YOLOv3 Realtime Sign Detection & Classification

Lewis Collum

Contents

1 Example Results

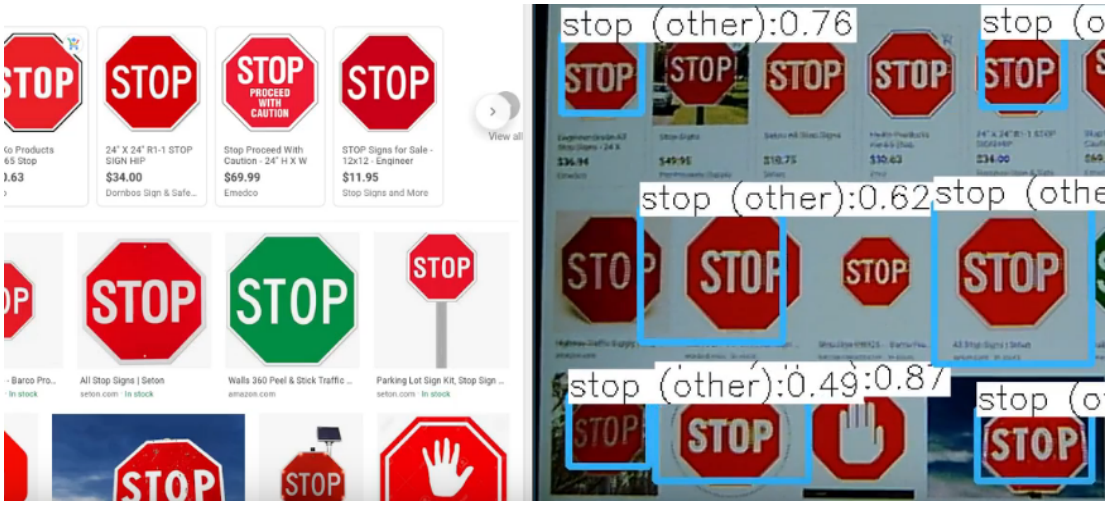

Fig. 1. Stop sign images on a monitor (left), and the processed video feed (right).

2 Choice of Object Detector

Object detection and classification is as simple as, detect signs then classify –not exactly. This is how object detection[1] algorithms started (e.g. R-CNN), but they can be too slow for real-time

1. Classification is not detection. Objects first need to be detected before they can be classified. But, for briefness, we say "object detection" when we really mean "object detection and classification."

(and embedded) object detection. Those looking for speed, use algorithms that extract classified objects from the frame in a single pass, as opposed to two passes. YOLOv3 is an algorithm that detects in a single pass, such that it detects **and** classifies signs at once.

There are multiple versions of YOLOv3. We are using YOLOv3-tiny-prn, which has the highest frames per second (FPS) compared to other commonly used algorithms (see figure 2). The sacrifice for speed is accuracy, as YOLOv3-tiny-prn borders around 35% average precision. Since we implemented sign detection on a Raspberry Pi as a proof-of-concept, this accuracy is acceptable.

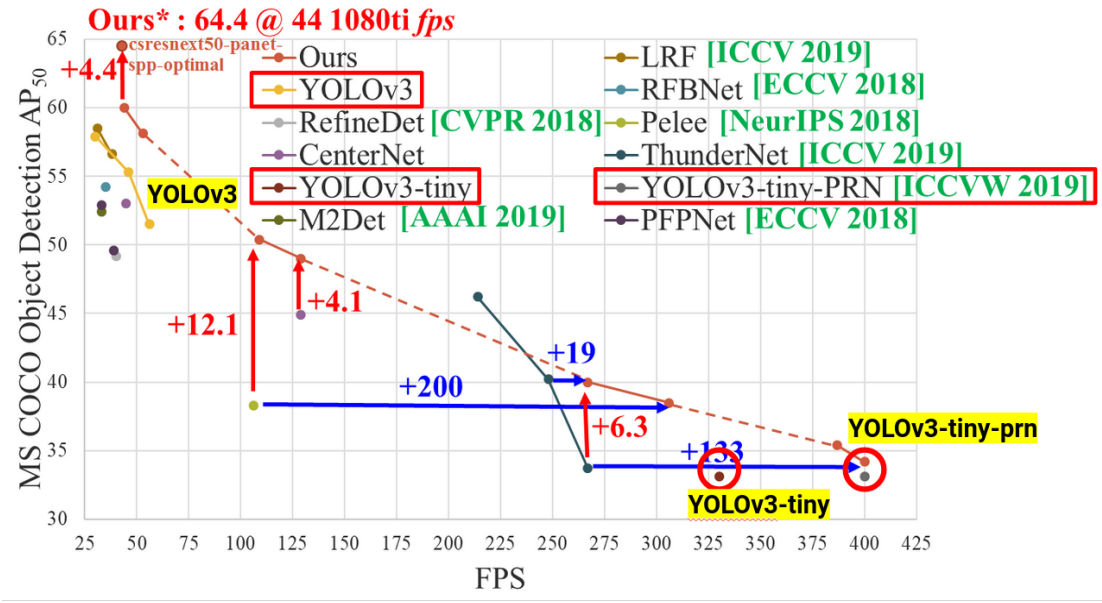

Fig. 2. YOLOv3-tiny-prn has the highest FPS, with an acceptable 35% average precision.

3 YOLOv3 Implementation

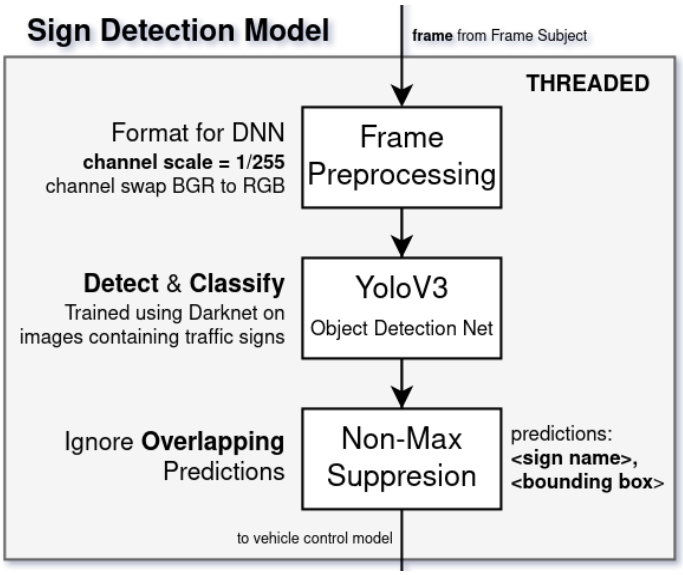

Fig. 3. The frame is formatted to fit in our YOLOv3 network, after the formatted frame is passed in, the resulting predictions are filtered by non-max suppresion and sent to the vehicle control model.

Before we started the automation system, we used OpenCV to import our trained Darknet[2] YOLOv3 model configurations. This allows us to pass frames into our custom-trained YOLOv3 network and receive sign predictions.

As the automation system is running, the incoming frame from the frame subject is first formatted for our YOLOv3 sign detection neural network. This includes scaling the image by $1/255$ so the image fits in our network, and flipping the blue and red channels. Subsequently, the formatted frame is passed into our YOLOv3 network. The network outputs predictions that include the name of the predicted sign, and the bounding box which provides the location of the sign. We pass this output through a non-max suppression algorithm to potentially reduce the number of overlapping predictions. This is all we need from our sign detection model. The predictions are now sent to the vehicle control model.

4 Training the YOLOv3 Network

4.1 Dataset: GTSDB

We trained our network on the GTSDB dataset containing 900 images from the point-of-view of vehicles on a road.

Fig. 4. An example image from the GTSDB dataset

4.2 YOLOv3 Darknet

We trained out YOLOv3 network with Darknet. Darknet requires that we provide it with annotations for each image. Annotations include information about the bounding box and class of each sign in an image. The GTSDB dataset we are using contains an annotation file but it is not in the same format as what Darknet requires. We use a script to convert this annotation file to the Darknet annotation format (as illustrated in figure 5).

2. Darknet is an open source neural network framework.

Fig. 5. Darknet annotation file, `00001.txt`, for a single image `00001.jpg`. The file contains only space-seperated numbers, with five fields.

We can train using Darknet as an executable and supply it with a directory configuration file and a model layers configuration file.

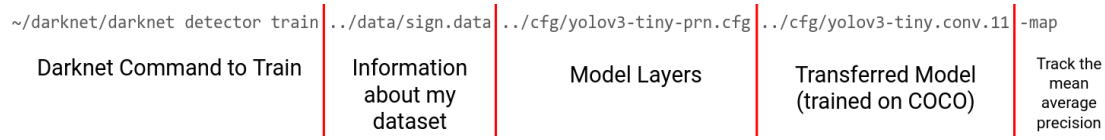

Fig. 6. A typical darknet command to train our YOLOv3 network. `yolov3-tiny.conv.11` is a trained network provided by Darknet, and we use it for transfer learning.

In reference to figure 6, our `sign.data` file looks like

```
classes = 43
train = <data directory>/train.txt
valid = <data directory>/test.txt
names = <data directory>/sign.names
backup = <weights directory>
```

And, a directory setup may look as follows:

```
- cfg
  - yolov3-tiny.conv.11  <Trained network (from Darknet), used for transfer learning>
  - yolov3-tiny-prn.cfg  <Network layer configuration>
- data
  - images_jpg  <raw jpg GTSDB images>
  - images_ppm  <GTSDB images converted to PPM>
  - labels  <Darknet text lettering>
  - obj  <Mixed PPM images and Annotations> IMPORTANT
  - sign.data  <directory configuration file>
  - sign.names  <class names sperated with a newline>
  - test.txt  <absolute path to testing images>
  - train.txt <absolute path to training images>
- weights
  - yolov3-tiny-prn_best.weights  <Output from Darknet>
```

4.3 Training/Testing Distribution of Images

We created the `test.txt` and `train.txt` files (mentioned in the previous section) by randomly selecting a proportion of images to be training images and testing images, and then providing the absolute path of the images in their respective `test.txt` or `train.txt` file.

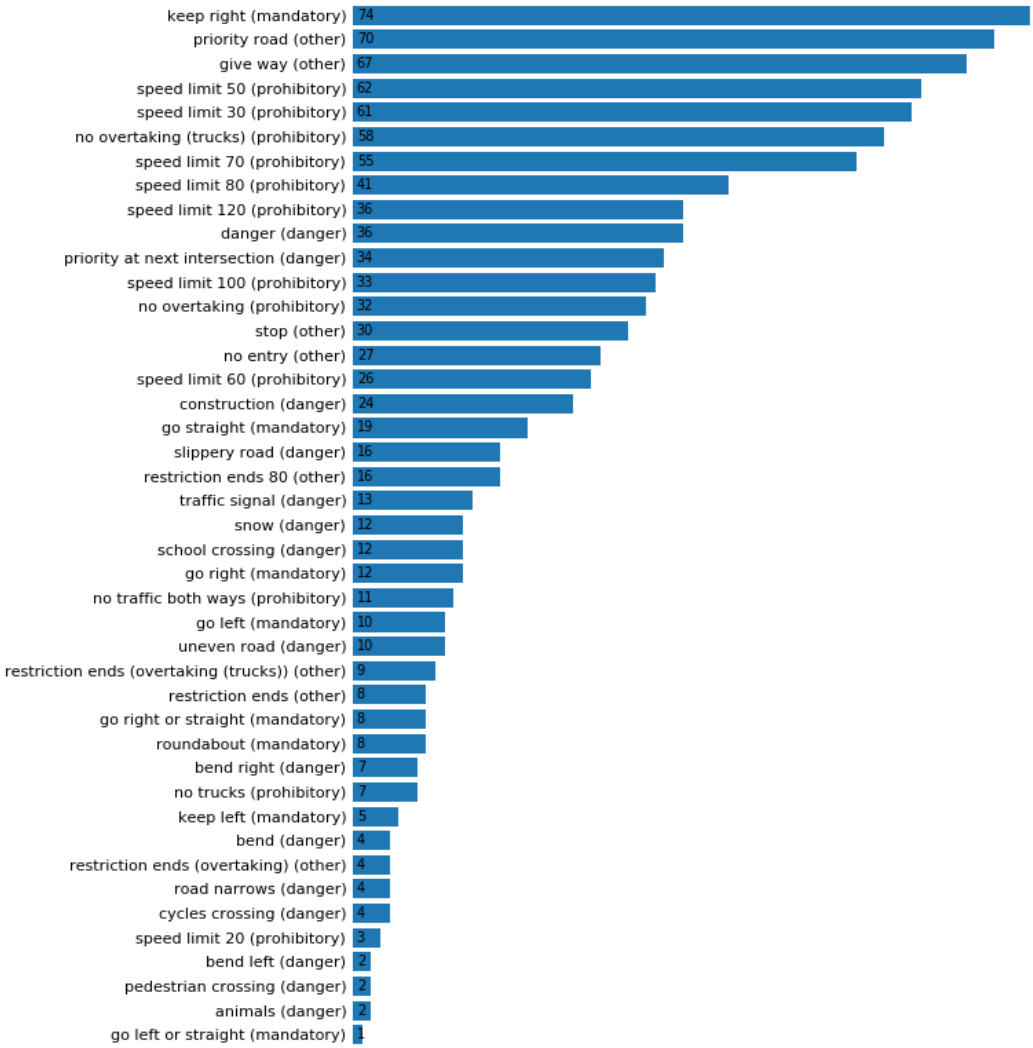

Fig. 7. Distribution of testing images.

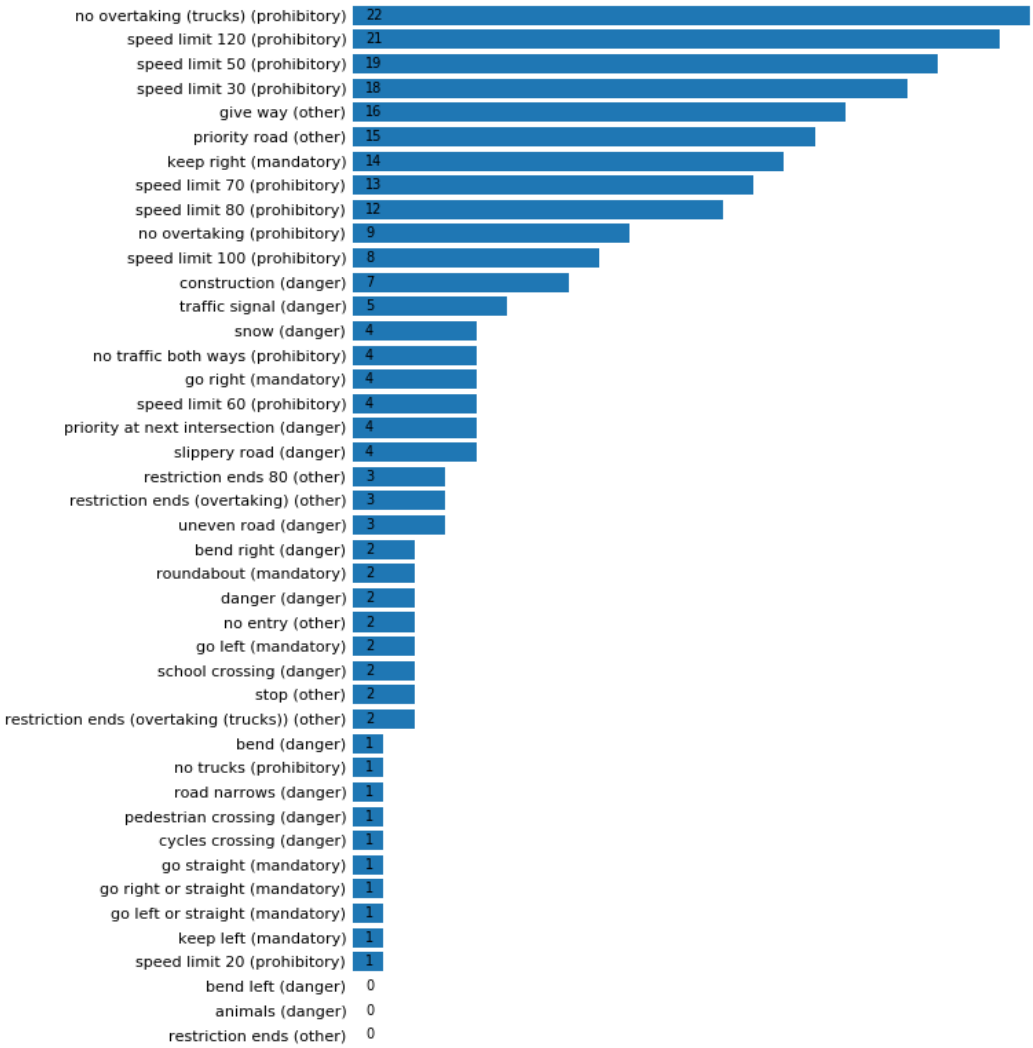

Fig. 8. Distribution of training images.

These distributions have some classes with only a few or no images. This causes a lower average precision for our network, since it does not have enough of those images to train and test on.

4.4 Results

We obtained a mean-average precision on the validation set of around 10% (see figure 9). However, more specifically, the average precision for easy to classify signs (such as stop signs and yield signs) was much higher than signs which had few training images, and signs that relied on textual classification (such as speed limits). On the Raspberry Pi, the YoloV3 network operates at around one frame-per-second (see figure 10).

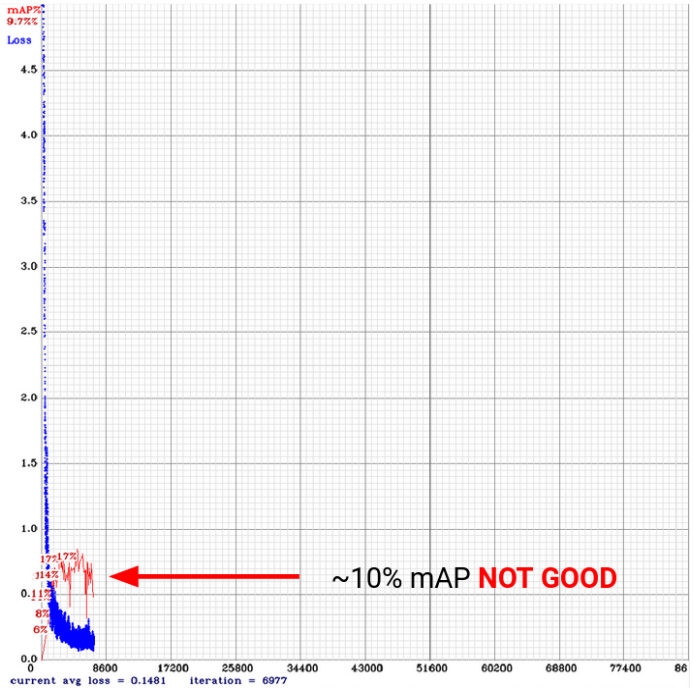

Fig. 9. mean average precision over 6000 training iteration at around 10%. Loss (in blue) drops quickly, due to transfer learning.

Once the model was trained, we loaded the model into our main application on the Raspberry Pi. We added the option, in the side-panel of our livestream, to show detected signs. Now, on the vehicle's website we can see the bounding box and class of detected signs (as seen in figure 10).

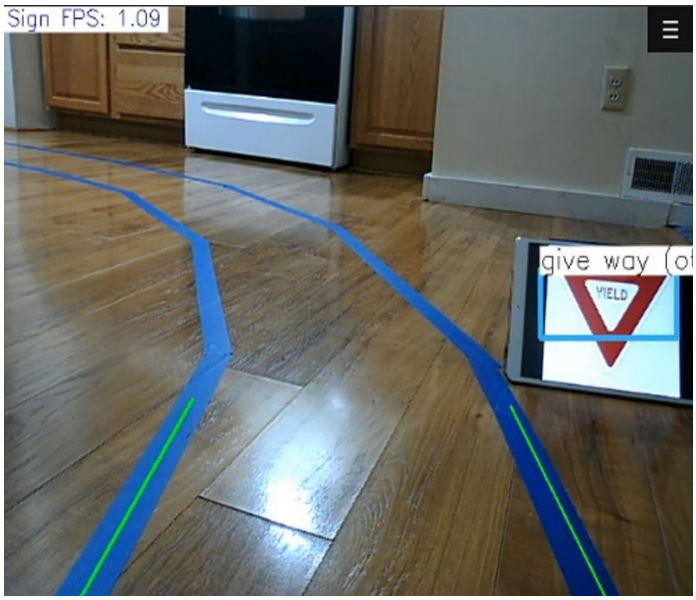

Fig. 10. Sign detection on the Raspberry Pi. Detects signs at about 1 frame-per-second.