Stop Sign Detection Report

Abstract

We try to solve the task of stop sign detection using popular object detection models such as YOLOv3 and the Laboratory for Intelligent and Safe Automobiles Traffic Sign Dataset. Here we will go over our methodology, dataset, and various other aspects that we learned over the course of the project.

Introduction

With the rise of autonomous systems such as Tesla, Waymo, and Uber's self-driving car projects, the need for quick and reliable sign detection systems couldn't be more important. These embedded systems are running in high-risk and fast-moving environments using limited onboard hardware to make sense of the world around them. Thus, sign detectors that run on these systems have to be incredibly accurate and incredibly efficient in time. We will attempt to build such a system and measure its accuracy and its efficiency.

Figure 1. Color Stop Image Detection

Method

We will attempt such a task by training the popular object detection model YOLOv3 using transfer learning from weights trained on the COCO dataset. Afterward, we will evaluate the model on our test set of stop sign images and see the mAP and time scores.

Experiment

Data

The first thing we need is a dataset to train these images. We will be using the LISA Traffic Sign dataset [2]. These are grayscale images extracted taken by several cameras from the perspective of a moving car. Various traffic signs and road markings have been labeled and corresponding bounding boxes identified. Because we are focusing primarily on stop sign detection, we will be extracting only images that contain a stop sign. Additionally, because the dataset is grayscale, YOLOv3, which normally runs on color images, has less information to use to identify stop signs. For example, instead of having access to the different RGB values of each pixel, we just have a single intensity value, and our information is 1/3 of what it could be.

Figure 2. Color Stop Sign Image

Figure 3. Grayscale Stop Sign Detection

Results

Metric	Train	Test
mAP	1.0	0.19
Precision	0.425	0.0924
Recall	1.0	0.229
F1 Score	0.597	0.132
Size	82 images	28 images
Speed	0.04 s / image	0.04 s / image

Here are the metrics for the model trained after 300 epochs on the training set. As we can see the metrics on the training set are very good. A mean average precision score of 1.0, a precision score of 0.425, a recall of 1.0, an F1 Score of 0.597 with a speed of 0.04 seconds per image. This is pretty expected since the model is most familiar with these images. However, the metrics on the test set are not so good. The results on the test were a mean average precision of 0.19, a precision of 0.09, a recall of 0.229, an F1 Score of 0.132. These are awful scores and definitely not acceptable for any realtime system. I believe the main issue has to do with what kind of images this dataset consisted of. While these scores, on the test set, are very poor, the speed is incredible at 0.04 seconds per image or in other words 25 frames per second. This would be very favorable in real-time environments, but the lacking precision remains are large negative factor.

How the dataset was collected was a major factor of the poor test results seen above. As we can see below two images of the test set. These are different photos, but they are very similar because each section of the dataset consists of a series of frames collected from video clips. This causes much of the data to be very similar to each other and not having much variation. Thus when we trained our model on a series of frames from a single video, the model was only exposed to certain conditions that of that video and was not able to generalize to frames from other video clips very well. Additionally, due to the very small size of our training set of 82 images, again it did not have enough information to learn enough features. The dataset itself was not bad, but we should have included a lot more images and a lot more varied ones by possibly combining datasets.

Figure 4. A Frame of Driving Video

Figure 5. Another Frame of Driving Video

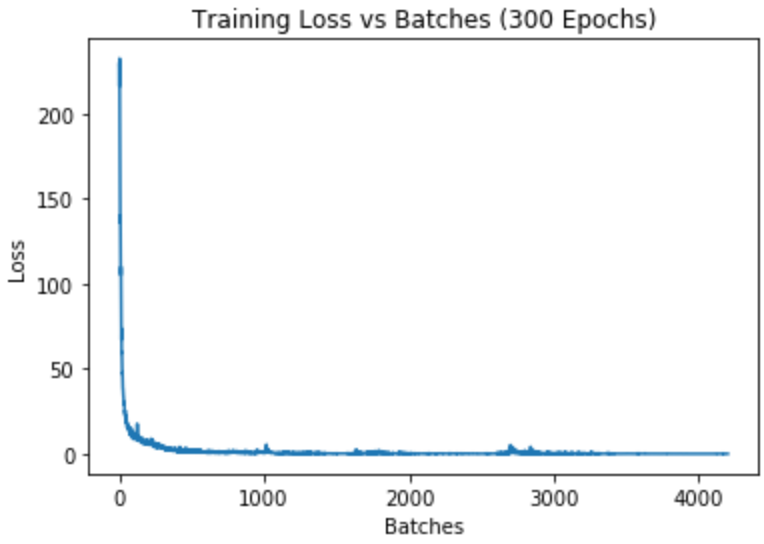

Figure 6. Training Loss vs Batches

Above we have our training loss graph. As you can see most of the learning happened within the first a couple hundred batches. (At 14 batches per epoch, that's about 20 epochs.) After that, the loss, for the most part, kept decreasing, but at a much slower rate. From this graph, it seems that the 280 epochs after the first 20 were not very important and that in future attempts, after about 20 epochs we have a good measure of whether or not the model is learning and how it is doing. With this information, we can try and tweak a lot of hyperparameters and try various data preprocessing ideas in a much shorter time.

Figure 7. Duplicate Detection

In Figure 7, we can our model detecting the stop sign in this image. Both detections were indeed correct on the stop sign, but the one on the right is little off from the stop sign and should not be included. We could fix this by increasing our confidence threshold or non-maximum suppression threshold. Currently, they are a confidence threshold of 0.8, and a non-maximum suppression threshold of 0.4. The downside of increasing these thresholds is that we may miss many stop signs that don't meet these thresholds.

Figure 8. Incorrect Detection

In Figure 8, we see that the trained model correctly detected the stop sign on the upper right, but it also has one more additional detection. This one is on the upper left and fairly small. From closer inspection, it is just a section of leaves from a tree. This definitely not a stop sign and an incorrect detection. I believe there are a few ways we can attempt to fight this incorrect prediction. We can simply just have more data, and thus make it less likely for these errors to occur. Additionally, we could also increase the confidence and non-maximum suppression thresholds to prevent this type of error. However, as mentioned before, it might make it harder to detect real stop signs. One last method that I can think of to attack this, is to include incorrect detections like these are negatives in the dataset, so the model absolutely knows this does not count as a stop sign. As it currently is just detecting a single class which is the stop sign class. We could add another class that just includes all the negatives.

Figure 9. Incorrect Detection and Detection Failure

In Figure 9, we can see a stop sign in the upper right and incorrect detection in the upper left. For the missed stop sign, we could lower our thresholds to be able to detect it, but again at risk of detecting non-stop signs like in the upper left. The better improvement would just be improving our dataset as we already know YOLOv3 is a well-established model that produces decent results. This image was taken from the test set, which shows that the model really is failing at stop sign detection on images it hasn't seen and thus it is not generalizing well.

Figure 10. Correct Detection

In Figure 10, we see the model correctly detecting the stop sign in the upper right and with a pretty good IOU from the appearance. This frame is taken from the same video clip from Figure 9, however, this frame is later in time and when the camera is much closer to the stop sign. This is an example of the model working for some test set images, so it shows that it has a lot of potential to be better.

Challenges

YOLOv3 Model Code

A lot of the code was meant to be run from the command line which was nice to get started and if I only wanted to train, test, and perform inference, but since I needed to examine different metrics, I needed to make it runnable from a Jupyter Notebook. This required refactoring large portions of code so that they didn't just take command line arguments, but were encapsulated into functions that could be imported.

Additionally, there were a couple of bugs in the code because it was written for an older version of Tensorflow that wasn't specified in the documentation. After hunting down, finding, and adapting a couple of lines of code to work with newer versions and downgrading to a version that would work, I got past this and it worked fine.

Dataset

The LISA dataset had its own method of annotating images that differed from how the YOLOv3 model expected. Because of this, I had to create a few python functions to translate and format the annotations into the following format right x center, y center, width, height and scaled from [0,1] from their original annotations in the form of x min, y min, x max, y max with pixel coordinates.

Datahub

Occasionally the server would kill my process in the middle of training, which would be pretty frustrating hours in and I would lose much of my progress. Luckily, by saving checkpoints at regular intervals, this work would not be lost and I could restart training from there. Additionally, changing up parts of the code that took long to run, helped the server not try to kill the process.

One of the most frustrating issues that I ran into with Datahub was that while trying to download the COCO dataset, which is massive about 13+ GB, my account ran out of disk space quota and I couldn't even access the machine through Jupyter Hub any longer. Luckily, after contacting ITS, they said I could just ssh directly in and delete the offending files. Doing that, the machines were accessible through Jupyter Hub again.

Overall Datahub has been fantastic and great to use, but there have been some difficulties to get past. It is definitely much better than running stuff purely through ssh and command line.

Improvements

Dataset

There are two ways in which I want to improve the dataset: variance and size. Currently, there are 82 images which are all pretty similar to each other due to being pulled from the same video segment. We can simply find more photos on the internet and other datasets, convert them to grayscale and add them to our dataset. This will help greatly with the variance and size of the dataset. One dataset that I have in mind from which I could pull images is the COCO Object Detection dataset and plus they multiple versions from different years. So there's a lot of images that I can pull from, convert to grayscale, and to the current dataset.

Other Object Detection Models

	backbone	AP	AP_{50}	AP_{75}	AP_S	AP_M	AP_L
Two-stage methods							
Faster R-CNN+++ [3]	ResNet-101-C4	34.9	55.7	37.4	15.6	38.7	50.9
Faster R-CNN w FPN [6]	ResNet-101-FPN	36.2	59.1	39.0	18.2	39.0	48.2
Faster R-CNN by G-RMI [4]	Inception-ResNet-v2 [19]	34.7	55.5	36.7	13.5	38.1	52.0
Faster R-CNN w TDM [18]	Inception-ResNet-v2-TDM	36.8	57.7	39.2	16.2	39.8	**52.1**
One-stage methods							
YOLOv2 [13]	DarkNet-19 [13]	21.6	44.0	19.2	5.0	22.4	35.5
SSD513 [9, 2]	ResNet-101-SSD	31.2	50.4	33.3	10.2	34.5	49.8
DSSD513 [2]	ResNet-101-DSSD	33.2	53.3	35.2	13.0	35.4	51.1
RetinaNet [7]	ResNet-101-FPN	39.1	59.1	42.3	21.8	42.7	50.2
RetinaNet [7]	ResNeXt-101-FPN	**40.8**	**61.1**	**44.1**	**24.1**	**44.2**	51.2
YOLOv3 608 × 608	Darknet-53	33.0	57.9	34.4	18.3	35.4	41.9

Figure 11. Comparison of Object Detection Models [3]

Single Shot Detector

This architecture developed by Google researchers combines region proposals and object classification into one network. It's AP50 Score is worse than YOLOv3 in this study, but I think it would still be interesting to see how it does at grayscale stop sign detection with a limited dataset.

Faster RCNN

According to Figure 11, Faster RCNN has the best AP50 score of 59.1 which is greater than YOLOv3's score of 57.9. Since it has a better score in this study, I would imagine it would do better than YOLOv3 in the grayscale stop sign detection task as well and be would worth investigating.

Multivariate Gaussian Classifier

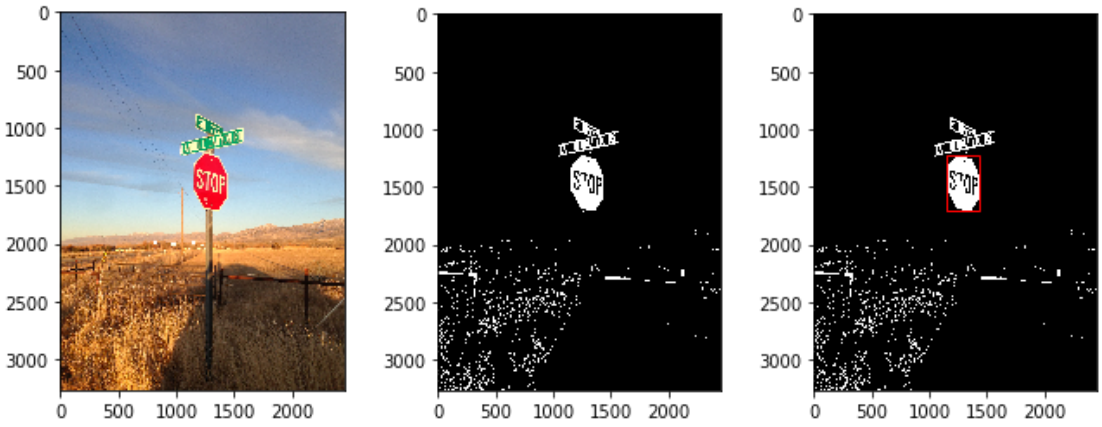

Figure 12. Multivariate Gaussian Classifier Example

In contrast to deep learning methods, we could also try using a multivariate Gaussian classifier to first detect stop sign pixels, then using shape similarity techniques we could identify stop sign shapes. This could be a more straight forward approach with us understanding how each step of the process works. However, this is really fragile when encountering very different lighting conditions in addition to occlusion. One stronger point to not use this more traditional method is that the time to fine-tune these parameters by hand, may not be worth it as it won't be as robust as modern object detection networks such as YOLOv3, SSD, and Faster RCNN.

Exploration

Detecting More Traffic Signs

Figure 13. YOLOv3 Detecting Various Traffic Signs

Another aspect that would be interesting to explore after improving stop sign detection with YOLOv3 is to expand the classes of objects to other traffic signs. This would be pretty easy since the dataset already includes a multitude of labeled traffic signs and we already have a conversion script. This makes a lot of sense, I would want to use this model in the area of autonomous vehicles.

Conclusion

The YOLOv3 network is certainly very capable of performing this task and could be a viable model of choice to be deployed in a real-time system such as an autonomous vehicle. It's incredible inference speed while maintaining accuracy has been shown with the training set and its results on the COCO dataset. The dataset of stop sign images just needs to be improved and I'm sure the model will perform much better.

References

1. YOLOv3 PyTorch Implementation, https://github.com/eriklindernoren/PyTorch-YOLOv3

2. Andreas Møgelmose, Mohan M. Trivedi, and Thomas B. Moeslund, "Vision based Traffic Sign Detection and Analysis for Intelligent Driver Assistance Systems: Perspectives and Survey," IEEE Transactions on Intelligent Transportation Systems, 2012.
3. Object Detection AP Chart.
https://medium.com/@jonathan_hui/object-detection-speed-and-accuracy-comparison-faster-r-cnn-r-fcn-ssd-and-yolo-5425656ae359