VEHICLE DETECTION PROJECT

Faisal Waris

Udacity
Self-Driving Nano Degree

Introduction

The goal of this project to process a video stream to detect and mark vehicles in each frame.

The project work is structured as follows:

1. Train a classifier to distinguish between car and non-car images. Test the classifier on out-of-sample data and tune classifier for accuracy.
2. Implement method for multi-scale, sliding-window search to identify vehicles and mark their locations in an image, utilizing the trained classifier.
3. Track vehicles from frame-to-frame to reduce false detection rates and smooth detection annotations
4. Implement a pipeline to process a video file and create an annotated video file (see Figure 1)

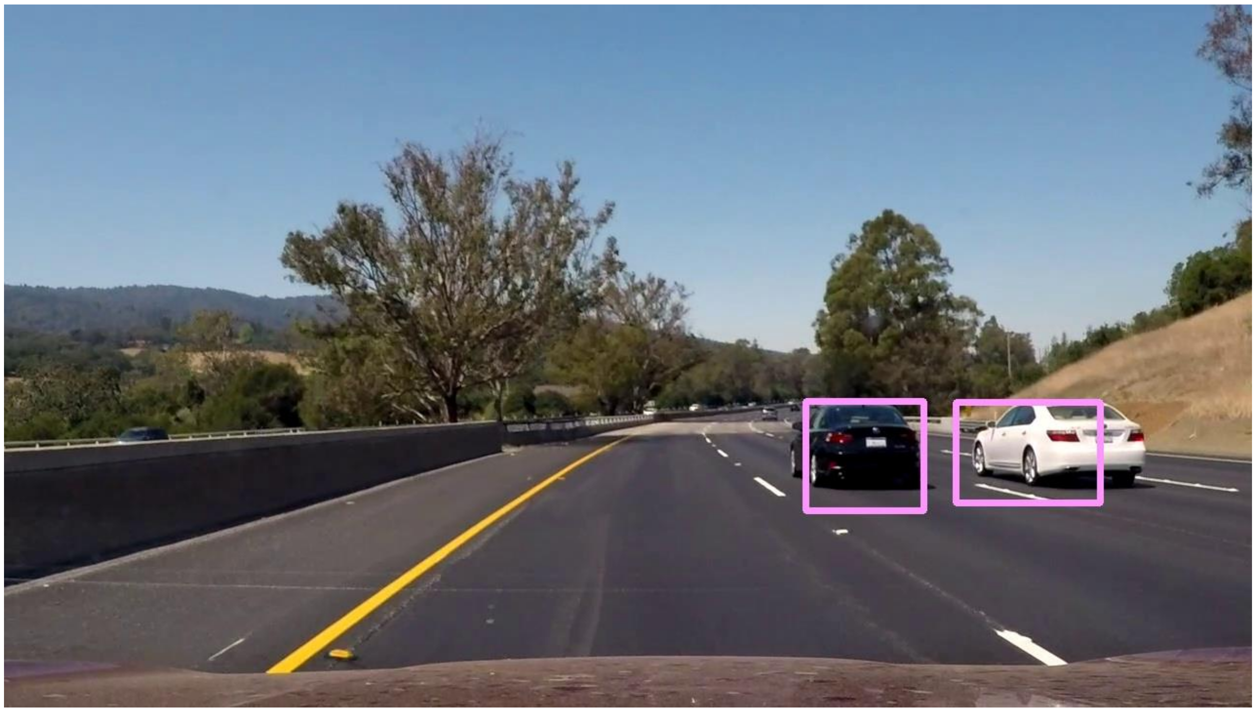

Figure 1: Example annotated video image

The steps outlined above are explained in detail in the subsequent sections of this report which follow the project organization section, next. The report concludes with a discussion of the performance of the approach taken and some of its limitations.

Project Organization

The vehicle detection project is available in a GitHub repo. The major parts of the project and its main output are described and linked to in the table below:

GitHub repo	https://github.com/fwaris/object_detect
Output video with vehicle tracking	https://github.com/fwaris/object_detect/blob/master/output_video/project_video_out.mp4

Keras model training	https://github.com/fwaris/object_detect/blob/master/CNTKerasModel/classifier.py
Main script file for processing video and outputting annotated video	https://github.com/fwaris/object_detect/blob/master/ObjectDetection/VehicleDetectHog.fsx
Object tracking	https://github.com/fwaris/object_detect/blob/master/ObjectDetection/ObjectTracking.fs
Non-maximum suppression	https://github.com/fwaris/object_detect/blob/master/ObjectDetection/NMS.fs
Vehicle detection using trained model	https://github.com/fwaris/object_detect/blob/master/ObjectDetection/Detector.fs
Search window settings	https://github.com/fwaris/object_detect/blob/master/ObjectDetection/DetectorSettings.fs

More detailed description of the processing is given in the rest of the document with links to the relevant sections of the code, when appropriate.

Training Data

This section describes the source datasets used and their preparation for training the classifier.

Two training sets were used:

a) Training set provided in the project repo
b) And the additional dataset provided by Udacity, annotated by Autti.

For Autti, the positive cases where extracted using the annotation CSV file and negative cases where extracted by randomly selecting 64x64 of the road, tree and sky sections. The code to process the data is provided in this project file.

Approx. 26K positive and 50K negative images were available for training. Out of the total available, 70% of the images were used for training, randomly selected from the full set and the rest used for validation. The preparation of the input data for classification is coded here.

Training the Classifier

After some early research into different methods, the **OpenCV** SVM was chosen because of its compatibility with the **OpenCV** HOGDescriptor class. The built-in **OpenCV** HOGDescriptor can be configured with a trained SVM detector to perform multi-scale, sliding window search for objects in an image. However, the SVM classifier did not perform well enough for the project purpose and was discarded in favor of convolution network based classifier. For reference, the SVM training and object detection code is provided in this file.

A Keras model was created using a modified version of the CIFAR-10 model code available from the site machinelearningmastery.com. Unlike the original model which was trained on TensorFlow, the project model was trained on the CNTK backend. It is available in this Python file in the repo. CNTK was used because of its compatibility with .Net for runtime evaluation. The trained CNTK model is available in the models folder of the repo.

The model was trained on the mobile version of the Nvidia GTX 1080 installed in an Acer Predator gaming laptop. The training time was about 10 minutes for the above datasets. The model accuracy on out-of-sample data is 99.9%.

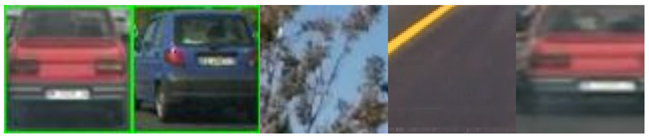

Figure 2: Examples of positive and negative images used for classifier training

Several color-space models were tried – BRG, HSV, HSL and YCrCb. For this task, YCrCb gave the best results and was chosen for image normalization.

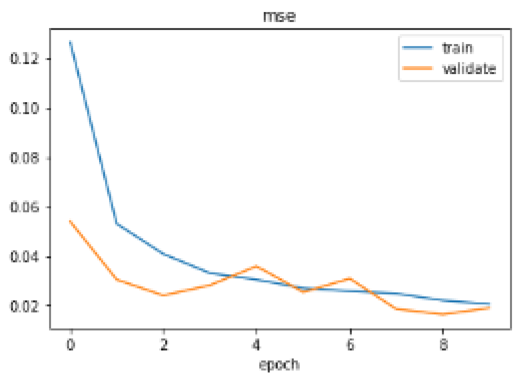

Figure 3: Training and validation loss progression

Vehicle Detection

A multi-scale, sliding-window search method was employed for object detection and location. The method used is based on the one described in the course material and has the following features:

- Only the lower part of the image is used to detect vehicles to reduce processing time.
- Smaller rectangles are used for distant detections and progressively larger ones for nearer ones
- Multiple detections of the same image are first consolidated with non-maximum suppression and then any remaining overlapping detections with a heat map based approach

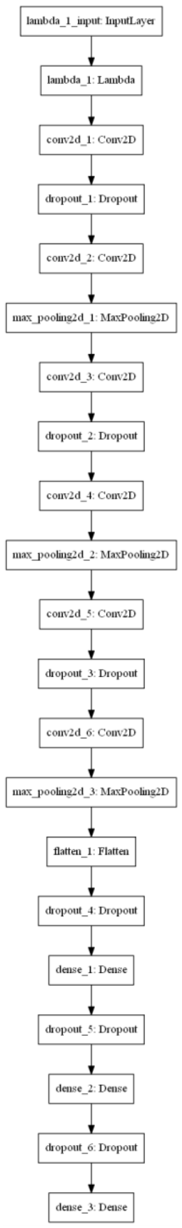

Figure 4: Visualization of convolutional net

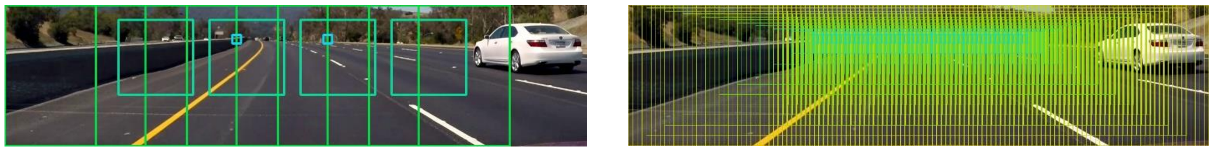

Figure 5: Partial and fuller renderings of search windows in lower part of the frame

The search windows Rect structures are cached to speed up image processing. The code to create and cache search windows is in the DetectorSettings.fs file.

Image data is extracted for each search window and resized to 64x64 to match the classifier input size. The classifier is run over the resulting image. It outputs a probability of detecting a vehicle. Only the search windows with high enough probabilities are retained.

There are 1034 windows searched for each frame. The evaluation of search windows is done in batches (of up to 2000) to reduce processing time. The neural net scoring code is in the Detector.fs file.

Multiple detections are first consolidated using non-maximum suppression and then using heat map processing as shown in Figure 6.

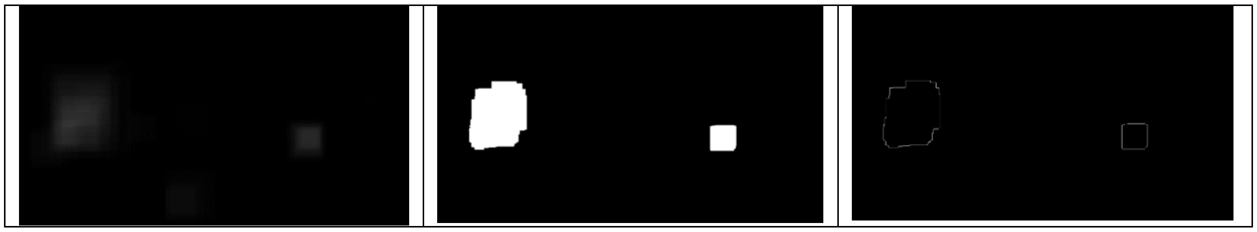
Figure 6: Heat map processing: Gray scale -> thresholded -> contoured

For the heatmap processing**,** OpenCV is used to first create a grayscale heat map image which is then thresholded to sharpen boundaries. Contours are extracted using the OpenCV FindContours function. Bounding boxes for the contours are extracted using the OpenCV BoundingRect function. The final bounding box rectangles are treated as candidate vehicle detections. Candidate detections are used by the tracking component to calculate the final position and size of the annotation rectangles.

Vehicle Tracking

Candidate or raw detections, from the processing outlined above, are handled by the tracking system to convert raw detections to a location of the vehicle in the current frame. Track processing is outlined below:

- At start, each raw detection rectangle is converted into a candidate track
- If a subsequent frame has a close detection, it is added to the track. When enough detections are collected, the track is actually rendered on the frame
- With two or more detections in a track, a regression line is fit between the center of the track rectangles. This line is used to get a predicted value of the center of the track in the next frame
- The track regression line is re-estimated with each new matching detection (up to the track rectangle limit)
- Each track also maintains the average size of rectangles calculated from the previous detections stored in the track
- The average size and predicted center are used to draw the bounding box for the track in the current frame
- To reduce jitter, a smoothing function is applied to the predicted center and rectangle size

The tracking code is provided in the ObjectTracking.fs file.

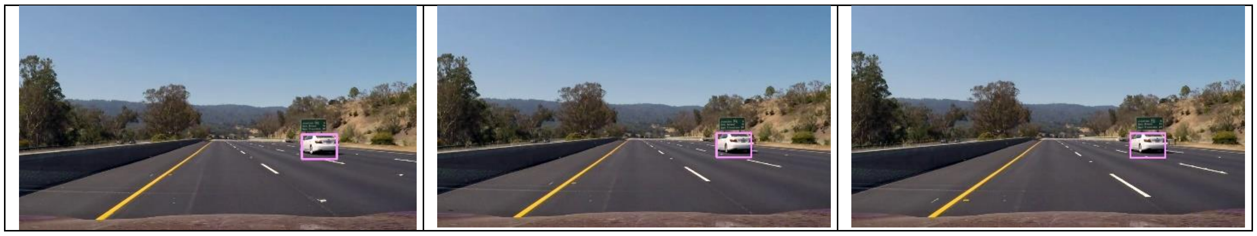
Figure 7: Tracking reduces jitter across adjacent frames

Tracking suppresses false positives as spurious detections are omitted and eliminates jitter

Video Processing

For the video, each frame is processed as follows:

1) normalize the frame by converting to YCrCb format
2) Evaluate the convolution net for each search window and output the probability of detecting a vehicle in the window. The evaluation is performed in batches of 2000.
3) Search windows with a high enough probability are then consolidated through non-maximum suppression and then via a heat map based approach
4) The raw detections are fed into the tracking system to create / update tracks
5) The tracks are then drawn on the output frame.

The main code for creating the video is in the VehicleDetecHog.fsx script file. The function `testVideoDetect` implements the controlling logic for creating an output video from an input video.

Discussion

In the early phases of the project, an SVM based approach was used for vehicle/non-vehicle classification. The SVM model achieved a 95% accuracy but that still resulted in many false positives, especially when applied with multi-scale, sliding window search on the video frame images.

To improve performance, the SVM classifier was traded in for a convolutional net. The neural net is able to achieve 99.9% accuracy (on test data). While the incidence of false positives was much lower, they were still present. Careful tuning of search window parameters (e.g. number of rows and horizontal shift in pixels from one window to next in the same row) further reduced the false positives (and false negatives) to a manageable number.

Addition of a tracking mechanism permitted false positive and negatives to be masked effectively without any discernable visual loss of accuracy. In addition, the tracking mechanism enables smooth video annotations and eliminates the inevitable jitter of individual frame detections.

The final result is mostly error free in terms of vehicle detection and tracking. Only at the very end the tracking is a little confused when the black car overtakes the white car.

The video processing takes about 1 second per frame using GPU based evaluation and thus is not real-time. Faster processing could be achieved by perhaps down sampling the video frame image to detect regions of interest more quickly that are then exploring promising regions only. Newer neural network approaches for object detection e.g. Fast R CNN may also be a viable option.