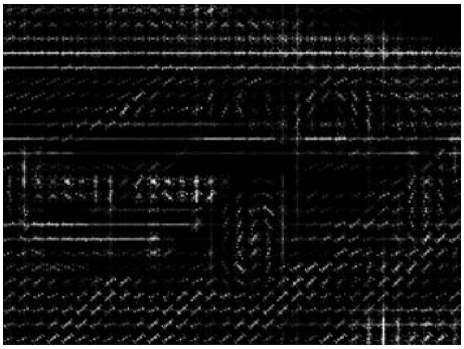


Fig-2: Visual representation of HOG Features

The implementation of HOG can be accomplished in python using scikit-image and OpenCV libraries. The implementation of the same has been demonstrated below.

In scikit-image,

```
fd, hog_image = hog(image, orientations=6, pixels_per_cell=(10, 10),
                    cells_per_block=(2, 2), visualise=True)
```

The primary inputs in this case are the cell size, block size and orientations. The output is the form of a tuple where ***fd*** is a flattened vector and ***hog_image*** is the visual representation of the hog features

In OpenCV,

```
hog = cv2.HOGDescriptor(winSize,blockSize,blockStride,cellSize,nbins,
                        derivAperture,winSigma,histogramNormType,
                        L2HysThreshold,gammaCorrection,nlevels)
```

The primary inputs are the same as scikit-image but having different representations. The

output is this case is just a flattened vector.

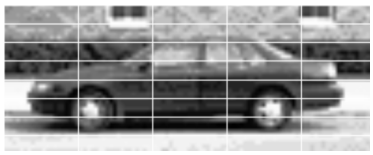
Dimensions of the flattened vector = No: of Blocks * No: of cell per block * No of Bins

Process Flow for the HOG-SVM Object Detection Approach:

This approach is adapted from (Dalal & Triggs, 2005), pyimagesearch[I] and various other blogs on the internet.

Step#1 - Labelled Dataset:

In this approach, we've taken a labelled dataset of 1000 images which containing 500 positive images which have a car and 500 negative images which do not have a car.



Positive: 500 [8]

Negative: 500 [8]

Fig-2: Positive and Negative Training Images

Step#2 - HOG Feature Extraction:

As mentioned previously, we later extracted the HOG feature vectors for these images and subsequently labelled them.

Step#3 - Training the SVM:

Next, we staged our labelled data and split them into a training and testing sets. Then we passed on the data to a Linear Support Vector Machine for training. Later we use our test data to get the confusion matrix. During this process, we found that by varying the parameter 'C' in the SVM input the accuracy of prediction changed. The inflection point for this change was found to be at C=0.1. Below are the results for the confusion matrix for C value 0.1 and 1.

Step#4 - Image Pyramids:

Image Pyramids are used to solve the problem of object scale. When we train our SVM and test on images of the same size, we do not encounter this problem. But when we try to test the Classifier on a larger image where multiple cars of different sizes are present the scale of the object differs throughout the image. In order to tackle this Image Pyramids provide a way to create a series of scaled down images through which we can parse our sliding window.

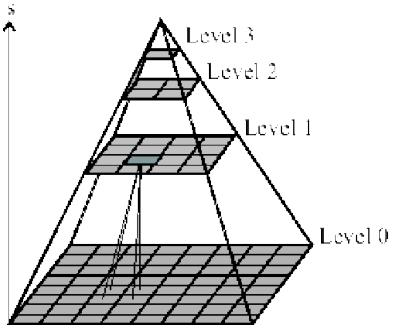


Fig-3: Image Pyramids [12]

Step#5 - Sliding Window:

The length of HOG feature vector that we get as an output is a function of our parameters which remains constant for an image of the same size with the same parameters. But since our real-world test images can be of different sizes we use a sliding window of the same size as our training images to capture the HOG features as it slides across the picture. We then capture all the coordinates of window's where the presence of a vehicle was detected.

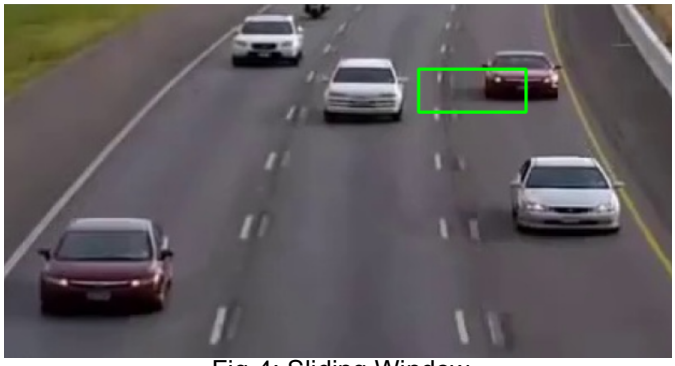


Fig-4: Sliding Window

Step#6 - Non-Maximum Suppression:

As the window slides across the image with a stride there is a possibility that the same object can get detected subsequently across multiple windows. To solve this, we use the Non-Maximum Suppression technique which allows us to filter out the redundant windows and keep only the unique ones detecting the vehicle in our case.

Results from HOG-SVM Approach:

Using a Train-Test split of 0.2 (20%) within our training set, the following are the results that we obtained for different C values in the Linear SVM Classifier.

Confusion Matrix	Predicted: NO	Predicted: YES
Actual: NO	99	1
Actual: YES	1	99

Fig-6: Confusion matix for C=0.1

Confusion Matrix	Predicted: NO	Predicted: YES
Actual: NO	97	2
Actual: YES	3	98

Fig-7: Confusion matix for C=1

Although the above results show a pretty high accuracy. A completely different test image like the one below has yielded very poor results. As we investigated the dataset for possible reasons, we found that the training set is highly skewed with vehicle images from their side view, unlike the image below.

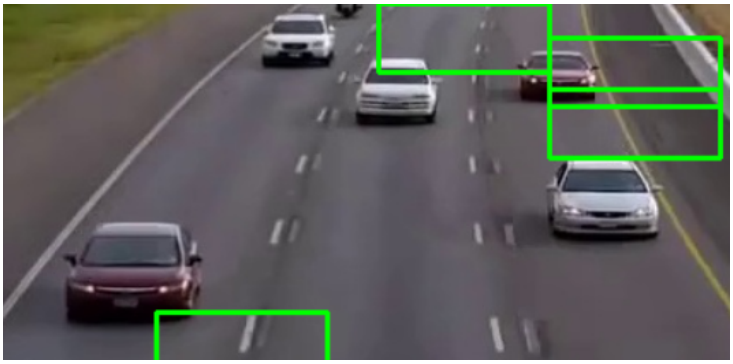


Fig-8: Predicted Cars in an unseen New

Hard Negative Mining: To solve the above problem, where our prediction accuracy goes down when we test our classifier on a completely different type image can be solved using the Hard-negative Mining technique. Where we record the sample of a few wrongly classified windows and pass them back to our SVM to retrain the model. Prediction accuracy is known to show a significant increase in the first run of hard-negative mining. Although this work is still in process the above problems were well addressed in the next approach that we have taken using the HAAR features and Cascade Classifier.

B) HAAR Features - Cascade Classifier Approach

Haar Cascade Classifier, often referred as Viola Jones object detection framework, is the first object detection algorithm which works very well in real time and was proposed by Paul Viola and Michael Jones in 2001. This object detector approach makes use of Haar like features which are digital image features traversed on images to extract features from the images. They owe their name to their visceral similarity with Haar Wavelet and were used in first real time face detector.

Haar Cascade classifier is a Machine Learning approach wherein a Cascade function (Cascade of boosted classifiers) is trained from a lot of Positive and Negative images, making use of Adaboost algorithm to select the best features from training data [1]. These features are subcategorized and then passed to the classifiers in different stages to reach the final stage where the object is detected. Basically, the algorithm has four stages:

1. Haar Feature Selection (Viola et. al., 2003)
2. Creating an Integral Image
3. Adaboost Training
4. Cascading Classifiers.

Step#1 - HAAR Feature Selection:

- The first step is to collect a lot of positive Images (images with car) and negatives images (images without car) to train the classifier.
- The second step is to extract features from these images. For this Haar like features are used. These are basically the convolution kernels, which may have Edge features, Line Features, Four- rectangle features, center surround features and 60 other such type of features. [2]

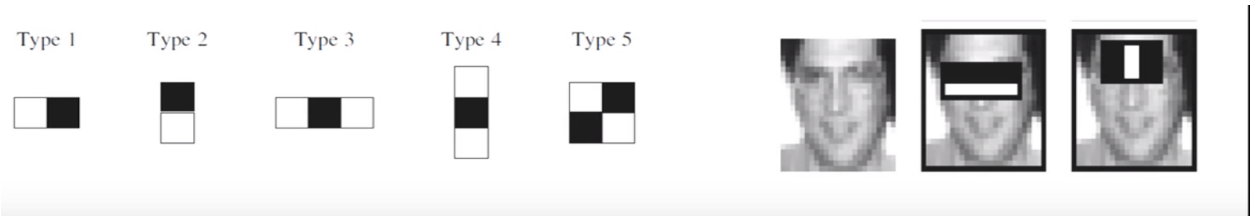


Fig-9: Haar Features applying on given image (source:

- Basically, a Haar Like feature considers adjacent rectangular regions at a specific location in a detection window, sums up the pixel intensities in each region and

calculates the difference between the sums. The difference is then used to categorize the subsection of the image, ending up with one value per feature.

- A plenty number of features are obtained by varying the sizes and locations of kernel (around 16,000 plus features for 24 x 24-pixel window)

But calculating these features by summing the pixels under black rectangle and white rectangle is a computationally heavy task, which is why the concept of Integral Image was introduced.

Step#2 - Integral Image:

- Creating an Integral Image is a method which saves greater amount of computational time by involving just four pixels rather than calculating sum of pixels under black and white rectangles.
- So, to calculate the value of pixel (x,y), sum of pixels above and to the left of (x,y) is taken.

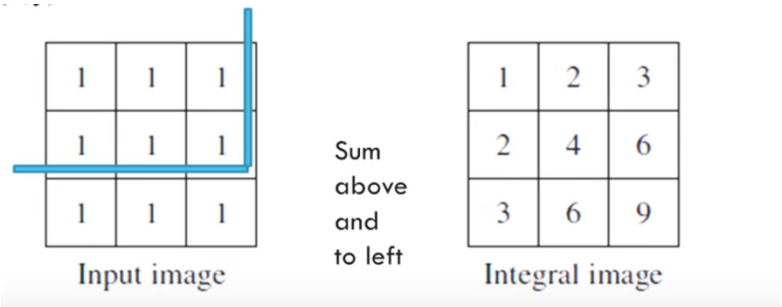


Fig-10: Integral Image [10]

Sum of all pixels under any rectangle is calculated by taking only four values at the corner of rectangle. So, for the figure below, the number of pixel under rectangle D will be calculated as: (Viola & Jones, 2004)[5]

Sum of all pixels in D = A+ (A+B+C+D) - (A+C-A+B)

= 1 + 4 - (2 + 3)

= D

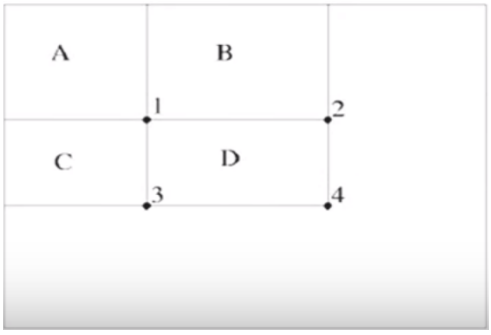


Fig-11: Integral Image [10]

The advantage of Haar features over other features is its calculation speed, which is achieved by applying integral image approach, any HAAR like features of varying size can be calculated in a constant period of time. So, this method cuts down the computation time of calculating pixels which makes the overall feature extraction fast. [9]

Step#3 - Adaboost Training

- The object detection algorithm incorporates a form of learning algorithm **AdaBoost (Adaptive Boosting),** to select the best features for further training.
- Adaboost is an algorithm that builds a strong classifier as a linear combination of weighted simple “weak” classifiers and helps finding the best features out of those 16,000 features. Here, the term “weak” implies that it alone can’t classify the image, but together with others forms a strong classifier.
- After these features are found, a weighted linear combination of these features is taken to detect an object in the window. If a feature can perform better than the random guess, it is taken to the next stage [10].

$$F(x) = \alpha_1 f_1(x) + \alpha_2 f_2(x) + \alpha_3 f_3(x) + \ldots$$

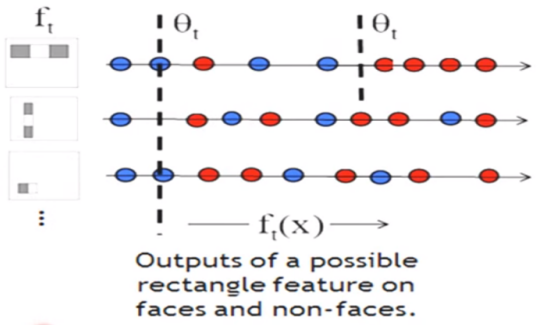
Strong classifier Weak classifier

Fig-12: Linear combination of Weak classifier

- Adaboost finds the single rectangular feature and threshold that best separates the positive and negative training samples.
- Every feature is applied to all the images, positive and negative both, and a threshold is given to find the error rate for all the features (weak classifier). The feature with minimum error rate is picked and then all the features are reweighted according to the below mentioned criteria:

  Incorrectly Classified: more weight

  Correctly Classified: less weight



Outputs of a possible rectangle feature on faces and non-faces.

Fig-13: Feature selection using Adaboost

- Final classifier is taken as a combination of weak classifiers, weighted according to the error they had by iterating the same process for some T (defined by user) rounds.

According to the paper (Viola & Jones, 2004)[5], even 200 features provide detection with 95% accuracy and their final features had around 6000 features (cutting down from 16,000). It is a big gain to achieve faster computation when dealing with big number of features, which is generally the case.

Step#4 - Cascading:

Since applying a big number of features to a single strong classifier takes high computation cost and time as well, the concept of Cascade of Classifiers was introduced. This process implements by not applying all the features on a window, instead features are grouped in to different stages of classifiers and apply one by one (normally few features in initial stages) [9]. If a window fails the first stage to detect features, it is discarded and not taken to the second stage. If a window passes the pattern of features, it is passed to the second stage of features and continue the process. Finally, the window which passes all the stages, detects the object.

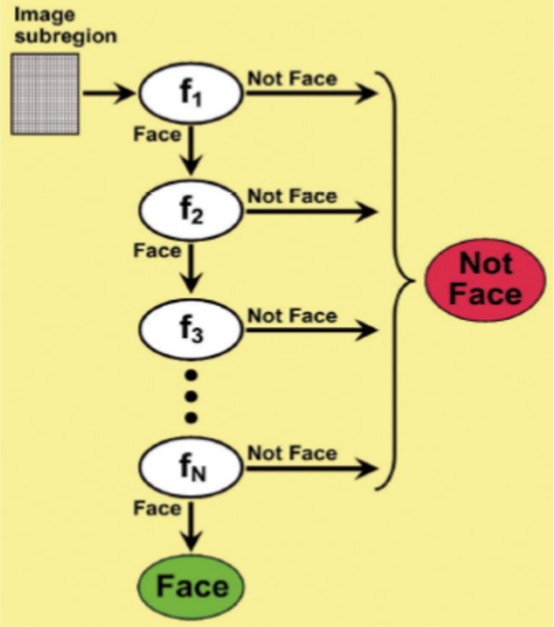


Fig-14: How object detection works [9]

Implementation of HAAR Cascades:

Pre-requisites:

1) Python version 2.7 or higher
2) CMake library and it's GUI version 3.8.1.
3) OpenCV version 3.1
4) OpenCV & CMake Shared and Static Libraries.

Once OpenCV is installed, verify the package is loading successfully in python using command:

```
import cv2
```

Steps to implement the HAAR Feature Extraction and execution of Cascade Classifier:

1) Select a video feed showing cars approaching a highway in a straight line.
2) In the video frame, select one car that is generic and has the orientation as other cars in the video frames.
3) Using online resource such as Image-Net, search for images that do not have vehicles in them. This set would be the Negative Image Set. For our example, we can use images with say “People” in it. The Image-Net search returns a list of web links to images with people in it.
4) Applying python, we download this list of Negative Image Set to our local machine.
5) During download, convert the images to Grayscale using function:

   ```
   cv2.IMREAD_GRAYSCALE()
   ```

   This is because HAAR-like features only have black and white images colors in the features.
6) This image set requires a cleansing step. Some images have blank display. We copy one such image into a folder named “Incorrect Img”. Using below function to read the image:

   ```
   cv2.imread()
   ```
7) Traverse through all the Negative image set and compare the images using above function. If a match is found, remove them.
8) Next, create a text file (neg_bg.txt) that contains the path to all images files in the Negative image set folder.

Once the above steps are executed, we have to run the below listed command line functions. These are the executable files from the OpenCV and CMake installation.

1. **Function 1: opencv_createsamples( )** - This function is used to prepare the dataset for the positive and test image set. This function is executed two times - once to create the positive training image set and second time to create the vector file with the first output. This execution is controlled by the input parameters passed to the function.
   a) Step 1: **Create Positive Training Image Set** - As input parameters, it accepts the positive image (imgcar.jpg here) and the neg_bt.txt file (created in step h.) which

contains path to all Negative Images. It superimposes the positive image on the negative images at varying angle of 0.5 radians along the x, y and z axes. The output is 1430 superimposed positive training image set denoted by parameter -num here.

```
opencv_createsamples -img imgcar.jpg -bg bg.txt -info
info/info.lst -pngoutput info -maxxangle 0.5 -maxyangle 0.5 -
maxzangle 0.5 -num 1430
```

Parameters:

-img	input path to positive image (imgcar.jpg here)
-bg	input path to neg_bg.txt file
-info	output path to create the info.lst file. It contains the file path info for all the positive training image set.
-pngoutput	output path to create the superimposed positive training image set
-axxangle -axyangle -axzangle	0.5 radians - Image rotation while superimposing it on the negative image set.
-num	1430 - how many positive training image set to be created.

b. **Step 2: Create Positive Image Set Vector file:** CreateSamples() function takes the positive image training set as input and generates a vector file. This file contains the information from the positive image and the superimposed positive image training set.

```
opencv_createsamples -info info/info.lst -num 1430 -w 20 -h 20 -
vec positives.vec
```

Parameters:

-info	input path to the info.lst file created in Step 1.
-num	number of input files to create the vector file.
-vec	output path to create the vector file for the positive training image set.

-w & -h	width and height (in pixels) for the output and stored to the vector file. This value should also be used as parameter for the TrainCascascade() function explained below.

2. **Function 2: opencv_traincascade( )** - Once the vector file is created by the CreateSamples() function, it is used as input to train on the positive training image set. This function runs in stages and each stage includes a subset of features. If a window fails the first stage to detect features, it is discarded, otherwise it is passed to the next stage. If a window has passed all the stages, it is concluded to match the pattern in features and hence the object is detected.

```
opencv_traincascade -data data -vec positives.vec -bg bg.txt -numPos
1200 -numNeg 600 -numStages 10 -w 20 -h 20
```

Parameters:

-data	output path to store the trained classifier
-vec	input path to vector file created by CreateSamples( ) function
-bg	input path the the neg_bg.txt file created for negative image set
-numPos & -numNeg	number of positive and training samples to be used in training of every stage out of the total positive training image set specified in the vector file.
-numStages	number of cascade stages to be trained
-w & -h	size of samples (in pixels). Should have same value used in CreateSamples( ) function i.e. 20 x 20 pixels

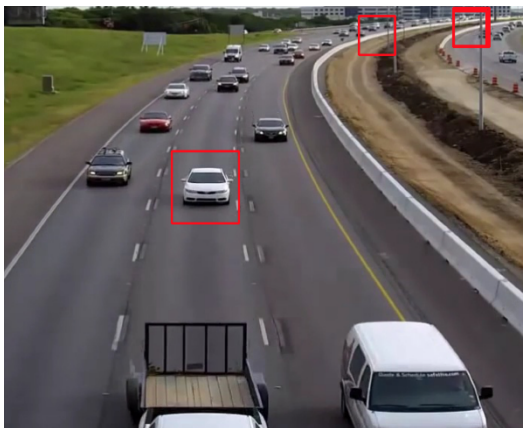
TrainingCascade( ) Function Stages Output:

```
===== TRAINING 1-stage =====
<BEGIN
POS count : consumed   1380 : 1384
NEG count : acceptanceRatio    900 : 0.347222
Precalculation time: 6
+----+---------+---------+
|  N |    HR   |    FA   |
+----+---------+---------+
|   1|        1|        1|
+----+---------+---------+
|   2|        1|        1|
+----+---------+---------+
|   3|        1|        1|
+----+---------+---------+
|   4| 0.998551| 0.593333|
+----+---------+---------+
|   5| 0.995652| 0.258889|
+----+---------+---------+
END>
Training until now has taken 0 days 0 hours 0 minutes 56 seconds.

===== TRAINING 2-stage =====
<BEGIN
POS count : consumed   1380 : 1390
NEG count : acceptanceRatio    900 : 0.0803715
Precalculation time: 5
+----+---------+---------+
|  N |    HR   |    FA   |
+----+---------+---------+
|   1|        1|        1|
+----+---------+---------+
|   2|        1|        1|
+----+---------+---------+
|   3| 0.995652| 0.788889|
+----+---------+---------+
|   4| 0.995652| 0.788889|
+----+---------+---------+
|   5| 0.998551| 0.634444|
+----+---------+---------+
|   6| 0.995652| 0.603333|
+----+---------+---------+
|   7| 0.995652|     0.41|
+----+---------+---------+
END>
```

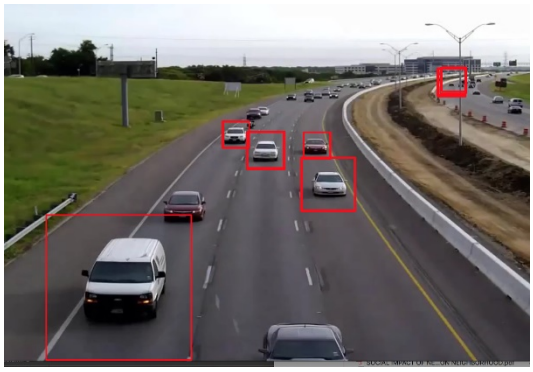
```
===== TRAINING 3-stage =====
<BEGIN
POS count : consumed   1380 : 1396
NEG count : acceptanceRatio    900 : 0.0335771
Precalculation time: 5
+----+---------+---------+
|  N |    HR   |    FA   |
+----+---------+---------+
|   1|        1|        1|
+----+---------+---------+
|   2|        1|        1|
+----+---------+---------+
|   3| 0.997101| 0.757778|
+----+---------+---------+
|   4| 0.997101| 0.757778|
+----+---------+---------+
|   5| 0.995652|     0.65|
+----+---------+---------+
|   6| 0.995652| 0.567778|
+----+---------+---------+
|   7| 0.995652| 0.416667|
+----+---------+---------+
END>
Training until now has taken 0 days 0 hours 2 minutes 5 seconds.

===== TRAINING 4-stage =====
<BEGIN
POS count : consumed   1380 : 1402
NEG count : acceptanceRatio    900 : 0.0139304
Precalculation time: 5
+----+---------+---------+
|  N |    HR   |    FA   |
+----+---------+---------+
|   1|        1|        1|
+----+---------+---------+
|   2|        1|        1|
+----+---------+---------+
|   3| 0.995652| 0.793333|
+----+---------+---------+
|   4| 0.998551| 0.771111|
+----+---------+---------+
|   5| 0.996377| 0.698889|
+----+---------+---------+
|   6| 0.995652|     0.64|
+----+---------+---------+
|   7| 0.995652| 0.426667|
+----+---------+---------+
END>
Training until now has taken 0 days 0 hours 2 minutes 40 seconds.
```

Insights:

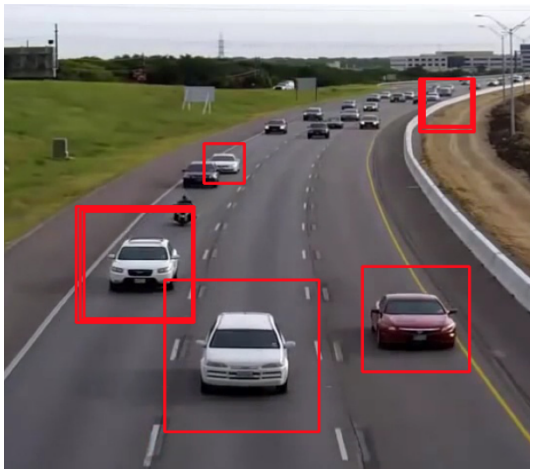
- HR = Hit Ratio => True Positive - Measure to indicate that algorithm returns “Yes” when vehicle is actually present.
- FA = False Alarm => False Positive - Measure to indicate that algorithm returns “Yes” when vehicle is missing in the image.
- Training Stages seen here are 1, 2 and 6. As the stages proceed, the acceptanceRatio, a measure for accuracy of detection improves from 0.34 to 0.008
- The output for the TrainCascade( ) function is a Cascade.xml file and stage.xml file for each Training stage. The Cascade xml file is a combination of the individual stage xml files for the listed number of stages given during function execution.
- This xml file is now the only input to our cv2.CascadeClassifier( ) function used in the Python code. We passed a range of images and also a video feed as input to detect the vehicles and below are the results seen



The above white vehicle image seen in the red box is the positive image used to train the Classifier.



Seen here, the Classifier is not only able to detect vehicles of sedan type with white color but also large size vehicles and some with different colors also.



Seen here, for this image file, the vehicle at the left side is detected multiple times. We would need to apply the Non Maximum Suppression Technique here, discussed in the HOG section to eliminate redundancies and only have a single detection seen for most other images.

We also observed that this problem was faced when a set of images are passed to the Classifier and not when video feed is given as input.

For this exercise, we have used a single positive image file as a reference for the Cascade Classifier and received very good results. To achieve stronger accuracy, the

HAAR Cascade Classifier should be trained on a range of positive images. This would require higher computation power to train on a variety of vehicle models with the orientations at different angles to improve accuracy.

To see implementation refer to the below GitHub Repository –

https://github.com/akpen/ml_project

5. Conclusion and Way Forward:

The understanding of HOG and HAAR Image Processing techniques enabled us to understand how the objects are detected behind the libraries such as OpenCV and SKimage. The capabilities of SVM and Adaboost Classifier techniques are aligned with HOG and HAAR techniques thereby providing accurate results.

As seen in HAAR-Cascade Classification implementation, a single positive image is able to detect vehicles both in images and video feed with considerable accuracy.

As we have seen in the above implementation

Way Forward -

This project has given a good understanding of the underlying methodology applied towards object detection. Future scope of the project can be -

a. Count vehicles from traffic cameras to understand road congestion.
b. Classify vehicles into small, medium and large size such as for sedan and trucks.
c. HOG - SVM model can be further improved using hard mining
d. Apply Convolutional Neural Network techniques to improve accuracy and compare results with HOG - SVM and HAAR - Cascading Classifiers.

6. Bibliography/References:

[1] Dalal, N., & Triggs, B. (2005). Histograms of oriented gradients for human detection. *Proceedings - 2005 IEEE Computer Society Conference on Computer Vision and Pattern Recognition, CVPR 2005*, *I*, 886–893. https://doi.org/10.1109/CVPR.2005.177

[2] Girshick, R. (n.d.). Girshick_Fast_R-CNN_ICCV_2015_paper, 1440–1448. https://doi.org/10.1109/iccv.2015.169

[3] Girshick, R., Donahue, J., Darrell, T., Berkeley, U. C., & Malik, J. (2012). Girshick_Rich_Feature_Hierarchies_2014_CVPR_paper, 2–9. https://doi.org/10.1109/CVPR.2014.81

[4] Ren, S., He, K., Girshick, R., & Sun, J. (2015). Faster R-CNN: Towards Real-Time Object Detection with Region Proposal Networks. *Nips*, 1–10. https://doi.org/10.1016/j.nima.2015.05.028

[5] Viola, P., & Jones, M. J. (2004). Robust Real-Time Face Detection. *International Journal of Computer Vision*, *57*(2), 137–154. https://doi.org/10.1023/B:VISI.0000013087.49260.fb

[6] Yann Le Cun

[7] ImageNet

[8] UIUC Image database for Car Detection

[9] Opencv tutorial for face detection
http://docs.opencv.org/trunk/d7/d8b/tutorial_py_face_detection.html

[10]https://www.youtube.com/watch?v=WfdYYNamHZ8

[11] Creating your own Haar Cascade OpenCV Python Tutorial

Link: https://pythonprogramming.net/haar-cascade-object-detection-python-opencv-tutorial/

7. Contribution:

Akshay Penmatcha: I have initially started off by implementing a Multi-Layer perceptron model on MNIST 0-9 numbers dataset to approach the vehicle detection problem through a neural networks perspective. Later after we decided to stick to the Feature extraction and Shallow-Learning techniques I took up the HOG Features and SVM Classifier approach part of the project that we talked about in the report. I've read the Dalal and Triggs paper and understood all the technical knowledge that we was required to take this approach. I then implemented the HOG-SVM algorithm to get the final results. I have written the report and all the theory that accompanied this approach by reading up various sources and doing tutorials. I have also reached out to a friend who is working in the Computer Vision to get some initial guidance to set forward the approach of our team. We as a team have met with a NYU Tandon Graduate student where we tried understanding the nuances involved in implementing a HAAR Cascade Classifier.

Priyanshi and Sunny:

a) Studied in detail the paper written by Viola and Jones paper in year 1999 focused on the implementation of upright Haar-like features.
b) Also, explored the paper written by Rainer, Alexander, Vadim which uses the novel concept of 45 degree rotated Haar-like features.
c) In our paper and presentation, we have covered in detail the underlying concept of Haar-Cascade Classifier using various techniques such as Integral imaging, Feature Extraction using Adaboost Algorithm and Unweighted combination of Weak classifiers.
d) Encountered challenges in installing OpenCV and CMake libraries and were able to successfully run it only on Sunny's laptop (Mac).
e) Implemented Haar Cascade classifier for face and eye detection using the Intel's xml files and dived deeper to understand the end-to-end process of generating the xml and vector files for positive object detection.
f) Implemented the learnings by training our own Cascade Classifier to detect vehicles in images as well as video feed. This implementation required thorough understanding of both the theory concepts as well as exploration of OpenCV libraries and other packages such as tensor flow. Priyanshi and Sunny have equally divided the extensive research required and the implementation of vehicle detection program.