Stopify - Real time road sign detection

Lane Breneman,Joshua Deare, Hari Caushik

June 1st, 2015

Introduction

Our clients, ON Semiconductor requested that we research and implement computer vision algorithms to be able to detect street signs from video or still images intended to be captured from a low cost camera inside an automobile. The algorithms were required to run on low cost development boards like the Beaglebone Black and the NVIDIA Jetson TK1. ON Semiconductor recently acquired Aptina, the leading producer of CMOS image sensors and hope to enter into the Automotive Safety industry sometime in the future. They intended this to be a research project in which we can research, develop and test a variety of stop sign detection algorithms and analyze them to determine which algorithms seem most promising, which development boards are most promising and provide a base for future senior design projects to contribute to.

Our clients at ON Semiconductor were Matthew Zochert, Carl Price and Don Reid. The members of our team were Joshua Deare, Lane Breneman and Hari Caushik. Throughout the year, we each contributed to the creation of the algorithm development framework, testing framework, capturing of images and video for the test dataset and presenting our work. Our clients tracked our progress, provided suggestions for our algorithms, use of hardware, and the presentation of our work and even provided us with starter code early on to ramp up on programming with the OpenCV C++ library.

Main Project Changes

Weekly Blog Posts

Project Documentation

Our project consists of four main components: the development framework, algorithms, testing framework and test dataset. The development framework provides a well defined interface to develop the computer vision algorithms and runs the algorithms on provided images and displays the output. The algorithm used by the development framework is linked based on rules provided by a Makefile. The algorithms process the images and determine whether they contain a stop sign or not. The testing framework is used to test the effectiveness of each of our algorithms based on a number of evaluation metrics. It essentially invokes the development framework and the algorithm linked to it on each image in the test dataset.

Development Framework

Usage The development framework, written in framework_images/framework_cpp/framework.cpp from the root project directory, is a C++ program that takes a command line argument, the path of the image to process, and returns a 1 if the algorithm determined that the image contained a stop sign or a 0 if not. An optional third argument can be provided to display the image being classified. For example, to classify framework_images/framework_cpp/pos_sample.jpg, an image containing a stop sign, the proper usage is

```
./framework pos_sample.jpg
```

to simply return the classification result without displaying it. To display the image, this would be

```
./framework pos_sample.jpg 1
```

although the actual value of the third argument does not matter, just its presence.

Implementation Details The development framework includes the framework_images/framework_cpp/algorithm.h header file that provides the interface the algorithms must be written to. This is the processImage function, with the signature

```
int processImage(Mat &img);
```

The framework loads the test image from the path specified in the command line argument onto memory with the OpenCV imread funtion as follows:

```
Mat img = imread(argv[1], CV_LOAD_IMAGE_COLOR);
```

The second argument specifies the loaded image to be a 3 channel color image and the output is stored in an OpenCV Mat object, a wrapper around a multi-channel 2-dimensional image array.

Algorithms

HOG

The HOG algorithm works by extracting a HOG Descriptor from a 64x128 image window this descriptor is then run through a SVM to see whether the window contains a stop sign. If it does the match is added to a vector of bounding rectangles which represent the matches. The window is then shifted over a little bit and then rerun. This repeats until the entire image has been processed. The matches are then run through non-maximum suppression to eliminate overlapping matches.

To run detector code you train a model to do this navigate to the hog directory. In the git projects this is in $/framework\_images/framework\_cpp/hog/$. From there you can run the command(assuming you have a tagged dataset):

```
./trainer [path to dataset json] [path to output model] -c 200 -e 1e-6 -i 450 -o .5 -s
    1.10 -r 5
```

The arguments:

- c - C value for the SVM (See CVSVM documentation)
- e - Epsilon value for the SVM (See CVSVM documentation)
- i - Max number of iterations while training
- o - Amount of overlap required to be considered a true positive
- s - How much the image pyramid scales when trying to match (See gpu::HOG documentation)
- r - How many overlapping rectangles are required during suppression to be considered a match

This will take a while but once its done it will output some basic statistics on how the new training set performed. To use this trained model we have to copy it to the models folder in the root git directory. With that done edit the name of the module files used in *detector_server.cpp* on line 10 like so:

```
svm.load(<absolute path model file>);
```

The path must be absolute since we don't know what directory 0*detector_server* will be run in and we want to be sure it finds the model.

With that done we can now run it, to do that you will need two terminal sessions. In the first run:

```
make detector_server
./detector_server
```

In the second navigate to to the $framework\_images/framework\_cpp$ directory and run:

```
make hog_server
make test
```

The make test isn't necessary but it will test that the server is running properly. Now you can use it like any other algorithm in the framework.

SURF and Bag of Words

SURF and Bag of Words(BOW) are actually two algorithms that are used in conjunction to classify an image. Classification is different than recognition in that it can't tell you where the stop sign is in the image, just that one exists. The algorithm works in three stages:

1. SURF is run over a tagged dataset, where it extracts the surf descriptors from any tags inside the image. These descriptors are added to a list.
2. The list of all the descriptors is passed to a K-Means clustering algorithm that will find any groups of clustered points. These groups are then used to train a BOW vocabulary.
3. We then run SURF over the full sized images in another test dataset. These SURF descriptors are then passed to BOW vocabulary which creates a new vector.
4. This vector is then used to train a SVM. It is labeled positive if there is a tagged stop sign in the image, otherwise it is label a negative example. The SVM is then trained/
5. To actually classify an image you just run 3 and 4 over an image but instead of training SVM you predict with it. If it predicts true there is a stop sign, otherwise there isn't.

This algorithm ended up have some issues that should be kept in mind before using:

1. It is slow, taking 1.5 seconds per a 400x300 image.
2. It is only kind of accurate, the best was 72
3. It is proprietary, meaning that you can't use it in any final commercial project. Additionally NVidia doesn't include it in the optimized OpenCV library which means that you will have to give up the OpenCV optimizations to use this algorithm on the jetson.

To use this algorithm we first need to train it. To do this we will navigate to the */framework_images/framework_cpp/bo*
in the git path. Once there we can run:

```
make trainer
./trainer [taged dataset .json] [name of file to save the model to] [name of file to save
    vocab to]
```

You will notice that this outputs two files, the first is the SVM model, the second contains the vocabulary object. Both of these files should be copied to to models folder. Next edit detect.h and set the MODEL and VOCAB constants to be the absolute path to the files that were just created. To run the detector you need to naviagte down to the *framework_cpp* directory. Build and run it using:

```
make bow
make test
```

ROI

Testing Framework

The testing framework is pretty straight forward and requires two prerequisites:

1. The main framework is already compiled with the algorithm you want to run
2. You have a tagged dataset with the corresponding .json file

With that done to run the testing framework use:

```
./tester [json file] [path to the result .csv you want create] [path to the framework
    executable]
```

This will create a .csv file with the result statistics for the algorithm.

Test Dataset

Resources

The following resources were particularly helpful:

What We Learned

Hari Caushik

By contributing to this project, I learned a fair bit of technical information. I improved my C++ programming skills and gained a familiarity with the OpenCV library. In addition, I learned about the major concepts in computer vision algorithms as well as image processing techniques. Throughout this project, I also learned about software design practices and made practical use of algorithmic complexity analysis.

A lot of non-technical information was also learned by doing this project. This project required a lot of documentation at the start while we were determining the specifications, during the implementation when we documented the pseudocode and algorithmic complexity of the algorithms as well as our final report. My technical writing skills have improved as a result. We also presented our project, particularly towards the end of the year at the Engineering Expo to the general public, at ON Semiconductor to ON employees and our final presentation video.

Doing a nine month project for an actual client in industry has taught me most of all that the main challenge is carefully defining an initially very loosely defined problem. Decisions that we make early on can have a very significant effect on the way our project may turn out. I have also learned that projects in the real world are very malleable. While the high level decisions we make early on guide our project in certain directions, specific requirements frequently change as we gain more knowledge through implementation.

This project allowed each of use to take a project management role at different times throughout the year. I learned most of all that establishing clear communication among all team members is essential to moving the project along smoothly. It is much easier to perform when each of us knows what is expected of us and have clearly defined responsibilities. When this breaks down, there tends to be a lot of confusion and consequently inaction.

Lane Breneman

Before this project I had used OpenCV for object recognition. That said I had no experience working with the C++ interface and very little C++ experience in general. Through this project I learned a lot about linking libraries, creating make files and using classes in C++. On a different note I also learned a lot about SVMs and machine learning. I had attempted to use these tools but had never made much progress, but after this project I feel I have the skills to employ them in future projects.

I learned about taking notes and extracting the important information from research papers in order to reimplement their algorithms. I also learned how important communication between team members is. This largely came from the fact the two of our basic algorithms turned out to be very similar and two of our advanced algorithms are also kind of similar. This limited the variety of algorithms we explored, thus lowering the quality of the project.

Our project integrated really nicely together, a large part of this I attribute to our good design. We found that we almost never needed to fight the framework in order to get our code to run, which is nice. Even when last minute we need to make a way to export videos it took about 5 minutes to extend the framework. This reinforced the most import fact that I knew about project work, that a good design is important.

Project management was a very mixed bag for our project. In the beginning we had very good project management, planning everything, dividing work and communicating often. This lead to the best parts of out project namely the framework and testing framework. Our later work were less thought out and more divided and I feel that it can be seen in quality of the work, as our algorithms could have been better.

Communication is key. We had a really great group and I enjoyed working on the project, but for most of the term the project became mostly individual. This lead to two of our algorithms being pretty much the same. If we had communicated in more detail we could have seen this sooner and adapted.

If I could make one change I would have done 1 advanced algorithm each that we worked on over the year. By advanced I mean more complicated than even the HOG or SURF algorithms. I feel the project would of had more to show and been more rewarding if we had.

Joshua Deare

Appendix 1: Essential Code Listings

Appendix 2: Other Material